



OpenShift Dedicated 4

BuildConfig を使用した Builds

OpenShift Dedicated のビルドに関する情報

OpenShift Dedicated 4 BuildConfig を使用した Builds

OpenShift Dedicated のビルドに関する情報

Legal Notice

Copyright © 2025 Red Hat, Inc.

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

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

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

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

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

Linux[®] is the registered trademark of Linus Torvalds in the United States and other countries.

Java[®] is a registered trademark of Oracle and/or its affiliates.

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

MySQL[®] is a registered trademark of MySQL AB in the United States, the European Union and other countries.

Node.js[®] is an official trademark of Joyent. Red Hat is not formally related to or endorsed by the official Joyent Node.js open source or commercial project.

The OpenStack[®] Word Mark and OpenStack logo are either registered trademarks/service marks or trademarks/service marks of the OpenStack Foundation, in the United States and other countries and are used with the OpenStack Foundation's permission. We are not affiliated with, endorsed or sponsored by the OpenStack Foundation, or the OpenStack community.

All other trademarks are the property of their respective owners.

Abstract

OpenShift Dedicated クラスターのビルド。

Table of Contents

第1章 イメージ BUILDS について	4
1.1. BUILDS	4
第2章 ビルド設定について	6
2.1. BUILDCONFIG	6
第3章 ビルド入力の作成	8
3.1. ビルド入力	8
3.2. DOCKERFILE ソース	9
3.3. イメージソース	9
3.4. GIT ソース	11
3.5. バイナリー (ローカル) ソース	20
3.6. 入力シークレットおよび CONFIG MAP	22
3.7. 外部アーティファクト	30
3.8. プライベートレジストリーでの DOCKER 認証情報の使用	30
3.9. ビルド環境	33
3.10. サービス提供証明書のシークレット	34
3.11. シークレットの制限	35
第4章 ビルド出力の管理	36
4.1. ビルド出力	36
4.2. アウトプットイメージの環境変数	36
4.3. アウトプットイメージのラベル	37
第5章 ビルドストラテジーの使用	38
5.1. DOCKER ビルド	38
5.2. SOURCE-TO-IMAGE ビルド	41
5.3. PIPELINE ビルド	47
5.4. WEB コンソールを使用したシークレットの追加	56
5.5. プルおよびプッシュの有効化	56
第6章 基本的なビルドの実行および設定	58
6.1. ビルドの開始	58
6.2. ビルドの中止	59
6.3. BUILDCONFIG の編集	60
6.4. BUILDCONFIG の削除	61
6.5. ビルドの詳細表示	62
6.6. ビルドログへのアクセス	62
第7章 BUILDS のトリガーおよび変更	64
7.1. ビルドトリガー	64
7.2. ビルドフック	76
第8章 高度なビルドの実行	79
8.1. ビルドリソースの設定	79
8.2. 最長期間の設定	80
8.3. 特定のノードへのビルドの割り当て	80
8.4. チェーンビルド	81
8.5. ビルドのプルーニング	81
8.6. ビルド実行ポリシー	81
第9章 ビルドでの RED HAT サブスクリプションの使用	83
9.1. RED HAT UNIVERSAL BASE IMAGE へのイメージストリームタグの作成	83

9.2. 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	83
9.3. SUBSCRIPTION MANAGER を使用した BUILDS の実行	84
9.4. RED HAT SATELLITE サブスクリプションを使用した BUILDS の実行	85
9.5. 関連情報	87
第10章 BUILDS のトラブルシューティング	88
10.1. リソースへのアクセスのための拒否の解決	88
10.2. サービス証明書の生成に失敗	88

第1章 イメージ BUILDS について

1.1. BUILDS

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

OpenShift Dedicated は、ビルドイメージからコンテナを作成し、それらをコンテナ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関する要件、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。Builds 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。

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 builds 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 builds 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 Dedicated は Buildah を使用して Dockerfile からコンテナイメージを構築します。Dockerfile を使用したコンテナイメージのビルドの詳細は、[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](#) を参照してください。

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イメージに注入し、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 builds 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. Pipeline ビルド



重要

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

OpenShift Dedicated 上の Jenkins イメージは完全にサポートされており、ユーザーは Jenkins ユーザードキュメントに従ってジョブで **jenkinsfile** を定義するか、ソースコントロール管理システムに保存する必要があります。

