OpenShift Container Platform 4.15

BuildConfig を使用してビルドする

ビルド

Last Updated: 2024-05-14

OpenShift Container Platform 4.15 BuildConfig を使用してビルドする

ビルド

法律上の通知

概要

OpenShift Container Platform 用のビルド

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 Container Platform 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 Container Platform 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、パイプライン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

OpenShift Container Platform は Buildah を使用して Dockerfile からコンテナーイメージをビルドします。Dockerfile を使用したコンテナーイメージのビルドについての詳細は、Dockerfile リファレンスドキュメント を参照してください。

ヒント

buildArgs 配列を使用して Docker ビルド引数を設定する場合は、Dockerfile リファレンスドキュメントの ARG および FROM の対話方法 について参照してください。

1.1.2. Source-to-Image ビルド

Source-to-Image (S2I) は再現可能な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、ビルダーおよび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。

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。

カスタムビルドは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. パイプラインビルド

重要

パイプラインビルドストラテジーは OpenShift Container Platform 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 Container Platform Pipeline にあります。

OpenShift Container Platform 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、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、パイプラインビルドストラテジーを利用して Jenkins パイプラインプラグインで使用できるように Jenkins パイプラインを定義することができます。このビルドについては、他のビルドタイプの場合と同様に OpenShift Container Platform での起動、モニタリング、管理が可能です。

パイプラインワークフローは、ビルド設定に直接組み込むか、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、ビルド設定の概念を定義し、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セットを記述します。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 Container Platform 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** を設定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リストを指定できます。
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 (S2I) 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 Cotainer Platform 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、シークレット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)
    ref: "master"
  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ルートディレクトリーに 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。

関連情報

- 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イメージストリームタグ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
    ref: "master"
  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

クラスターで **ImageDigestMirrorSet**、**ImageTagMirrorSet**、または **ImageContentSourcePolicy** オブジェクトを使用してリポジトリーミラーリングを設定する場合、ミラーリングされたレジストリーにはグローバルプルシークレットのみを使用できます。プロジェクトにプルシークレットを追加することはできません。

プルシークレットを必要とするイメージ

プルシークレットを必要とするインプットイメージを使用する場合には、プルシークレットをビルドで使用されるサービスアカウントにリンクできます。デフォルトで、ビルドは **builder** サービスアカウントを使用します。シークレットにインプットイメージをホストするリポジトリーに一致する認証情報が含まれる場合、プルシークレットはビルドに自動的に追加されます。プルシークレットをビルドで使用されるサービスアカウントにリンクするには、以下を実行します。

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

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。

プルシークレットを必要とするミラーリングされたレジストリーのイメージ

ミラーリングされたレジストリーからインプットイメージを使用する場合、**build error: failed to pull image** メッセージが表示される場合、以下のいずれかの方法を使用してエラーを解決できます。

- ビルダーイメージのリポジトリーおよびすべての既知のミラーの認証情報が含まれる入力シークレットを作成します。この場合、イメージレジストリーおよびそのミラーに対する認証情報のプルシークレットを作成します。
- 入力シークレットを **BuildConfig** オブジェクトのプルシークレットとして使用します。

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 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 (Uniform Resource Identifier) が含まれます。特定の Git リファレンスをチェックアウトするには、**ref** フィールドの値を指定する必要があります。SHA1 タグまたはブランチ名は、**ref** として有効です。**ref** フィールドのデフォルト値は **master** です。

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 Container Platform 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: **main**)。

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ビルド設定の **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"
    ref: "master"
    httpProxy: http://proxy.example.com
    httpsProxy: https://proxy.example.com
    noProxy: somedomain.com, otherdomain.com
```

注記

パイプライン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。

関連情報

- 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. ソースクローンのシークレット

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。

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。

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 Container Platform 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、作成されるビルドが参照されるシークレット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シークレットが **BuildConfig** が後に作成される namespace になければなりません。このシークレットには、接頭辞 **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 (Uniform Resource Identifier) パターンを使用します。これは以下のように定義されます。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 がシークレットのアノテーションの URI パターンと一致する場合に、OpenShift Container Platform はそのシークレットへの参照を **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: *` または有効なホスト名、あるいは ***.** が先頭に指定された IP アドレス
- パス: /* または、 / の後に * 文字などの文字がオプションで後に続きます。

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

複数のシークレットが特定の **BuildConfig** の Git URI と一致する場合は、OpenShift Container Platform 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、作成したシークレットの名前に設定して実行できます。この例では **basicsecret** です。

```
apiVersion: "build.openshift.io/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、以下のコマンドを実行します。

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

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 認証情報が必要です。

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、以下のように **[http]** セクションを追加します。

```
# cat .gitconfig
```

出力例

```
[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 Source-to-Image (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。

関連情報

- 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 (software configuration management) サーバーに対して認証する場合に **--username** と **--password** の組み合わせ、またはトークンが必要です。

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. シークレット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 **--username** および **--password** 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 ed25519 -C "your_email@example.com"
```

注記

SSH キーのパスフレーズを作成すると、OpenShift Container Platform 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> \
    --from-file=<path/to/known_hosts> \ 1
    --type=kubernetes.io/ssh-auth
```

1 オプション: このフィールドを追加すると、厳密なサーバーホストキーチェックが有効になります。

警告

シークレットの作成中に **known_hosts** ファイルをスキップすると、ビルドが中間者 (MITM) 攻撃を受ける可能性があります。

注記

know_hosts ファイルにソースコードのホストのエントリーが含まれていることを確認してください。

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 (Transport Layer Security) 認証局 (CA) のセットは OpenShift Container Platform 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 Container Platform 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。

1. 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。以下のコマンドを入力します。

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

   a. 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** ファイルを使用して SSH ベースの認証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=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 ファイルおよび認証局 (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て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=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 (certificate authority) 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用して Basic 認証のシークレットを作成し、以下を実行します。

```
$ 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** ファイルで Basic 認証の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-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

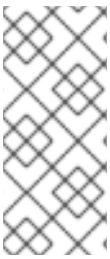
- **.gitconfig** ファイルと CA 証明書ファイルを合わせて Basic 認証シークレットを作成するには、以下を実行します。

```
$ oc create secret generic <secret_name> \
    --from-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。これは、バイナリーファイルを提供することができないためです。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デプロイメントします。**--from-dir** を使用して、デプロイメント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デプロイメントされます。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。Ddockerfile 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デプロイメント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設定マップ

重要

入力シークレットおよび設定マップのコンテンツがビルドの出力コンテナーイメージに表示されないようにするには、Docker build と source-to-image build ストラテジーでビルドボリュームを使用します。

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入力設定マップ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用してアクセスされる 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. シークレットの概要

Secret オブジェクトタイプはパスワード、OpenShift Container Platform 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: <username> (3)
  password: <password>
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。値は **data** フィールドによってのみ返されます。

(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設定されます。

3.6.1.1. 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。

3.6.1.2. 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

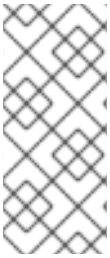
example.com/my-secret-type などの他の任意のタイプを指定できます。これらのタイプはサーバー側では実行されませんが、シークレットの作成者がその種類のキー/値の要件に従う意図があることを示します。

3.6.1.3. シークレットの更新

シークレットの値を変更する場合、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。コントローラーが古い **resourceVersion** を使用して Pod を再起動できるように、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。

3.6.2. 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。

YAML の不透明なシークレットオブジェクトの定義

```
apiVersion: v1
kind: Secret
metadata:
  name: mysecret
type: Opaque 1
data:
  username: <username>
  password: <password>
```

1 **opaque** シークレットを指定します。

Docker 設定の JSON ファイルシークレットオブジェクトの定義

```
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

3.6.3. 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
```

関連情報

- 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: <username> (1)
  password: <password> (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を設定するビルド設定の YAML

```
apiVersion: build.openshift.io/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
```

3.6.4. 入力シークレットおよび設定マップの追加

認証情報およびその他の設定データをソース管理に配置せずにビルドに提供するには、入力シークレットおよび入力設定マップを定義します。

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。この情報をソース管理に配置せずに利用可能にするには、入力シークレットおよび入力設定マップを定義します。

手順

既存の **BuildConfig** オブジェクトに入力シークレットおよび/または設定マップを追加するには、以下を行います。

1. **ConfigMap** オブジェクトがない場合はこれを作成します。

```
$ 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設定マップが作成されます。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。

ヒント

または、以下の YAML を適用して設定マップを作成できます。

```
apiVersion: core/v1
kind: ConfigMap
metadata:
  name: settings-mvn
data:
  settings.xml: |
    <settings>
    … # Insert maven settings here
    </settings>
```

2. **Secret** オブジェクトがない場合はこれを作成します。

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

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。

ヒント

または、以下の YAML を適用して入力シークレットを作成できます。

```
apiVersion: core/v1
kind: Secret
metadata:
  name: secret-mvn
type: kubernetes.io/ssh-auth
data:
  ssh-privatekey: |
    # Insert ssh private key, base64 encoded
```

3. 設定マップおよびシークレットを既存の **BuildConfig** オブジェクトの **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設定マップを新規の **BuildConfig** オブジェクトに追加するには、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 Container Platform 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** オブジェクトの作成時に、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。

3.6.5. 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しますが、セキュリティーの理由で空になります。

入力設定マップは、**assemble** スクリプトの実行後に切り捨てられません。

3.6.6. 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 の場所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設定マップデータを参照する 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"]
```

重要

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します。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ます。

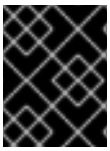
入力シークレットおよび設定マップのコンテンツがビルド出力コンテナーイメージに表示されないようにして、この削除プロセスを完全に回避するには、代わりに Docker ビルドストラテジーで ビルドボリュームを使用 します。

3.6.7. 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設定マップはすべて、**/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設定マップを適切に使用する必要があります。Custom ストラテジーでは、Custom ストラテジーのオプションで説明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。



重要

レジストリーのプルシークレットが namespace とノードの両方に存在する場合、ビルドがデフォルトで namespace でのプルシークレットの使用に設定されます。

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" ]
```

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イメージ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。

同じレジストリー内に、レジストリーパスに固有の認証情報を指定して、複数のリポジトリーに認証情報を指定できます。

注記

OpenShift Container Platform コンテナーイメージレジストリーでは、OpenShift Container Platform 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:
  index.docker.io/v1/: 1
    auth: "YWRfbGzhcGU6R2labnRib21ifTE=" 2
    email: "user@example.com" 3
```