開発者は、Pipeline 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。このビルドは他のビルドタイプの場合と同様に OpenShift Dedicated での起動、モニタリング、管理が可能です。

Pipeline ワークフローは、ビルド設定に直接組み込むか、Git リポジトリ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関する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 Dedicated を使用したアプリケーションの作成方法の選択に応じて Web コンソールまたは CLI のいずれを使用している場合でも、**BuildConfig** は通常自動的に作成され、いつでも編集できます。**BuildConfig** を設定する部分や利用可能なオプションを理解しておく、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 builds 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 builds 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** コンテナイメージを使用します。
- 6 コンテナ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、builds 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を説明します。

3.1. ビルド入力

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

- インラインの Dockerfile 定義
- 既存イメージから抽出したコンテンツ
- Git リポジトリ
- 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** 内には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、各入力タイプに関する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 ❷
  sourcePath: /usr/lib/somefile.jar
contextDir: "app/dir" ❸
dockerfile: "FROM centos:7\nRUN yum install -y httpd" ❹

```

- ❶ 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
- ❷ `myinputimage` の `/usr/lib/somefile.jar` は、`<workingdir>/app/dir/injected/dir` に保存されます。
- ❸ ビルドの作業ディレクトリーは `<original_workingdir>/app/dir` になります。
- ❹ 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せん。

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" ❶

```

- ❶ `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: ❶
  - from: ❷
    kind: ImageStreamTag
    name: myinputimage:latest
    namespace: mynamespace
  paths: ❸
  - destinationDir: injected/dir ❹
    sourcePath: /usr/lib/somefile.jar ❺
  - from:
    kind: ImageStreamTag
    name: myotherinputimage:latest
    namespace: myothernamespace
  pullSecret: mysecret ❻
  paths:
  - destinationDir: injected/dir
    sourcePath: /usr/lib/somefile.jar

```

- ❶ 1つ以上のインプットイメージおよびファイルの配列
- ❷ コピーされるファイルが含まれるイメージへの参照
- ❸ ソース/宛先パスの配列
- ❹ 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
- ❺ 参照イメージの中からコピーするファイルの場所
- ❻ 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、builds 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 builds をサポートしません。

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: ❶
    uri: "https://github.com/openshift/ruby-hello-world"
    ref: "master"
  contextDir: "app/dir" ❷
  dockerfile: "FROM openshift/ruby-22-centos7\nUSER example" ❸
```

- ❶ **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。
- ❷ **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オーバーライドできます。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) をオーバーライドできます。
- ❸ 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** 操作を使用してから **FETCH_HEAD** をチェックアウトします。

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

Pipeline ストラテジー builds 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などの通常アクセスできないリポジトリ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
- 信頼された認証局



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 Dedicated はソースクローンのシークレット参照を自動生成します。この動作により、追加の設定なしに、作成される builds が参照される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があります。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。**BuildConfig** がソースクローンシークレット参照なしで作成され、その Git ソース URI がシークレットアノテーションの URI パターンと一致する場合に、OpenShift Dedicated は **BuildConfig** にそのシークレットへの参照を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まないようにしてください。

たとえば、Git リポジトリの 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、OpenShift Dedicated は最も長い一致を持つ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. ソースクロンシークレットの手動による追加

ソースクロ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、builds 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。

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。

リポジトリのキーは通常 **\$HOME/.ssh/** ディレクトリにあり、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 Dedicated でビルドができなくなります。パスフレーズを求めるプロンプトが出されても、空白のままにします。

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

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

CA 証明書のシークレットを作成した場合に、OpenShift Dedicated はその証明書を使用して、Git clone 操作時に Git サーバーにアクセスします。提示される 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大幅に向上します。

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。以下のコマンドを実行します。

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

2. 次のコマンドを入力して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ることで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、次のコマンドを入力します。

```
$ 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たシークレットを作成するには、次のコマンドを入力します。

```
$ 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 証明書を使用して基本認証シークレットを作成するには、次のコマンドを入力します。

```
$ 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. Git 設定ファイルを使用した基本認証シークレットの作成

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

前提条件

- Basic 認証の認証情報
- **.gitconfig** ファイル

手順

- **.gitconfig** ファイルを使用して基本認証シークレットを作成するには、次のコマンドを入力します。

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

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

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

前提条件

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

手順

- **.gitconfig** ファイルと CA 証明書を使用して基本認証シークレットを作成するには、次のコマンドを入力します。