```
    docker.io/my-namespace/my-user/my-image: 4
      auth: "GzhYWRGU6R2fbclabnRgbkSp=""
      email: "user@example.com"
    docker.io/my-namespace: 5
      auth: "GzhYWRGU6R2deesfrRgbkSp=""
      email: "user@example.com"
```

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
4. namespace 内の特定イメージの URL および認証情報
5. レジストリー namespace の URL および認証情報

複数のコンテナーイメージレジストリーを定義するか、同じレジストリーに複数のリポジトリーを定義することができます。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** ファイルが必要です。

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. **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
```

pushSecret フィールドを指定する代わりに、プッシュシークレットをビルドで使用されるサービスアカウントにリンクできます。デフォルトで、ビルドは **builder** サービスアカウントを使用します。シークレットにビルドのアウトプットイメージをホストするリポジトリーに一致する認証情報が含まれる場合、プッシュシークレットはビルドに自動的に追加されます。

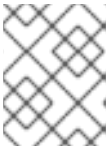
```
$ oc secrets link builder dockerhub
```

3. 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。

pullSecret フィールドを指定する代わりに、プルシークレットをビルドで使用されるサービスアカウントにリンクできます。デフォルトで、ビルドは **builder** サービスアカウントを使用します。シークレットにビルドのインプットイメージをホストするリポジトリーに一致する認証情報が含まれる場合、プルシークレットはビルドに自動的に追加されます。**pullSecret** フィールドを指定する代わりに、プルシークレットをビルドで使用されるサービスアカウントにリンクするには、以下を実行します。

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

注記

この機能を使用するには、**from** イメージを **BuildConfig** 仕様に指定する必要があります。**oc new-build** または **oc new-app** で生成される Docker ストラテジービルドは、場合によってこれを実行しない場合があります。

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。

重要

この方法では、シークレットをビルド Pod コンソールの出力でプレーンテキストとして表示します。これを回避するには、代わりに入力シークレットおよび設定マップを使用します。

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: build.openshift.io/v1
kind: BuildConfig
metadata:
  name: secret-example-bc
spec:
  strategy:
    sourceStrategy:
      env:
      - name: MYVAL
        valueFrom:
          secretKeyRef:
            key: myval
            name: mysecret
```

関連情報

- 入力シークレットおよび設定マップ

3.10. 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.beta.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.beta.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** は外部にルーティング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** ファイルの認証局 (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。

3.11. 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 イメージ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リストの一部になります。

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、カスタムラベルのリストを指定することも可能です。 このラベルは、ビルド設定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

OpenShift Container Platform は Buildah を使用して Dockerfile からコンテナーイメージをビルドします。Dockerfile を使用したコンテナーイメージのビルドについての詳細は、Dockerfile リファレンスドキュメント を参照してください。

ヒント

buildArgs 配列を使用して Docker ビルド引数を設定する場合は、Dockerfile リファレンスドキュメントの ARG および FROM の対話方法 について参照してください。

5.1.1. Dockerfile FROM イメージの置き換え

Dockerfile の **FROM** 命令は、**BuildConfig** オブジェクトの **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 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**) 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、環境変数をビルド設定の **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、ビルド設定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。

ヒント

Dockerfile リファレンスドキュメントの Understand how ARG and FROM interact を参照してください。

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** オブジェクトの **dockerStrategy** 定義の中にあります。以下に例を示します。

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

注記

name および **value** フィールドのみがサポートされます。**valueFrom** フィールドの設定は無視されます。

5.1.5. Docker ビルドによる層の非表示

Docker ビルドは通常、Dockerfile のそれぞれの命令を表す層を作成します。**imageOptimizationPolicy** を **SkipLayers** に設定することにより、すべての命令がベースイメージ上部の単一層にマージされます。

手順

- **imageOptimizationPolicy** を **SkipLayers** に設定します。

```
strategy:
  dockerStrategy:
    imageOptimizationPolicy: SkipLayers
```

5.1.6. ビルドボリュームの使用

ビルドボリュームをマウントして、実行中のビルドに、アウトプットコンテナーイメージで永続化しない情報にアクセスできます。

ビルドボリュームは、ビルド時にビルド環境や設定が必要なリポジトリーの認証情報など、機密情報のみを提供します。ビルドボリュームは、データが出力コンテナーイメージに保持される ビルド入力 とは異なります。

実行中のビルドがデータを読み取るビルドボリュームのマウントポイントは機能的に pod volume mounts に似ています。

前提条件

- 入力シークレット、設定マップ、またはその両方を BuildConfig オブジェクトに追加している。

手順

- **BuildConfig** オブジェクトの **dockerStrategy** 定義で、ビルドボリュームを **volumes** 配列に追加します。以下に例を示します。

```
spec:
  dockerStrategy:
    volumes:
      - name: secret-mvn (1)
        mounts:
        - destinationPath: /opt/app-root/src/.ssh (2)
        source:
          type: Secret (3)
          secret:
            secretName: my-secret (4)
      - name: settings-mvn (5)
        mounts:
        - destinationPath: /opt/app-root/src/.m2 (6)
        source:
          type: ConfigMap (7)
          configMap:
            name: my-config (8)
      - name: my-csi-volume (9)
        mounts:
        - destinationPath: /opt/app-root/src/some_path (10)
        source:
          type: CSI (11)
          csi:
            driver: csi.sharedresource.openshift.io (12)
```

```
        readOnly: true (13)
        volumeAttributes: (14)
          attribute: value
```

(1) (5) (9) 必須。一意な名前

(2) (6) (10) 必須。マウントポイントの絶対パス。**..** または **:** を含めないでください。こうすることで、ビルダーが生成した宛先パスと競合しなくなります。**/opt/app-root/src** は、多くの Red Hat S2I 対応イメージのデフォルトのホームディレクトリーです。

(3) (7) (11) 必須。ソースのタイプは、**ConfigMap**、**Secret**、または **CSI**。

(4) (8) 必須。ソースの名前。

(12) 必須。一時 CSI ボリュームを提供するドライバー。

(13) 必須。この値は **true** に設定する必要があります。読み取り専用ボリュームを提供します。

(14) オプション:一時 CSI ボリュームのボリューム属性。サポートされる属性キーおよび値については、CSI ドライバーのドキュメントを参照してください。

注記

共有リソース CSI ドライバーは、テクノロジープレビュー機能としてサポートされています。

5.2. SOURCE-TO-IMAGE ビルド

Source-to-Image (S2I) は再現可能な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、ビルダーおよび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。

5.2.1. Source-to-Image (S2I) 増分ビルドの実行

Source-to-Image (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これを作成します。

```
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

関連情報

- 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** Source-to-Image (S2I) スクリプトを上書きできます。

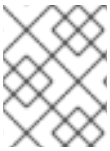
手順

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。

- 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。
- 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。以下に例を示します。

```
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 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 つの方法があります。2 種類 (環境ファイルおよび BuildConfig 環境の値の使用) 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 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、Source-to-Image (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、ビルド中の 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。

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 ビルド設定環境の使用

環境変数をビルド設定の **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"
```

関連情報

- 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、ビルド設定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 ソースファイルを無視する

Source-to-Image (S2I) は **.s2iignore** ファイルをサポートします。これには、無視する必要のあるファイルパターンのリストが含まれます。このファイルには、無視すべきファイルパターンのリストが含まれます。 **.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。

5.2.5. Source-to-Image 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. Source-to-Image ビルドプロセスについて

ビルドプロセスは次の 3 つの基本要素で構成されます。これら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
- Source-to-Image (S2I) スクリプト
- ビルダーイメージ

S2I は、最初の **FROM** 命令として、ビルダーイメージで Dockerfile を生成します。S2I によって生成される Dockerfile は Buildah に渡されます。

5.2.5.2. Source-to-Image スクリプトの作成方法

Source-to-Image (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. ビルド設定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. **io.openshift.s2i.scripts-url** ラベルを含むデフォルトの URL 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、ビルド設定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	**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。このスクリプトが必要です。このスクリプトのワークフローは以下のとおりです。 1. オプション: ビルドのアーティファクトを復元します。増分ビルドをサポートする必要がある場合、**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	**run** スクリプトはアプリケーションを実行します。このスクリプトが必要です。
save-artifacts	**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。このスクリプトはオプションです。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	**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。このスクリプトはオプションです。
test/run	**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プロセスを作成できます。このスクリプトはオプションです。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. オプション: 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。 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。  **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ための推奨される場所は、イメージリポジトリーの **test/test-app** ディレクトリーです。

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** ディレクトリーにデプロイメントされることが前提とされています。

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
```

run スクリプト:

```
#!/bin/bash

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

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
```

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
```

関連情報

- S2I イメージ作成のチュートリアル

5.2.6. ビルドボリュームの使用

ビルドボリュームをマウントして、実行中のビルドに、アウトプットコンテナーイメージで永続化しない情報にアクセスできます。

ビルドボリュームは、ビルド時にビルド環境や設定が必要なリポジトリーの認証情報など、機密情報のみを提供します。ビルドボリュームは、データが出力コンテナーイメージに保持される ビルド入力 とは異なります。

実行中のビルドがデータを読み取るビルドボリュームのマウントポイントは機能的に pod volume mounts に似ています。

前提条件

- 入力シークレット、設定マップ、またはその両方を BuildConfig オブジェクトに追加している。

手順

- **BuildConfig** オブジェクトの **sourceStrategy** 定義で、ビルドボリュームを **volumes** 配列に追加します。以下に例を示します。

```
spec:
  sourceStrategy:
    volumes:
      - name: secret-mvn (1)
        mounts:
        - destinationPath: /opt/app-root/src/.ssh (2)
        source:
          type: Secret (3)
          secret:
            secretName: my-secret (4)
      - name: settings-mvn (5)
        mounts:
        - destinationPath: /opt/app-root/src/.m2 (6)
        source:
          type: ConfigMap (7)
          configMap:
            name: my-config (8)
      - name: my-csi-volume (9)
        mounts:
        - destinationPath: /opt/app-root/src/some_path (10)
        source:
          type: CSI (11)
          csi:
            driver: csi.sharedresource.openshift.io (12)
            readOnly: true (13)
            volumeAttributes: (14)
              attribute: value
```

1 5 9 必須。一意な名前

2 6 10 必須。マウントポイントの絶対パス。**..** または **:** を含めないでください。こうすることで、ビルダーが生成した宛先パスと競合しなくなります。**/opt/app-root/src** は、多くの Red Hat S2I 対応イメージのデフォルトのホームディレクトリーです。

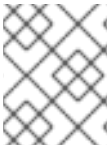
3 7 11 必須。ソースのタイプは、**ConfigMap**、**Secret**、または **CSI**。

4 8 必須。ソースの名前。

12 必須。一時 CSI ボリュームを提供するドライバー。

13 必須。この値は **true** に設定する必要があります。読み取り専用ボリュームを提供します。

14 オプション:一時 CSI ボリュームのボリューム属性。サポートされる属性キーおよび値については、CSI ドライバーのドキュメントを参照してください。



注記

共有リソース CSI ドライバーは、テクノロジープレビュー機能としてサポートされています。

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。

カスタムビルドは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リストをビルダー 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、**strategy** YAML ファイルの **secretSource** および **mountPath** フィールドを編集します。

```
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、環境変数をビルド設定の **customStrategy** 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

1. ビルド時に使用されるカスタムの HTTP プロキシーを定義します。

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

2. ビルド設定で定義された環境変数を管理するには、以下のコマンドを入力します。

```
$ oc set env <enter_variables>
```

5.3.4. カスタムビルダーイメージの使用

OpenShift Container Platform のカスタムビルドストラテジーにより、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。パッケージ、JAR、WAR、インストール可能な ZIP、ベースイメージなどの個別のアーティファクトを生成するためにビルドが必要な場合は、カスタムビルドストラテジーを使用してカスタムビルダーイメージを使用します。

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。

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。
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
PUSH_DOCKERCFG_PATH	**podman 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 Container Platform 内でビルドを実行す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。

5.4. パイプラインビルド

重要

パイプラインビルドストラテジーは OpenShift Container Platform 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 Container Platform Pipeline にあります。

OpenShift Container Platform 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、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、パイプラインビルドストラテジーを利用して Jenkins パイプラインプラグインで使用できるように Jenkins パイプラインを定義することができます。このビルドについては、他のビルドタイプの場合と同様に OpenShift Container Platform での起動、モニタリング、管理が可能です。

パイプラインワークフローは、ビルド設定に直接組み込むか、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 Container Platform Pipeline について

重要

パイプラインビルドストラテジーは OpenShift Container Platform 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 Container Platform Pipeline にあります。

OpenShift Container Platform 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、これをソースコントロール管理システムに保存します。

Pipeline により、OpenShift Container Platform 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 Container Platform 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 Container Platform Jenkins 同期プラグイン

OpenShift Container Platform Jenkins 同期プラグインは、ビルド設定およびビルドオブジェクトを Jenkins ジョブおよびビルドと同期し、以下を提供します。

- Jenkins での動的なジョブおよび実行の作成。
- イメージストリーム、イメージストリームタグまたは設定マップからのエージェント Pod テンプレートの動的作成。
- 環境変数の挿入。
- OpenShift Container Platform Web コンソールでのパイプラインの可視化。
- Jenkins Git プラグインとの統合。これにより、OpenShift Container Platform ビルドからの Jenkins Git プラグインにコミット情報が渡されます。
- シークレットを Jenkins 認証情報エントリーに同期。

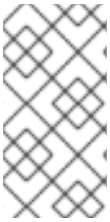
OpenShift Container Platform Jenkins クライアントプラグイン

OpenShift Container Platform Jenkins Client プラグインは、OpenShift Container Platform 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) スタイルの Jenkins Pipeline 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 Container Platform コマンドラインツール (**oc**) を使用します。

OpenShift 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 Container Platform 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 Container Platform 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 Container Platform 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。

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



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. パイプラインビルド用の Jenkins ファイルの提供

重要

パイプラインビルドストラテジーは OpenShift Container Platform 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 Container Platform Pipeline にあります。

OpenShift Container Platform 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、これをソースコントロール管理システムに保存します。

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デフォルト設定されます。

jenkinsfile は Jenkins エージェント Pod で実行されます。 ここでは OpenShift Container Platform DSL を使用する場合に OpenShift Container Platform 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

Jenkins ファイルを指定するには、以下のいずれかを実行できます。

- ビルド設定に Jenkins ファイルを埋め込む
- Jenkins ファイルを含む 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

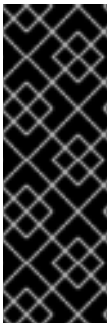
```
kind: "BuildConfig"
apiVersion: "v1"
metadata:
  name: "sample-pipeline"
spec:
  strategy:
    jenkinsPipelineStrategy:
      jenkinsfile: |-
        node('agent') {
          stage 'build'
          openshiftBuild(buildConfig: 'ruby-sample-build', showBuildLogs: 'true')
          stage 'deploy'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



重要

パイプラインビルドストラテジーは OpenShift Container Platform 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 Container Platform Pipeline にあります。

OpenShift Container Platform 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、これをソースコントロール管理システムに保存します。

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

定義した後に、環境変数はビルド設定に関連する 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、YAML ファイルを編集します。

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

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

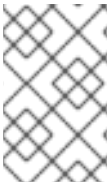
5.4.3.1. BuildConfig 環境変数と Jenkins ジョブパラメーター間のマッピング

Pipeline ストラテジーのビルド設定への変更に従い、Jenkins ジョブが作成/更新されると、ビルド設定の環境変数は 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、ビルド設定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、ビルド設定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。ビルド設定値が優先されます。
- **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。
    - ビルド設定にリスト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。ビルド設定および **oc start-build -e** で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、ビルド設定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

重要

パイプラインビルドストラテジーは OpenShift Container Platform 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 Container Platform Pipeline にあります。

OpenShift Container Platform 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、これをソースコントロール管理システムに保存します。

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 Container Platform 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>
```

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

```
$ oc new-app jenkins-ephemeral 1
```

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。

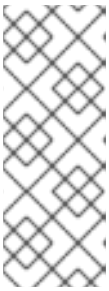
2. 以下の内容で **nodejs-sample-pipeline.yaml** という名前のファイルを作成します。

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** オブジェクトを作成します。

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

3. **jenkinsPipelineStrategy** で **BuildConfig** オブジェクトを作成したら、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 Container Platform 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

1 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** のエージェント 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パイプラインのビルド設定は、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、宣言型のパイプラインスタイルを使用して記述されていますが、以前のスクリプト化されたパイプラインスタイルもサポートされます。

4. OpenShift Container Platform 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
```

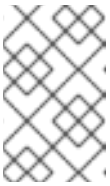
注記

または、OpenShift Container Platform 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、エージェント Pod が起動される

- Pipeline がエージェント Pod で実行されるか、エージェント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。
- パイプライン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 Container Platform 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

OpenShift Container Platform Web コンソールから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 Container Platform 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** 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、ビルド設定にプルシークレットを設定し、プッシュします。

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。

- ビルド設定にプルシークレットを設定します。

プッシュを有効にするには、以下を実行します。

- ビルド設定にプッシュシークレットを設定します。

第6章 BUILDAH によるカスタムイメージビルド

OpenShift Container Platform 4.15 では、ホストノードに docker ソケットは存在しません。これは、カスタムビルドの **mount docker socket** オプションがカスタムビルドイメージ内で使用できる docker ソケットを提供しない可能性がゼロではないことを意味します。

イメージのビルドおよびプッシュにこの機能を必要とする場合、Buildah ツールをカスタムビルドイメージに追加し、これを使用してカスタムビルドロジック内でイメージをビルドし、プッシュします。以下の例は、Buildah でカスタムビルドを実行する方法を示しています。

注記

カスタムビルドストラテジーを使用するためには、デフォルトで標準ユーザーが持たないパーミッションが必要です。このパーミッションはユーザーがクラスターで実行される特権付きコンテナー内で任意のコードを実行することを許可します。このレベルのアクセスを使用するとクラスターが危険にさらされる可能性があるため、このアクセスはクラスターで管理者権限を持つ信頼されたユーザーのみに付与される必要があります。

6.1. 前提条件

- カスタムビルドパーミッションを付与する 方法について確認してください。

6.2. カスタムビルドアーティファクトの作成

カスタムビルドイメージとして使用する必要のあるイメージを作成する必要があります。

手順

1. 空のディレクトリーからはじめ、以下の内容を含む **Dockerfile** という名前のファイルを作成します。

```
FROM registry.redhat.io/rhel8/buildah
# In this example, `/tmp/build` contains the inputs that build when this
# custom builder image is run. Normally the custom builder image fetches
# this content from some location at build time, by using git clone as an example.
ADD dockerfile.sample /tmp/input/Dockerfile
ADD build.sh /usr/bin
RUN chmod a+x /usr/bin/build.sh
# /usr/bin/build.sh contains the actual custom build logic that will be run when
# this custom builder image is run.
ENTRYPOINT ["/usr/bin/build.sh"]
```

2. 同じディレクトリーに、**dockerfile.sample** という名前のファイルを作成します。このファイルはカスタムビルドイメージに組み込まれ、コンテンツビルドによって生成されるイメージを定義します。

```
FROM registry.access.redhat.com/ubi9/ubi
RUN touch /tmp/build
```

3. 同じディレクトリーに、**build.sh** という名前のファイルを作成します。このファイルには、カスタムビルドの実行時に実行されるロジックが含まれます。

```
#!/bin/sh
```

```
# Note that in this case the build inputs are part of the custom builder image, but normally this
# is retrieved from an external source.
cd /tmp/input
# OUTPUT_REGISTRY and OUTPUT_IMAGE are env variables provided by the custom
# build framework
TAG="${OUTPUT_REGISTRY}/${OUTPUT_IMAGE}"


# performs the build of the new image defined by dockerfile.sample
buildah --storage-driver vfs bud --isolation chroot -t ${TAG} .


# buildah requires a slight modification to the push secret provided by the service
# account to use it for pushing the image
cp /var/run/secrets/openshift.io/push/.dockercfg /tmp
(echo "{ \"auths\": " ; cat /var/run/secrets/openshift.io/push/.dockercfg ; echo "}") >
/tmp/.dockercfg


# push the new image to the target for the build
buildah --storage-driver vfs push --tls-verify=false --authfile /tmp/.dockercfg ${TAG}
```

6.3. カスタムビルダーイメージのビルド

OpenShift Container Platform を使用してカスタムストラテジーで使用するカスタムビルダーイメージをビルドし、プッシュすることができます。

前提条件

- 新規カスタムビルダーイメージの作成に使用されるすべての入力を定義します。

手順

1. カスタムビルダーイメージをビルドする **BuildConfig** オブジェクトを定義します。

```
$ oc new-build --binary --strategy=docker --name custom-builder-image
```

2. カスタムビルドイメージを作成したディレクトリーから、ビルドを実行します。

```
$ oc start-build custom-builder-image --from-dir . -F
```

ビルドの完了後に、新規のカスタムビルダーイメージが **custom-builder-image:latest** という名前のイメージストリームタグのプロジェクトで利用可能になります。