```
$ 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 builds に関する **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 builds 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 builds 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 builds を起動することはできません。

バイナリー builds 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内容を展開します。**--from-dir** を使用して、展開されるアーカイブに URL を指定することもできます。
- **--from-archive**: 指定したアーカイブはビルダーに送信され、ビルドコンテキストディレクトリーに展開されます。このオプションは **--from-dir** と同様に動作しますが、このオプションの引数がディレクトリーの場合には常にアーカイブがホストに最初に作成されます。

上記のそれぞれの例では、以下のようになります。

- **BuildConfig** に **Binary** のソースタイプが定義されている場合には、これは事実上無視され、クライアントが送信する内容に置き換えられます。
- **BuildConfig** に **Git** のソースタイプが定義されている場合には、**Binary** と **Git** は併用できないので、動的に無効にされます。この場合、ビルダーに渡されるバイナリーストリームのデータが優先されます。

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

oc new-build --binary=true を使用すると、バイナリー builds 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 builds に関して特別な意味を持ちません。

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

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

3.6. 入力シークレットおよび CONFIG MAP



重要

入力シークレットおよび config map のコンテンツがビルドの出力コンテナ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 config map 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 アプリケーションを説明していますが、`/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 Dedicated クライアント設定ファイル、**dockercfg** ファイル、プライベートソースリポジトリーの認証情報などの機密情報を保持するメカニズムを提供します。シークレットは機密内容を Pod から切り離します。シークレットはボリュームプラグインを使用してコンテナ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せん。
- 5 **stringData** マップのキーに関連付けられた値は単純なテキスト文字列で構成されます。

3.6.1.1. シークレットのプロパティ

キーのプロパティ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。 **kubectrl 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

- JSON または YAML ファイルからシークレットオブジェクトを作成するには、次のコマンドを入力します。

```
$ 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 ❶
data:
  username: <username>
  password: <password>

```

- ❶ 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 ❶
data:
  .dockerconfigjson:bm5ubm5ubm5ubm5ubm5ubm5ubmdnZ2dnZ2dnZ2dnZ2dnZ2cg
  YXV0aCBrZXlzCg== ❷

```

- ❶ 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。
- ❷ base64 でエンコードされた docker 設定 JSON ファイルの出力

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. 次のコマンドを入力して、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. 次のコマンドを入力してログを取得します。

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

3. 以下のコマンドを入力して 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シークレットのYAMLファイル

```

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 ファイル

```

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 ファイル

```

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

```

環境変数にシークレットデータを入力する BuildConfig オブジェクトの YAML ファイル

```

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 config map 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 config map 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 config map 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 config map を追加するには、以下を行います。

1. **ConfigMap** オブジェクトが存在しない場合は、次のコマンドを入力して作成します。

```

$ 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 config map 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 config map 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** オブジェクトが存在しない場合は、次のコマンドを入力して作成します。

```
$ 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. config map 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
```

4. シークレットおよび config map 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"
```

ビルド中に、ビルドプロセスは、**settings.xml** ファイルと **id_rsa** ファイルを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します。OpenShift Dedicated S2I ビルダージェネレータでは、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"
```

次のコマンドを入力して、新しい **BuildConfig** オブジェクトを作成するとき、宛先ディレクトリーを指定することもできます。

```
$ 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ジェネレータ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、ビルダージェネレータ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ます、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。

3.7. 外部アーティファクト

ソースリポジトリ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 (**Source** ビルドストラテジーのみ)
- **BuildConfig** オブジェクト内の変数の設定
- **oc start-build --env** コマンドを使用して変数を明示的に指定する (手動でトリガーされる builds のみ)

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 builds を提供できます。これにより、プライベートコンテナ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 Dedicated コンテナイメージレジストリーでは、OpenShift Dedicated が自動的にシークレットを生成するので、この作業は必要ありません。

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

```
auths:
  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. 次のコマンドを入力して、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
```

これにより、**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、builds 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イメージレジストリーからビルダーコンテナ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 builds 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、builds 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、ビルドで使用されるサービスアカウントにプルシークレットをリンクするには、次のコマンドを入力します。