6.4. カスタムビルダーイメージの使用

カスタムビルダーイメージとカスタムストラテジーを併用する **BuildConfig** オブジェクトを定義し、カスタムビルドロジックを実行することができます。

前提条件

- 新規カスタムビルダーイメージに必要なすべての入力を定義します。

- カスタムビルダーイメージをビルドします。

手順

1. **buildconfig.yaml** という名前のファイルを作成します。このファイルは、プロジェクトに作成され、実行される **BuildConfig** オブジェクトを定義します。

```
kind: BuildConfig
apiVersion: build.openshift.io/v1
metadata:
  name: sample-custom-build
  labels:
    name: sample-custom-build
  annotations:
    template.alpha.openshift.io/wait-for-ready: 'true'
spec:
  strategy:
    type: Custom
    customStrategy:
      forcePull: true
      from:
        kind: ImageStreamTag
        name: custom-builder-image:latest
        namespace: <yourproject> 1
  output:
    to:
      kind: ImageStreamTag
      name: sample-custom:latest
```

1 プロジェクト名を指定します。

2. **BuildConfig** を作成します。

```
$ oc create -f buildconfig.yaml
```

3. **imagestream.yaml** という名前のファイルを作成します。このファイルはビルドがイメージをプッシュするイメージストリームを定義します。

```
kind: ImageStream
apiVersion: image.openshift.io/v1
metadata:
  name: sample-custom
spec: {}
```

4. imagestream を作成します。

```
$ oc create -f imagestream.yaml
```

5. カスタムビルドを実行します。

```
$ oc start-build sample-custom-build -F
```

ビルドが実行されると、以前にビルドされたカスタムビルダーイメージを実行する Pod が起動

します。Pod はカスタムビルダーイメージのエントリーポイントとして定義される **build.sh** ロジックを実行します。**build.sh** ロジックは Buildah を起動し、カスタムビルダーイメージに埋め込まれた **dockerfile.sample** をビルドしてから、Buildah を使用して新規イメージを **sample-custom image stream** にプッシュします。

第7章 基本的なビルドの実行および設定

以下のセクションでは、ビルドの開始および中止、**BuildConfigs** の編集、**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。

7.1. 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、以下のコマンドを入力します。

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

7.1.1. 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、以下のコマンドを入力します。

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

7.1.2. ビルドログのストリーミング

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

手順

- **stdout** でビルドのログを手動でストリーミングするには、以下のコマンドを実行します。

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

7.1.3. 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、以下のコマンドを実行します。

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

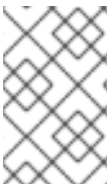
7.1.4. ソースを使用したビルドの開始

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

- 以下のコマンドを使用してソースからビルドを開始し、タグ **v2** からローカル Git リポジトリーの内容をアーカイブとして送信します。

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

7.2. 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

- 手動でビルドを取り消すには、以下のコマンドを入力します。

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

7.2.1. 複数ビルドのキャンセル

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

- 複数ビルドを手動で取り消すには、以下のコマンドを入力します。

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

7.2.2. すべてのビルドのキャンセル

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

- すべてのビルドを取り消すには、以下のコマンドを実行します。

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

7.2.3. 指定された状態のすべてのビルドのキャンセル

特定の状態にあるビルドをすべて取り消すことができます (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

- 特定の状態のすべてのビルドを取り消すには、以下のコマンドを入力します。

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

7.3. BUILDCONFIG の編集

ビルド設定を編集するには、Developer パースペクティブの Builds ビューで Edit BuildConfig オプションを使用します。

以下のいずれかのビューを使用して **BuildConfig** を編集できます。

- Form view を使用すると、標準のフォームフィールドおよびチェックボックスを使用して **BuildConfig** を編集できます。
- YAML ビュー を使用すると、操作を完全に制御して **BuildConfig** を編集できます。

データを失うことなく、Form view と YAML view を切り替えることができます。Form ビュー のデータは YAML ビュー に転送されます (その逆も同様です)。

手順

1. Developer パースペクティブの Builds ビューで、メニュー をクリックし、Edit BuildConfig オプションを表示します。
2. Edit BuildConfig をクリックし、Form view オプションを表示します。
3. Git セクションで、アプリケーションの作成に使用するコードベースの Git リポジトリー URL を入力します。その後、URL は検証されます。
    - オプション: Show Advanced Git Options をクリックし、以下のような詳細を追加します。
        - Git Reference: アプリケーションのビルドに使用するコードが含まれるブランチ、タグ、またはコミットを指定します。

- **Context Dir**: アプリケーションのビルドに使用するアプリケーションのコードが含まれるサブディレクトリーを指定します。
- **Source Secret**: プライベートリポジトリーからソースコードをプルするための認証情報で **Secret Name** を作成します。

4. **Build from** セクションで、ビルド元となるオプションを選択します。以下のオプションで使用できます。
   - **イメージストリームタグ** は、所定のイメージストリームおよびタグのイメージを参照します。ビルド元およびプッシュ元の場所に指定するプロジェクト、イメージストリーム、およびタグを入力します。
   - **イメージストリームイメージ** は、所定のイメージストリームのイメージとおよびイメージ名を参照します。ビルドするイメージストリームイメージを入力します。また、プッシュ先となるプロジェクト、イメージストリーム、およびタグも入力します。
   - **Docker image**: Docker イメージは Docker イメージリポジトリーを使用して参照されます。また、プッシュ先の場所を参照するように、プロジェクト、イメージストリーム、タグを入力する必要があります。
5. オプション: **環境変数** セクションで**Name** と **Value** フィールドを使用して、プロジェクトに関連付けられた環境変数を追加します。環境変数を追加するには、**Add Value** または **Add from ConfigMap** と **Secret** を使用します。
6. オプション: 以下の高度なオプションを使用してアプリケーションをさらにカスタマイズできます。

   トリガー

   ビルダーイメージの変更時に新規イメージビルドをトリガーします。**Add Trigger** をクリックし、**Type** および **Secret** を選択して、トリガーを追加します。

   シークレット

   アプリケーションのシークレットを追加します。**Add secret** をクリックし、**Secret** および **Mount point** を選択して、さらにシークレットを追加します。

   Policy

   Run policy をクリックして、ビルド実行ポリシーを選択します。選択したポリシーは、ビルド設定から作成されるビルドを実行する順番を決定します。

   フック

   Run build hooks after image is builtを選択して、ビルドの最後にコマンドを実行し、イメージを検証します。**Hook type**、**Command** および **Arguments** をコマンドに追加しあ m す。
7. **Save** をクリックして **BuildConfig** を保存します。

7.4. 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、以下のコマンドを入力します。

```
$ 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。

- **BuildConfig** を削除して、 **BuildConfig** からインスタンス化されたビルドを保持するには、以下のコマンドの入力時に **--cascade=false** フラグを指定します。

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

7.5. 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、以下のコマンドを入力します。

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

7.6. 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、以下のコマンドを入力します。

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

7.6.1. 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、以下のコマンドを入力します。

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

7.6.2. 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。

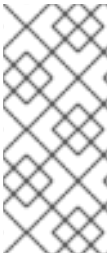
手順

- **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、以下のコマンドを入力します。

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

7.6.3. 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 Container Platform 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。これはデフォルトになります。
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リストを生成します。
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8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。

8.1. 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。

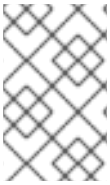
- Webhook
- イメージの変更
- 設定の変更

8.1.1. Webhook のトリガー

Webhook のトリガーにより、要求を OpenShift Container Platform 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 を使用してこれらのトリガーを定義できます。

OpenShift Container Platform の Webhook は現在、Git ベースのソースコード管理システム (SCM) システムのそれぞれのプッシュイベントの類似の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 Container Platform コントロールプレーンホストは、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 Container Platform 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。トリガーを設定して、トリガーを手動で追加できます。

Webhook すべてに対して、**WebHookSecretKey** という名前のキーでシークレットと、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
```

8.1.1.1. 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際に、シークレットを指定する必要があります。このシークレットは、Webhook の設定時に GitHub に指定する 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 の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 は以下のように設定されます。

出力例

```
https://<openshift_api_host:port>/apis/build.openshift.io/v1/namespaces/<namespace>/buildconfigs/<name>/webhooks/<secret>/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。

      出力例

```
<https://api.starter-us-east-
1.openshift.com:443/apis/build.openshift.io/v1/namespaces/<namespace>/buildconfigs/<na
me>/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。

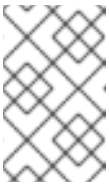
c. GitHub リポジトリーで、**Settings → Webhooks** から **Add Webhook** を選択します。

d. **Payload URL** フィールドに、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際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>/apis/build.openshift.io/v1/namespaces/<namespace>/buildcon
gs/<name>/webhooks/<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。

注記

ビルドは、GitHub Webhook イベントからの **ref** 値が、**BuildConfig** リソースの **source.git** フィールドで指定された **ref** 値と一致する場合にのみトリガーされます。

関連情報

- Gogs

8.1.1.2. 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、シークレット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。 ペイロード URL は以下のように設定されます。

出力例

```
https://<openshift_api_host:port>/apis/build.openshift.io/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>/apis/build.openshift.io/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。

8.1.1.3. Bitbucket Webhook の使用

Bitbucket webhook は、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、シークレット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。ペイロード URL は以下のように設定されます。

出力例

```
https://<openshift_api_host:port>/apis/build.openshift.io/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>/apis/build.openshift.io/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。

8.1.1.4. 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。このシークレットは、呼び出し元がビルドをトリガーするために使用する必要のある URL に追加されます。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: "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。

   出力例

   ```
   https://<openshift_api_host:port>/apis/build.openshift.io/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>/apis/build.openshift.io/v1/namespaces/<namespace>/buildcon
gs/<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。

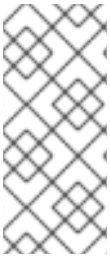
```
git:
  uri: "<url to git repository>"
  ref: "<optional git reference>"
  commit: "<commit hash identifying a specific git commit>"
  author:
    name: "<author name>"
    email: "<author e-mail>"
  committer:
    name: "<committer name>"
    email: "<committer e-mail>"
  message: "<commit message>"
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>/apis/build.openshift.io/v1/namespaces/<namespace>/buildcon
gs/<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 Container Platform 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 Container Platform は、**HTTP 200 OK** 応答の一部として JSON 形式で警告を返します。

8.1.1.5. Webhook URL の表示

以下のコマンドを使用して、ビルド設定に関連する webhook URL を表示できます。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。

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>
```

8.1.2. イメージ変更トリガーの使用

開発者は、ベースイメージが変更するたびにビルドを自動的に実行するように設定できます。

イメージ変更のトリガーを使用すると、アップストリームイメージ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、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 1 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

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 Container Platform で実行する名前が **docker-registry** の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、**ImageStream** 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. **ImageStreams** 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。

参照されるイメージストリームで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してからビルドを開始できます。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。

関連情報

- v1 コンテナーレジストリー

8.1.3. ビルドのイメージ変更トリガーの識別

開発者は、イメージ変更トリガーがある場合は、どのイメージの変更が最後のビルドを開始したかを特定できます。これは、ビルドのデバッグやトラブルシューティングに役立ちます。

BuildConfig の例

```
apiVersion: build.openshift.io/v1
kind: BuildConfig
metadata:
  name: bc-ict-example
  namespace: bc-ict-example-namespace
spec:

  # ...

  triggers:
  - imageChange:
      from:
        kind: ImageStreamTag
        name: input:latest
        namespace: bc-ict-example-namespace
  - imageChange:
      from:
        kind: ImageStreamTag
        name: input2:latest
        namespace: bc-ict-example-namespace
    type: ImageChange
status:
  imageChangeTriggers:
  - from:
      name: input:latest
      namespace: bc-ict-example-namespace
    lastTriggerTime: "2021-06-30T13:47:53Z"
    lastTriggeredImageID: image-registry.openshift-image-registry.svc:5000/bc-ict-example-
namespace/input@sha256:0f88ffbeb9d25525720bfa3524cb1bf0908b7f791057cf1acfae917b11266a69

  - from:
      name: input2:latest
      namespace: bc-ict-example-namespace
    lastTriggeredImageID:  image-registry.openshift-image-registry.svc:5000/bc-ict-example-
namespace/input2@sha256:0f88ffbeb9d25525720bfa3524cb2ce0908b7f791057cf1acfae917b11266a6
9

  lastVersion: 1
```

注記

この例では、イメージ変更トリガーに関係のない要素を省略します。

前提条件

- 複数のイメージ変更トリガーを設定している。これらのトリガーは 1 つまたは複数のビルドがトリガーされています。

手順

1. **buildConfig.status.imageChangeTriggers** で、最新のタイムスタンプを持つ **lastTriggerTime** を特定します。
   This **ImageChangeTriggerStatus**

   > Then you use the `name` and `namespace` from that build to find the corresponding image change trigger in `buildConfig.spec.triggers`.

2. **imageChangeTriggers** でタイムスタンプを比較して最新のものを特定します。

イメージ変更のトリガー

ビルド設定で、**buildConfig.spec.triggers** はビルドトリガーポリシー (**BuildTriggerPolicy**) の配列です。

各 **BuildTriggerPolicy** には **type** フィールドと、ポインターフィールドのセットがあります。各ポインターフィールドは、**type** フィールドに許可される値の 1 つに対応します。そのため、**BuildTriggerPolicy** を 1 つのポインターフィールドのみに設定できます。

イメージ変更のトリガーの場合、**type** の値は **ImageChange** です。次に、**imageChange** フィールドは、以下のフィールドを持つ **ImageChangeTrigger** オブジェクトへのポインターです。

- **lastTriggeredImageID**: このフィールドは例では提供されず、OpenShift Container Platform 4.8 で非推奨となり、今後のリリースでは無視されます。これには、最後のビルドがこの **BuildConfig** からトリガーされた際に **ImageStreamTag** の解決されたイメージ参照が含まれます。

- **paused**: このフィールドは、この例では示されていませんが、この特定のイメージ変更トリガーを一時的に無効にするのに使用できます。

- **from**: このフィールドを使用して、このイメージ変更トリガーを駆動する **ImageStreamTag** を参照します。このタイプは、コア Kubernetes タイプである **OwnerReference** です。

from フィールドには、注意フィールド **kind があります。イメージ変更トリガーの場合、サポートされる値は ImageStreamTag のみです。 namespace**: このフィールドを使用して **ImageStreamTag** の namespace を指定します。** **name**: このフィールドを使用して **ImageStreamTag** を指定します。

イメージ変更のトリガーのステータス

ビルド設定で、**buildConfig.status.imageChangeTriggers** は **ImageChangeTriggerStatus** 要素の配列です。それぞれの **ImageChangeTriggerStatus** 要素には、前述の例に示されている **from**、**lastTriggeredImageID**、および **lastTriggerTime** 要素が含まれます。

最新の **lastTriggerTime** を持つ **ImageChangeTriggerStatus** は、最新のビルドをトリガーしました。**name** および **namespace** を使用して、ビルドをトリガーした **buildConfig.spec.triggers** でイメージ変更トリガーを特定します。

lastTriggerTime は最新のタイムスタンプ記号で、最後のビルドの **ImageChangeTriggerStatus** を示します。この **ImageChangeTriggerStatus** には、ビルドをトリガーした **buildConfig.spec.triggers** のイメージ変更トリガーと同じ **name** および **namespace** があります。

関連情報

- v1 コンテナーレジストリー

8.1.4. 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。

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。

8.1.4.1. 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。

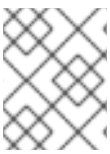
```
$ oc set triggers bc <name> --from-github
```

- 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、以下を実行してヘルプドキュメントを参照してください。

```
$ oc set triggers --help
```

8.2. 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。すべての場合に、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。

8.2.1. コミット後のビルドフックの設定

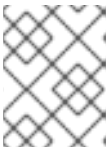
ビルド後のフックを設定する方法は複数あります。以下の例に出てくるすべての形式は同等で、**bundle exec rake test --verbose** を実行します。

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:

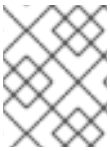
```
postCommit:
  command: ["/bin/bash", "-c", "bundle exec rake test --verbose"]
```

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。

8.2.2. 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9章 高度なビルドの実行

以下のセクションでは、ビルドリソースおよび最長期間の設定、ビルドのノードへの割り当て、チェーンビルド、ビルドのプルーニング、およびビルド実行ポリシーなどの高度なビルド操作について説明します。

9.1. ビルドリソースの設定

デフォルトでは、ビルドは、メモリーや CPU など、バインドされていないリソースを使用して Pod により完了されます。これらのリソースは制限できます。

手順

リソースの使用を制限する方法は 2 つあります。

- プロジェクトのデフォルトコンテナー制限でリソース制限を指定して、リソースを制限します。
- リソースの制限をビルド設定の一部として指定し、リソースの使用を制限します。** 以下の例では、**resources**、 **cpu**、および **memory** パラメーターはそれぞれオプションです。

```
apiVersion: "v1"
kind: "BuildConfig"
metadata:
  name: "sample-build"
spec:
  resources:
    limits:
      cpu: "100m" 1
      memory: "256Mi" 2
```

1 **cpu** は CPU のユニットで、 **100m** は 0.1 CPU ユニット (100 * 1e-3) を表します。

2 **memory** はバイト単位です。**256Mi** は 268435456 バイトを表します (256 * 2 ^ 20)。

ただし、クォータがプロジェクトに定義されている場合には、以下の 2 つの項目のいずれかが必要です。

- 明示的な **requests** で設定した **resources** セクション:

```
resources:
  requests: 1
    cpu: "100m"
    memory: "256Mi"
```

1 **requests** オブジェクトは、クォータ内のリソースリストに対応するリソースリストを含みます。

- プロジェクトに定義される制限範囲。 **LimitRange** オブジェクトからのデフォルト値がビルドプロセス時に作成される Pod に適用されます。
  適用されない場合は、クォータ基準を満たさないために失敗したというメッセージが出され、ビルド Pod の作成は失敗します。

9.2. 最長期間の設定

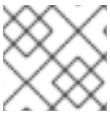
BuildConfig オブジェクトの定義時に、**completionDeadlineSeconds** フィールドを設定して最長期間を定義できます。このフィールドは秒単位で指定し、デフォルトでは設定されません。設定されていない場合は、最長期間は有効ではありません。

最長期間はビルドの Pod がシステムにスケジュールされた時点から計算され、ビルダーイメージをプルするのに必要な時間など、ジョブが有効である期間を定義します。指定したタイムアウトに達すると、ジョブは OpenShift Container Platform により終了されます。

手順

- 最長期間を設定するには、**BuildConfig** に **completionDeadlineSeconds** を指定します。以下の例は **BuildConfig** の一部で、**completionDeadlineSeconds** フィールドを 30 分に指定しています。

```
spec:
  completionDeadlineSeconds: 1800
```



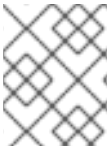
注記

この設定は、パイプラインストラテジーオプションではサポートされていません。

9.3. 特定のノードへのビルドの割り当て

ビルドは、ビルド設定の **nodeSelector** フィールドにラベルを指定して、特定のノード上で実行するようにターゲットを設定できます。**nodeSelector** の値は、ビルド Pod のスケジュール時の **Node** ラベルに一致するキー/値のペアに指定してください。

nodeSelector の値は、クラスター全体のデフォルトでも制御でき、値を上書きできます。ビルド設定で **nodeSelector** のキー/値ペアが定義されておらず、 **nodeSelector:{}** が明示的に空になるように定義されていない場合にのみ、デフォルト値が適用されます。値を上書きすると、キーごとにビルド設定の値が置き換えられます。



注記

指定の **NodeSelector** がこれらのラベルが指定されているノードに一致しない場合には、ビルドは **Pending** の状態が無限に続きます。

手順

- 以下のように、**BuildConfig** の **nodeSelector** フィールドにラベルを割り当て、特定の一度で実行されるビルドを割り当てます。

```
apiVersion: "v1"
kind: "BuildConfig"
metadata:
  name: "sample-build"
spec:
  nodeSelector: (1)
    key1: value1
    key2: value2
```

1 このビルド設定に関連するビルドは、**key1=value2** と **key2=value2** ラベルが指定されたノードでのみ実行されます。

9.4. チェーンビルド

コンパイル言語 (Go、C、C++、Java など) の場合には、アプリケーションイメージにコンパイルに必要な依存関係を追加すると、イメージのサイズが増加したり、悪用される可能性のある脆弱性が発生したりする可能性があります。

これらの問題を回避するには、2 つのビルドをチェーンでつなげることができます。1 つ目のビルドでコンパイルしたアーティファクトを作成し、2 つ目のビルドで、アーティファクトを実行する別のイメージにそのアーティファクトを配置します。

以下の例では、Source-to-Image (S2I) ビルドが docker ビルドに組み合わされ、別のランタイムイメージに配置されるアーティファクトがコンパイルされます。

注記

この例では、S2I ビルドと docker ビルドをチェーンでつないでいますが、1 つ目のビルドは、必要なアーティファクトを含むイメージを生成するストラテジーを使用し、2 つ目のビルドは、イメージからの入力コンテンツを使用できるストラテジーを使用できます。

最初のビルドは、アプリケーションソースを取得して、**WAR** ファイルを含むイメージを作成します。このイメージは、**artifact-image** イメージストリームにプッシュされます。アウトプットアーティファクトのパスは、使用する S2I ビルダーの **assemble** スクリプトにより異なります。この場合、**/wildfly/standalone/deployments/ROOT.war** に出力されます。