■

```
$ 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、場合によっては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注入できます。



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 config map 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 config map](#)

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 に自動的に注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する必要があります。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注入できます。

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関する説明は、以下のセクションを使用します。

4.1. ビルド出力

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

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
OPENSIFT_BUILD_NAME	ビルドの名前
OPENSIFT_BUILD_NAMESPACE	ビルドの namespace
OPENSIFT_BUILD_SOURCE	ビルドのソース URL
OPENSIFT_BUILD_REFERENCE	ビルドで使用する Git 参照
OPENSIFT_BUILD_COMMIT	ビルドで使用するソースコミット

また、S2I 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

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。

Label	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。

ビルドされたイメージのカスタムラベル

```
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 Dedicated は Buildah を使用して Dockerfile からコンテナイメージを構築します。Dockerfile を使用したコンテナイメージのビルドの詳細は、[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](#) を参照してください。

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** パラメーター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** 命令を **BuildConfig** オブジェクトの **from** パラメーターに置き換えるには、**BuildConfig** オブジェクトに次の設定を追加します。

```
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

- ビルドの **dockerfilePath** フィールドを設定して、Dockerfile を見つけるために別のパスを使用します。

```
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。

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: "version"
  value: "latest"
```



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、config map、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
        mounts:
          - destinationPath: /opt/app-root/src/.ssh
            source:
              type: Secret
              secret:
                secretName: my-secret
      - name: settings-mvn
        mounts:
          - destinationPath: /opt/app-root/src/.m2
            source:
              type: ConfigMap
              configMap:
                name: my-config
```

ここでは、以下のようになります。

name

一意の名前を指定します。

destinationPath

マウントポイントの絶対パスを指定します。.. 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。

type

ソースのタイプ (**ConfigMap**、**Secret**、または **CSI**) を指定します。

secretName

ソースの名前を指定します。

関連情報

- [ビルド入力](#)
- [入力シークレットおよび config map](#)

5.2. SOURCE-TO-IMAGE ビルド

Source-to-Image (S2I) は再現可能なコンテナ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イメージに注入し、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 builds 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" ❶
    incremental: true ❷
```

- ❶ 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。
- ❷ このフラグでは、増分ビルドを試行するかどうかを制御します。ビルダーイメージで増分ビルドがサポートされていない場合は、ビルドは成功しますが、**save-artifacts** スクリプトがないため、増分ビルドに失敗したというログメッセージが表示されます。

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は、S2I 要件を参照してください。

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) スクリプトをオーバーライドできます。

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 スクリプトをオーバーライドするには、次のいずれかのアクションを実行します。
 - アプリケーションのソースリポジトリの **.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。
 - **BuildConfig** オブジェクトの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 ビルドプロセスでは、**run**、**assemble**、**save-artifacts** パスに追加されます。これらの名前を持つスクリプトのいずれかまたはすべてが存在する場合、ビルドプロセスでは、イメージで提供されている同じ名前のスクリプトの代わりにこれらのスクリプトが使用されます。



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

環境変数をソースビルドプロセスと結果のイメージで使用できるようにするには、環境ファイルと **BuildConfig** 環境値の 2 つの方法があります。どちらの方法でも指定した変数は、ビルドプロセス中および出カ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は、各イメージのイメージの使用に関する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
assemble	<p>assemble スクリプトは、ソースからアプリケーションアーティファクトをビルドし、イメージ内の適切なディレクトリーに配置します。このスクリプトが必要です。このスクリプトのワークフローは以下のとおりです。</p> <ol style="list-style-type: none"> 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	<p>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。</p>

スクリプト	説明
save-artifacts	<p>save-artifacts スクリプトは、次に続くビルドプロセスを加速できるようにすべての依存関係を収集します。このスクリプトはオプションです。以下に例を示します。</p> <ul style="list-style-type: none"> ● Ruby の場合は、Bundler でインストールされる gems ● Java の場合は、.m2 のコンテンツ <p>これらの依存関係は tar ファイルに集められ、標準出力としてストリーミングされます。</p>
usage	<p>usage スクリプトでは、ユーザーに、イメージの正しい使用方法を通知します。このスクリプトはオプションです。</p>
test/run	<p>test/run スクリプトでは、イメージが正しく機能しているかどうかを確認するための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。</p> <ol style="list-style-type: none"> 1. イメージをビルドします。 2. イメージを実行して usage スクリプトを検証します。 3. s2i build を実行して assemble スクリプトを検証します。 4. 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。 <div style="display: flex; align-items: flex-start; margin-top: 20px;"> <div style="flex: 1; border: 1px solid #ccc; background: repeating-linear-gradient(45deg, transparent, transparent 2px, #ccc 2px, #ccc 4px); margin-right: 10px;"></div> <div style="flex: 2;"> <p>注記</p> <p>test/run スクリプトでビルドしたテストアプリケーションを配置するための推奨される場所は、イメージリポジトリの test/test-app ディレクトリーです。</p> </div> </div>

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。各例では、**tar** の内容が **/tmp/s2i** ディレクトリーに展開されていることを前提としています。

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、config map、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
        mounts:
          - destinationPath: /opt/app-root/src/.ssh
            source:
              type: Secret
              secret:
                secretName: my-secret
      - name: settings-mvn
        mounts:
          - destinationPath: /opt/app-root/src/.m2
            source:
              type: ConfigMap
              configMap:
                name: my-config
```