```
apiVersion: build.openshift.io/v1
kind: BuildConfig
metadata:
  name: artifact-build
spec:
  output:
    to:
      kind: ImageStreamTag
      name: artifact-image:latest
  source:
    git:
      uri: https://github.com/openshift/openshift-jee-sample.git
      ref: "master"
  strategy:
    sourceStrategy:
      from:
        kind: ImageStreamTag
        name: wildfly:10.1
        namespace: openshift
```

2 つ目のビルドは、1 つ目のビルドからのアウトプットイメージ内にある WAR ファイルへのパスが指定されているイメージソースを使用します。インライン **dockerfile** は、 **WAR** ファイルをランタイムイメージにコピーします。

```
apiVersion: build.openshift.io/v1
```

```
kind: BuildConfig
metadata:
  name: image-build
spec:
  output:
    to:
      kind: ImageStreamTag
      name: image-build:latest
  source:
    dockerfile: |-
      FROM jee-runtime:latest
      COPY ROOT.war /deployments/ROOT.war
    images:
    - from: 1
        kind: ImageStreamTag
        name: artifact-image:latest
      paths: 2
      - sourcePath: /wildfly/standalone/deployments/ROOT.war
        destinationDir: "."
  strategy:
    dockerStrategy:
      from: 3
        kind: ImageStreamTag
        name: jee-runtime:latest
  triggers:
  - imageChange: {}
    type: ImageChange
```

1. **from** は、docker ビルドに、以前のビルドのターゲットであった **artifact-image** イメージストリームからのイメージの出力を追加する必要があることを指定します。
2. **paths** は、現在の docker ビルドに追加するターゲットイメージからのパスを指定します。
3. ランタイムのイメージは、docker ビルドのソースイメージとして使用します。

この設定の結果、2 番目のビルドのアウトプットイメージに、**WAR** ファイルの作成に必要なビルドツールを含める必要がなくなります。また、この 2 番目のビルドにはイメージ変更のトリガーが含まれているので、1 番目のビルドがバイナリーアーティファクトで新規イメージを実行して作成するたびに、2 番目のビルドが自動的に、そのアーティファクトを含むランタイムイメージを生成するためにトリガーされます。そのため、どちらのビルドも、ステージが 2 つある単一ビルドのように振る舞います。

9.5. ビルドのプルーニング

デフォルトで、ライフサイクルを完了したビルドは無制限に保持されます。保持される以前のビルドの数を制限することができます。

手順

1. **successfulBuildsHistoryLimit** または **failedBuildsHistoryLimit** の正の値を **BuildConfig** に指定して、保持される以前のビルドの数を制限します。 以下は例になります。

```
apiVersion: "v1"
kind: "BuildConfig"
```

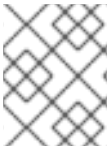
```
metadata:
  name: "sample-build"
spec:
  successfulBuildsHistoryLimit: 2 (1)
  failedBuildsHistoryLimit: 2 (2)
```

(1) **successfulBuildsHistoryLimit** は、**completed** のステータスのビルドを最大 2 つまで保持します。

(2) **failedBuildsHistoryLimit** はステータスが **failed**、**cancelled** または **error** のビルドを最大 2 つまで保持します。

2. 以下の動作のいずれかを実行して、ビルドのプルーニングをトリガーします。
   - ビルド設定が更新された場合
   - ビルドがそのライフサイクルを完了するのを待機します。

ビルドは、作成時のタイムスタンプで分類され、一番古いビルドが先にプルーニングされます。



注記

管理者は、'oc adm' オブジェクトプルーニングコマンドを使用して、ビルドを手動でプルーニングできます。

9.6. ビルド実行ポリシー

ビルド実行ポリシーでは、ビルド設定から作成されるビルドを実行する順番を記述します。これには、**Build** の **spec** セクションにある **runPolicy** フィールドの値を変更してください。

既存のビルド設定の **runPolicy** 値を変更することも可能です。以下を実行します。

- **Parallel** から **Serial** や **SerialLatestOnly** に変更して、この設定から新規ビルドをトリガーすると、新しいビルドは並列ビルドすべてが完了するまで待機します。 これは、順次ビルドは、一度に 1 つしか実行できないためです。
- **Serial** を **SerialLatestOnly** に変更して、新規ビルドをトリガーすると、現在実行中のビルドと直近で作成されたビルド以外には、キューにある既存のビルドがすべてキャンセルされます。最新のビルドが次に実行されます。

第10章 ビルドでの RED HAT サブスクリプションの使用

以下のセクションを使用して、OpenShift Container Platform ビルド内に Red Hat サブスクリプションコンテンツをインストールします。

10.1. RED HAT UNIVERSAL BASE IMAGE へのイメージストリームタグの作成

ビルド内に Red Hat Enterprise Linux (RHEL) パッケージをインストールするには、Red Hat Universal Base Image (UBI) を参照するイメージストリームタグを作成します。

クラスター内の **すべてのプロジェクトで** UBI を利用可能にするには、イメージストリームタグを **openshift** namespace に追加します。または、**特定のプロジェクトで** UBI を利用可能にするには、イメージストリームタグをそのプロジェクトに追加します。

イメージストリームタグは、他のユーザーにプルシークレットを公開せずに、インストールプルシークレットにある **registry.redhat.io** の認証情報を使用して UBI へのアクセスを許可します。この方法は、各プロジェクトで **registry.redhat.io** の認証情報を使用してプルシークレットをインストールするよう各開発者に求める方法よりも便利です。

手順

- **openshift** namespace で **ImageStreamTag** を作成し、これを開発者に対してすべてのプロジェクトで利用可能にするには、以下を実行します。

```
$ oc tag --source=docker registry.redhat.io/ubi9/ubi:latest ubi9:latest -n openshift
```

ヒント

または、以下の YAML を適用して **openshift** namespace に **ImageStreamTag** を作成できます。

```
apiVersion: image.openshift.io/v1
kind: ImageStream
metadata:
  name: ubi9
  namespace: openshift
spec:
  tags:
  - from:
      kind: DockerImage
      name: registry.redhat.io/ubi9/ubi:latest
    name: latest
    referencePolicy:
      type: Source
```

- 単一プロジェクトで **ImageStreamTag** を作成するには、以下を実行します。

```
$ oc tag --source=docker registry.redhat.io/ubi9/ubi:latest ubi:latest
```

ヒント

または、以下の YAML を適用して単一のプロジェクトに **ImageStreamTag** を作成できます。

```
apiVersion: image.openshift.io/v1
kind: ImageStream
metadata:
  name: ubi9
spec:
  tags:
  - from:
      kind: DockerImage
      name: registry.redhat.io/ubi9/ubi:latest
    name: latest
    referencePolicy:
      type: Source
```

10.2. ビルドシークレットとしてのサブスクリプションエンタイトルメントの追加

Red Hat サブスクリプションを使用してコンテンツをインストールするビルドには、ビルドシークレットとしてエンタイトルメントキーを含める必要があります。

前提条件

サブスクリプションを通じて Red Hat Enterprise Linux (RHEL) パッケージリポジトリーにアクセスできる必要があります。

これらのリポジトリーにアクセスするためのエンタイトルメントシークレットは、クラスターがサブスクライブされるときに、Insights Operator によって自動的に作成されます。

重要

クラスター管理者であるか、**openshift-config-managed** プロジェクトのシークレットにアクセスする権限を持っている必要があります。

手順

1. エンタイトルメントシークレットを **openshift-config-managed** namespace からビルドの namespace にコピーします。

```
$ cat << EOF > secret-template.txt
kind: Secret
apiVersion: v1
metadata:
  name: etc-pki-entitlement
type: Opaque
data: {{ range \$key, \$value := .data }}
  {{ \$key }}: {{ \$value }} {{ end }}
EOF
$ oc get secret etc-pki-entitlement -n openshift-config-managed -o=go-template-file --
template=secret-template.txt | oc apply -f -
```

2. etc-pki-entitlement シークレットをビルド設定の Docker ストラテジーでビルドボリュームとして追加します。

```
strategy:
  dockerStrategy:
    from:
      kind: ImageStreamTag
      name: ubi9:latest
    volumes:
    - name: etc-pki-entitlement
      mounts:
      - destinationPath: /etc/pki/entitlement
      source:
        type: Secret
        secret:
          secretName: etc-pki-entitlement
```

10.3. SUBSCRIPTION MANAGER を使用したビルドの実行

10.3.1. Subscription Manager を使用した Docker ビルド

Docker ストラテジービルドでは、**yum** または **dnf** を使用して追加の Red Hat Enterprise Linux (RHEL) パッケージをインストールできます。

前提条件

エンタイトルメントキーは、ビルドストラテジーのボリュームとして追加する必要があります。

手順

以下を Dockerfile の例として使用し、Subscription Manager でコンテンツをインストールします。

```
FROM registry.redhat.io/ubi9/ubi:latest
RUN rm -rf /etc/rhsm-host (1)
RUN yum --enablerepo=codeready-builder-for-rhel-9-x86_64-rpms install \ (2)
    nss_wrapper \
    uid_wrapper -y && \
    yum clean all -y
RUN ln -s /run/secrets/rhsm /etc/rhsm-host (3)
```

(1) **yum** または **dnf** コマンドを実行する前に、**/etc/rhsm-host** ディレクトリーとそのすべての内容を削除するコマンドを Dockerfile に含める必要があります。

(2) Red Hat Package Browser を使用して、インストールされているパッケージの正しいリポジトリーを見つけます。

(3) イメージと他の Red Hat コンテナーイメージとの互換性を維持するために、**/etc/rhsm-host** のシンボリックリンクを復元する必要があります。

10.4. RED HAT SATELLITE サブスクリプションを使用したビルドの実行

10.4.1. Red Hat Satellite 設定のビルドへの追加

Red Hat Satellite を使用してコンテンツをインストールするビルドは、Satellite リポジトリーからコンテンツを取得するための適切な設定を提供する必要があります。

前提条件

- Satellite インスタンスからコンテンツをダウンロードするために、**yum** 互換リポジトリー設定ファイルを提供するか、これを作成する必要があります。

  サンプルリポジトリーの設定

```
[test-<name>]
name=test-<number>
baseurl = https://satellite.../content/dist/rhel/server/7/7Server/x86_64/os
enabled=1
gpgcheck=0
sslverify=0
sslclientkey = /etc/pki/entitlement/...-key.pem
sslclientcert = /etc/pki/entitlement/....pem
```

手順

1. Satellite リポジトリーの設定ファイルを含む **ConfigMap** を作成します。

```
$ oc create configmap yum-repos-d --from-file /path/to/satellite.repo
```

2. Satellite リポジトリー設定およびエンタイトルメントキーをビルドボリュームとして追加します。

```
strategy:
  dockerStrategy:
    from:
      kind: ImageStreamTag
      name: ubi9:latest
    volumes:
    - name: yum-repos-d
      mounts:
      - destinationPath: /etc/yum.repos.d
      source:
        type: ConfigMap
        configMap:
          name: yum-repos-d
    - name: etc-pki-entitlement
      mounts:
      - destinationPath: /etc/pki/entitlement
      source:
        type: Secret
        secret:
          secretName: etc-pki-entitlement
```

10.4.2. Red Hat Satellite サブスクリプションを使用した Docker ビルド

Docker ストラテジービルドは、Red Hat Satellite リポジトリーを使用してサブスクリプションコンテンツをインストールできます。

前提条件

- エンタイトルメントキーと Satellite リポジトリー設定がビルドボリュームとして追加しておく。

手順

以下のサンプル Dockerfile を使用して、Satellite を使用してコンテンツをインストールします。

```
FROM registry.redhat.io/ubi9/ubi:latest
RUN rm -rf /etc/rhsm-host (1)
RUN yum --enablerepo=codeready-builder-for-rhel-9-x86_64-rpms install \ (2)
    nss_wrapper \
    uid_wrapper -y && \
    yum clean all -y
RUN ln -s /run/secrets/rhsm /etc/rhsm-host (3)
```

1. **yum** または **dnf** コマンドを実行する前に、**/etc/rhsm-host** ディレクトリーとそのすべての内容を削除するコマンドを Dockerfile に含める必要があります。
2. ビルドのインストール済みパッケージの正しいリポジトリーを見つけるには、Satellite システム管理者に問い合わせてください。
3. イメージと他の Red Hat コンテナーイメージとの互換性を維持するために、**/etc/rhsm-host** のシンボリックリンクを復元する必要があります。

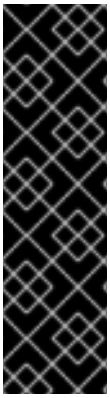
関連情報

- Red Hat Satellite サブスクリプションと使用する証明書でビルドを使用する方法

10.5. SHAREDSECRET オブジェクトを使用したビルドの実行

SharedSecret オブジェクトを使用すると、ビルドでクラスターのエンタイトルメントキーにセキュアにアクセスできます。

SharedSecret オブジェクトを使用すると、namespace 間でシークレットを共有および同期できます。



重要

Shared Resource CSI Driver は、テクノロジープレビュー機能としてのみ提供されます。テクノロジープレビュー機能は、Red Hat 製品サポートのサービスレベルアグリーメント (SLA) の対象外であり、機能的に完全ではない場合があります。Red Hat は、実稼働環境でこれらを使用することを推奨していません。テクノロジープレビュー機能は、最新の製品機能をいち早く提供して、開発段階で機能のテストを行いフィードバックを提供していただくことを目的としています。

Red Hat のテクノロジープレビュー機能のサポート範囲に関する詳細は、テクノロジープレビュー機能のサポート範囲 を参照してください。

前提条件

- フィーチャーゲートを使用して **TechPreviewNoUpgrade** 機能セットを有効にしている。詳細は、フィーチャーゲートを使用した機能の有効化 を参照してください。

- 次のアクションを実行するためのパーミッションがある。
  - ビルド設定を作成し、ビルドを開始します。
  - **oc get sharedsecrets**コマンドを入力し、空でないリストを取得して、使用可能な**SharedSecret** CR インスタンスを見つけます。
  - namespace で使用可能な **builder** サービスアカウントが、指定された **SharedSecret** CR インスタンスの使用を許可されているかどうかを確認します。つまり、**oc adm policy who-can use <identifier of specific SharedSecret>** を使用して、namespace の**builder**サービスアカウントが一覧表示されているかどうかを確認できます。

注記

このリストの最後の 2 つの前提条件のいずれも満たされない場合は、必要なロールベースアクセス制御 (RBAC) を自身で確立するか、誰かに依頼して確立します。これにより、**SharedSecret** CR インスタンスを検出し、サービスアカウントを有効にして **SharedSecret** CR インスタンスを使用できるようになります。

手順

1. **oc apply** を使用して、クラスターのエンタイトルメントシークレットで **SharedSecret** オブジェクトインスタンスを作成します。

重要

SharedSecret オブジェクトを作成するには、クラスター管理者の権限が必要です。

YAML Role オブジェクト定義を使用した oc apply -f コマンドの例

```
$ oc apply -f - <<EOF
kind: SharedSecret
apiVersion: sharedresource.openshift.io/v1alpha1
metadata:
  name: etc-pki-entitlement
spec:
  secretRef:
    name: etc-pki-entitlement
    namespace: openshift-config-managed
EOF
```

2. **SharedSecret** オブジェクトにアクセスするための権限を **builder** サービスアカウントに付与するロールを作成します。

oc apply -f コマンドの例

```
$ oc apply -f - <<EOF
apiVersion: rbac.authorization.k8s.io/v1
kind: Role
metadata:
  name: builder-etc-pki-entitlement
  namespace: build-namespace
rules:
```

```
  - apiGroups:
      - sharedresource.openshift.io
    resources:
      - sharedsecrets
    resourceNames:
      - etc-pki-entitlement
    verbs:
      - use
EOF
```

3. 次のコマンドを実行して、**builder** サービスアカウントに **SharedSecret** オブジェクトにアクセスするための権限を付与する **RoleBinding** オブジェクトを作成します。

oc create rolebinding コマンドの例

```
$ oc create rolebinding builder-etc-pki-entitlement --role=builder-etc-pki-entitlement --
serviceaccount=build-namespace:builder
```

4. CSI ボリュームマウントを使用して、エンタイトルメントシークレットを **BuildConfig** オブジェクトに追加します。

YAML BuildConfig オブジェクト定義の例

```
apiVersion: build.openshift.io/v1
kind: BuildConfig
metadata:
  name: uid-wrapper-rhel9
  namespace: build-namespace
spec:
  runPolicy: Serial
  source:
    dockerfile: |
      FROM registry.redhat.io/ubi9/ubi:latest
      RUN rm -rf /etc/rhsm-host (1)
      RUN yum --enablerepo=codeready-builder-for-rhel-9-x86_64-rpms install \ (2)
          nss_wrapper \
          uid_wrapper -y && \
          yum clean all -y
      RUN ln -s /run/secrets/rhsm /etc/rhsm-host (3)
  strategy:
    type: Docker
    dockerStrategy:
      volumes:
        - mounts:
            - destinationPath: "/etc/pki/entitlement"
          name: etc-pki-entitlement
          source:
            csi:
              driver: csi.sharedresource.openshift.io
              readOnly: true (4)
              volumeAttributes:
                sharedSecret: etc-pki-entitlement (5)
            type: CSI
```

1. **yum** または **dnf** コマンドを実行する前に、**/etc/rhsm-host** ディレクトリーとそのすべての内容を削除するコマンドを Dockerfile に含める必要があります。
2. Red Hat Package Browser を使用して、インストールされているパッケージの正しいリポジトリーを見つけます。
3. イメージと他の Red Hat コンテナーイメージとの互換性を維持するために、**/etc/rhsm-host** のシンボリックリンクを復元する必要があります。
4. ビルドで共有リソースをマウントするには、**readOnly** を **true** に設定する必要があります。
5. ビルドに含める **SharedSecret** オブジェクトの名前を参照します。

5. **BuildConfig** オブジェクトからビルドを開始し、**oc** コマンドを使用してログを追跡します。

```
$ oc start-build uid-wrapper-rhel9 -n build-namespace -F
```

10.6. 関連情報

- Insights Operator を使用した単純なコンテンツアクセス証明書のインポート
- フィーチャーゲートを使用した機能の有効化
- イメージストリームの管理
- ビルドストラテジー

第11章 ストラテジーによるビルドのセキュリティー保護

OpenShift Container Platform のビルドは特権付きコンテナーで実行されます。使用されるビルドストラテジーに応じて、権限がある場合は、ビルドを実行してクラスターおよびホストノードでの自らのパーミッションをエスカレートすることができます。セキュリティー対策として、ビルドを実行できるユーザーおよびそれらのビルドに使用されるストラテジーを制限します。カスタムビルドは特権付きコンテナー内で任意のコードを実行できるためにソースビルドより安全性が低くなります。そのためデフォルトで無効にされます。Dockerfile 処理ロジックにある脆弱性により、権限がホストノードで付与される可能性があるため、docker ビルドパーミッションを付与する際には注意してください。

デフォルトで、ビルドを作成できるすべてのユーザーには docker および Source-to-Image (S2I) ビルドストラテジーを使用するためにパーミッションが付与されます。クラスター管理者権限を持つユーザーは、ビルドストラテジーをユーザーにぐローバルに制限する方法についてのセクションで言及されているようにカスタムビルドストラテジーを有効にできます。

許可ポリシーを使用して、どのユーザーがどのビルドストラテジーを使用してビルドできるかについて制限することができます。各ビルドストラテジーには、対応するビルドサブリソースがあります。ストラテジーを使用してビルド作成するには、ユーザーにビルドを作成するパーミッションおよびビルドストラテジーのサブリソースで作成するパーミッションがなければなりません。ビルドストラテジーのサブリソースでの create パーミッションを付与するデフォルトロールが提供されます。

表11.1 ビルドストラテジーのサブリソースおよびロール

ストラテジー	サブリソース	ロール
Docker	ビルド/docker	system:build-strategy-docker
Source-to-Image (S2I)	ビルド/ソース	system:build-strategy-source
カスタム	ビルド/カスタム	system:build-strategy-custom
JenkinsPipeline	ビルド/jenkinspipeline	system:build-strategy-jenkinspipeline

11.1. ビルドストラテジーへのアクセスのグローバルな無効化

特定のビルドストラテジーへのアクセスをグローバルに禁止するには、クラスター管理者の権限を持つユーザーとしてログインし、**system:authenticated** グループから対応するロールを削除し、アノテーション **rbac.authorization.kubernetes.io/autoupdate: "false"** を適用してそれらを API の再起動間での変更から保護します。以下の例では、docker ビルドストラテジーを無効にする方法を示します。

手順

1. **rbac.authorization.kubernetes.io/autoupdate** アノテーションを適用します。