ここでは、以下のようになります。

name

一意の名前を指定します。

destinationPath

マウントポイントの絶対パスを指定します。.. 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です。

type

ソースのタイプ (**ConfigMap**、**Secret**、または **CSI**) を指定します。

secretName

ソースの名前を指定します。

関連情報

- [ビルド入力](#)
- [入力シークレットおよび config map](#)

5.3. PIPELINE ビルド



重要

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

OpenShift Dedicated 上の Jenkins イメージは完全にサポートされており、ユーザーは Jenkins ユーザードキュメントに従ってジョブで **jenkinsfile** を定義するか、ソースコントロール管理システムに保存する必要があります。

開発者は、Pipeline 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。このビルドは他のビルドタイプの場合と同様に OpenShift Dedicated での起動、モニタリング、管理が可能です。

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

5.3.1. OpenShift Dedicated Pipeline について



重要

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

OpenShift Dedicated 上の Jenkins イメージは完全にサポートされており、ユーザーは Jenkins ユーザードキュメントに従ってジョブで **jenkinsfile** を定義するか、ソースコントロール管理システムに保存する必要があります。

Pipeline により、OpenShift Dedicated でのアプリケーションのビルド、デプロイ、およびプロモートに対する制御が可能になります。Jenkins Pipeline ビルドストラテジー、**jenkinsfiles**、および Jenkins クライアントプラグインによって提供される OpenShift Dedicated ドメイン固有言語 (DSL) を組み合わせて使用することで、あらゆるシナリオに対応した高度なビルド、テスト、デプロイ、およびプロモートパイプラインを作成できます。

OpenShift Dedicated Jenkins 同期プラグイン

OpenShift Dedicated Jenkins Sync プラグインは、ビルド設定とビルドオブジェクトを Jenkins ジョブおよびビルドと同期させ、次の機能を提供します。

- 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 config map からのエージェント Pod テンプレートの動的作成。
- 環境変数の注入。
- OpenShift Dedicated Web コンソールでのパイプラインの視覚化。
- Jenkins Git プラグインとの統合。これにより、OpenShift Dedicated ビルドからのコミット情報が Jenkins Git プラグインに渡されます。
- シークレットを Jenkins 認証情報エントリに同期。

OpenShift Dedicated Jenkins Client プラグイン

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

Jenkins クライアントプラグインを Jenkins マスターにインストールして、アプリケーションの **jenkinsfile** 内で OpenShift Dedicated DSL を使用できるようにする必要があります。このプラグインは、OpenShift Dedicated Jenkins イメージを使用する場合、デフォルトでインストールされ、有効になります。

プロジェクト内で OpenShift Dedicated Pipeline を使用するには、Jenkins Pipeline ビルドストラテジーを使用する必要があります。このストラテジーはソースリポジトリのルートで **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。

5.3.2. パイプライン builds 用の Jenkins ファイルの提供



重要

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

OpenShift Dedicated 上の Jenkins イメージは完全にサポートされており、ユーザーは Jenkins ユーザードキュメントに従ってジョブで **jenkinsfile** を定義するか、ソースコントロール管理システムに保存する必要があります。

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に組み込む必要があります。

- リポジトリのルートにある **jenkinsfile** という名前のファイル。
- リポジトリのソース **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 (指定した場合)。指定しないと、リポジトリ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 上で実行されます。OpenShift Dedicated DSL を使用する場合は、OpenShift Dedicated クライアントバイナリーが利用可能になっている必要があります。