```
$ oc edit clusterrolebinding system:build-strategy-docker-binding
```

出力例

```
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRoleBinding
metadata:
```

```
  annotations:
    rbac.authorization.kubernetes.io/autoupdate: "false" (1)
  creationTimestamp: 2018-08-10T01:24:14Z
  name: system:build-strategy-docker-binding
  resourceVersion: "225"
  selfLink: /apis/rbac.authorization.k8s.io/v1/clusterrolebindings/system%3Abuild-strategy-
docker-binding
  uid: 17b1f3d4-9c3c-11e8-be62-0800277d20bf
roleRef:
  apiGroup: rbac.authorization.k8s.io
  kind: ClusterRole
  name: system:build-strategy-docker
subjects:
- apiGroup: rbac.authorization.k8s.io
  kind: Group
  name: system:authenticated
```

(1) **rbac.authorization.kubernetes.io/autoupdate** アノテーションの値を **"false"** に変更します。

2. ロールを削除します。

```
$ oc adm policy remove-cluster-role-from-group system:build-strategy-docker
system:authenticated
```

3. ビルドストラテジーのサブリソースもこれらのロールから削除されることを確認します。

```
$ oc edit clusterrole admin
```

```
$ oc edit clusterrole edit
```

4. ロールごとに、無効にするストラテジーのリソースに対応するサブリソースを指定します。

   a. **admin** の docker ビルドストラテジーの無効化

```
kind: ClusterRole
metadata:
  name: admin
...
- apiGroups:
  - ""
  - build.openshift.io
  resources:
  - buildconfigs
  - buildconfigs/webhooks
  - builds/custom (1)
  - builds/source
  verbs:
  - create
  - delete
  - deletecollection
  - get
  - list
```

```
  - patch
  - update
  - watch
...
```

1 **builds/custom** と **builds/source** を追加して、**admin** ロールが割り当てられたユーザーに対して docker ビルドをグローバルに無効にします。

11.2. ユーザーへのビルドストラテジーのグルーバルな制限

一連の特定ユーザーのみが特定のストラテジーでビルドを作成できます。

前提条件

- ビルドストラテジーへのグローバルアクセスを無効にします。

手順

- ビルドストラテジーに対応するロールを特定ユーザーに割り当てます。たとえば、**system:build-strategy-docker** クラスターロールをユーザー **devuser** に追加するには、以下を実行します。

```
$ oc adm policy add-cluster-role-to-user system:build-strategy-docker devuser
```

警告

ユーザーに対して **builds/docker** サブリソースへのクラスターレベルでのアクセスを付与することは、そのユーザーがビルドを作成できるすべてのプロジェクトにおいて、docker ストラテジーを使用してビルドを作成できることを意味します。

11.3. プロジェクト内でのユーザーへのビルドストラテジーの制限

ユーザーにビルドストラテジーをグローバルに付与するのと同様に、プロジェクト内の特定ユーザーのセットのみが特定ストラテジーでビルドを作成することを許可できます。

前提条件

- ビルドストラテジーへのグローバルアクセスを無効にします。

手順

- ビルドストラテジーに対応するロールをプロジェクト内の特定ユーザーに付与します。たとえば、プロジェクト **devproject** 内の **system:build-strategy-docker** ロールをユーザー **devuser** に追加するには、以下を実行します。

```
$ oc adm policy add-role-to-user system:build-strategy-docker devuser -n devproject
```

第12章 ビルド設定リソース

以下の手順でビルドを設定します。

12.1. ビルドコントローラー設定パラメーター

build.config.openshift.io/cluster リソースは以下の設定パラメーターを提供します。

パラメーター	説明
Build	ビルドの処理方法についてのクラスター全体の情報を保持します。正規名および唯一の有効な名前となるのは **cluster** です。 **spec**: ビルドコントローラー設定のユーザーが設定できる値を保持します。
buildDefaults	ビルドのデフォルト情報を制御します。 **defaultProxy**: イメージのプルまたはプッシュ、およびソースのダウンロードを含む、ビルド操作のデフォルトのプロキシー設定が含まれます。 **BuildConfig** ストラテジーに **HTTP_PROXY**、**HTTPS_PROXY**、および **NO_PROXY** 環境変数を設定することで、値を上書きできます。 **gitProxy**: Git 操作のプロキシー設定のみが含まれます。設定されている場合、これは **git clone** などの Git コマンドのプロキシー設定を上書きします。 ここで設定されていない値は DefaultProxy から継承されます。 **env**: 指定される変数がビルドに存在しない場合にビルドに適用される一連のデフォルト環境変数。 **imageLabels**: 結果として生成されるイメージに適用されるラベルのリスト。**BuildConfig** に同じ名前のラベルを指定することでデフォルトのラベルを上書きできます。 **resources**: ビルドを実行するためのリソース要件を定義します。
ImageLabel	**name**: ラベルの名前を定義します。ゼロ以外の長さを持つ必要があります。
buildOverrides	ビルドの上書き設定を制御します。 **imageLabels**: 結果として生成されるイメージに適用されるラベルのリスト。表にあるものと同じ名前のラベルを **BuildConfig** に指定する場合、ラベルは上書きされます。 **nodeSelector**: セレクター。 ビルド Pod がノードに適合させるには True である必要があります。 **tolerations**: ビルド Pod に設定された既存の容認を上書きする容認のリスト。
BuildList	**items**: 標準オブジェクトのメタデータ。

12.2. ビルド設定の設定

build.config.openshift.io/cluster リソースを編集してビルドの設定を行うことができます。

手順

- **build.config.openshift.io/cluster** リソースを編集します。

```
$ oc edit build.config.openshift.io/cluster
```

以下は、**build.config.openshift.io/cluster** リソースの例になります。

```
apiVersion: config.openshift.io/v1
kind: Build(1)
metadata:
  annotations:
    release.openshift.io/create-only: "true"
  creationTimestamp: "2019-05-17T13:44:26Z"
  generation: 2
  name: cluster
  resourceVersion: "107233"
  selfLink: /apis/config.openshift.io/v1/builds/cluster
  uid: e2e9cc14-78a9-11e9-b92b-06d6c7da38dc
spec:
  buildDefaults:(2)
    defaultProxy:(3)
      httpProxy: http://proxy.com
      httpsProxy: https://proxy.com
      noProxy: internal.com
    env:(4)
    - name: envkey
      value: envvalue
    gitProxy:(5)
      httpProxy: http://gitproxy.com
      httpsProxy: https://gitproxy.com
      noProxy: internalgit.com
    imageLabels:(6)
    - name: labelkey
      value: labelvalue
    resources:(7)
      limits:
        cpu: 100m
        memory: 50Mi
      requests:
        cpu: 10m
        memory: 10Mi
  buildOverrides:(8)
    imageLabels:(9)
    - name: labelkey
      value: labelvalue
    nodeSelector:(10)
      selectorkey: selectorvalue
    tolerations:(11)
    - effect: NoSchedule
      key: node-role.kubernetes.io/builds
operator: Exists
```

```
-
```

1. **Build**: ビルドの処理方法についてのクラスター全体の情報を保持します。正規名および唯一の有効な名前となるのは **cluster** です。
2. **buildDefaults**: ビルドのデフォルト情報を制御します。
3. **defaultProxy**: イメージのプルまたはプッシュ、およびソースのダウンロードを含む、ビルド操作のデフォルトのプロキシー設定が含まれます。
4. **env**: 指定される変数がビルドに存在しない場合にビルドに適用される一連のデフォルト環境変数。
5. **gitProxy**: Git 操作のプロキシー設定のみが含まれます。設定されている場合、これは **git clone** などの Git コマンドのプロキシー設定を上書きします。
6. **imageLabels**: 結果として生成されるイメージに適用されるラベルのリスト。**BuildConfig** に同じ名前のラベルを指定することでデフォルトのラベルを上書きできます。
7. **resources**: ビルドを実行するためのリソース要件を定義します。
8. **buildOverrides**: ビルドの上書き設定を制御します。
9. **imageLabels**: 結果として生成されるイメージに適用されるラベルのリスト。表にあるものと同じ名前のラベルを **BuildConfig** に指定する場合、ラベルは上書きされます。
10. **nodeSelector**: セレクター。 ビルド Pod がノードに適合させるには True である必要があります。
11. **tolerations**: ビルド Pod に設定された既存の容認を上書きする容認のリスト。

第13章 ビルドのトラブルシューティング

ビルドの問題をトラブルシューティングするために、以下を使用します。

13.1. リソースへのアクセスのための拒否の解決

リソースへのアクセス要求が拒否される場合:

問題

ビルドが以下のエラーで失敗します。

```
requested access to the resource is denied
```

解決策

プロジェクトに設定されているイメージのクォータのいずれかの上限を超えています。現在のクォータを確認して、適用されている制限数と、使用中のストレージを確認してください。

```
$ oc describe quota
```

13.2. サービス証明書の生成に失敗

リソースへのアクセス要求が拒否される場合:

問題

サービス証明書の生成は以下を出して失敗します (サービスの **service.beta.openshift.io/serving-cert-generation-error** アノテーションには以下が含まれます)。

出力例

```
secret/ssl-key references serviceUID 62ad25ca-d703-11e6-9d6f-0e9c0057b608, which does not match 77b6dd80-d716-11e6-9d6f-0e9c0057b60
```

解決策

証明書を生成したサービスがすでに存在しないか、サービスに異なる **serviceUID** があります。古いシークレットを削除し、サービスのアノテーション (**service.beta.openshift.io/serving-cert-generation-error** および **service.beta.openshift.io/serving-cert-generation-error-num**) をクリアして証明書の再生成を強制的に実行する必要があります。

```
$ oc delete secret <secret_name>
```

```
$ oc annotate service <service_name> service.beta.openshift.io/serving-cert-generation-error-
```

```
$ oc annotate service <service_name> service.beta.openshift.io/serving-cert-generation-error-num-
```

注記

アノテーションを削除するコマンドでは、削除するアノテーション名の後に **-** を付けます。

第14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、クラスター管理者で **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。

14.1. 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

- 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 ファイルで、**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
```

14.2. 関連情報

- **ConfigMap** の作成
- シークレットおよび **ConfigMap**
- カスタム PKI の設定