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 ❶
```

- ❶ 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。

5.3.3. パイプライン builds の環境変数の使用



重要

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

OpenShift Dedicated 上の Jenkins イメージは完全にサポートされており、ユーザーは Jenkins ユーザードキュメントに従ってジョブで **jenkinsfile** を定義するか、ソースコントロール管理システムに保存する必要があります。

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** 定義に追加します。

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。

5.3.3.1. 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。

5.3.4. Pipeline ビルドのチュートリアル



重要

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

OpenShift Dedicated 上の Jenkins イメージは完全にサポートされており、ユーザーは Jenkins ユーザードキュメントに従ってジョブで **jenkinsfile** を定義するか、ソースコントロール管理システムに保存する必要があります。

この例では **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、デプロイ、検証する OpenShift Dedicated Pipeline を作成する方法を示します。

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 パイプライン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、パイプラインに指示を出します。



注記

この例では、アプリケーションに Git リポジトリを設定しません。

以下の **jenkinsfile** の内容は、OpenShift Dedicated DSL を使用して Groovy で記述されています。ソースリポジトリに **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' ❶
def templateName = 'nodejs-mongodb-example' ❷
pipeline {
  agent {
    node {
      label 'nodejs' ❸
    }
  }
  options {
    timeout(time: 20, unit: 'MINUTES') ❹
  }
  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() ❺
              if (openshift.selector("secrets", templateName).exists()) { ❻
                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 このパイプラインに 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

前述の例は、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 Dedicated クラスタに 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 Dedicated 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 コンソールにアクセスして作成したパイプラインに移動して **Build Now** をクリックし、パイプラインを開始することもできます。

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
- パイプラインが必要な場合には、エージェント Pod が起動される

- パイプラインがエージェント 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。
 - パイプラインは、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。
- エージェント Pod は、パイプラインで以前に要求されていた場合には削除される



注記

OpenShift Web コンソールで表示すると、最適な方法で Pipeline の実行を視覚的に視覚化できます。Web コンソールにログインして、Builds → Pipelines に移動し、パイプラインを確認します。

5.4. 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 Dedicated Web コンソールからプライベートリポジトリ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、次の手順を実行します。

1. 新規の OpenShift Dedicated プロジェクトを作成します。
2. プライベートのソースコードリポジトリ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。

5.5. 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 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関する方法を説明します。

6.1. ビルドの開始

現在のプロジェクトに既存のビルド設定から新規ビルドを手動で起動できます。

手順

- ビルドを手動で開始するには、次のコマンドを入力します。

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

6.1.1. ビルドの再実行

--from-build フラグを使用してビルドを手動で再度実行します。

手順

- 手動でビルドを再実行するには、以下のコマンドを入力します。

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

6.1.2. ビルドログのストリーミング

--follow フラグを指定して、**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
```

6.1.3. ビルド開始時の環境変数の設定

--env フラグを指定して、ビルドの任意の環境変数を設定します。

手順

- 必要な環境変数を指定するには、以下のコマンドを実行します。

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

6.1.4. ソースを使用したビルドの開始

Git ソースプル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オーバーライドされます。



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

- ソースコードリポジトリーからビルドを開始し、ローカル Git リポジトリーの内容をタグ **v2** からアーカイブとして送信するには、次のコマンドを入力します。

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

6.2. ビルドの中止

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

手順

- 手動でビルドを取り消すには、以下のコマンドを入力します。

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

6.2.1. 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>
```

6.2.2. 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>
```

6.2.3. 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>
```

6.3. BUILDCONFIG の編集

ビルド設定を編集するには、Builds ページの **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. Builds ページで Options メニュー  をクリックして **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 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を指定します。

- **Source Secret** プライベートリポジトリ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ます。

7. **Save** をクリックして **BuildConfig** を保存します。

6.4. 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>
```

6.5. 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 **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も含まれます。

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>
```

6.6. 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>
```

6.6.1. 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>
```

6.6.2. 特定バージョンのビルドに関する 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** に関する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>
```

6.6.3. 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** の一部として、**BUILD_LOGLEVEL** 環境変数を指定します。



注記

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

ソース builds 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7章 BUILDS 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 builds をトリガーし、builds を変更する方法に関する概要を説明します。

7.1. ビルドトリガー

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

- Webhook
- イメージの変更
- 設定の変更

7.1.1. Webhook のトリガー

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

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

プッシュイベントを処理する場合に、OpenShift Dedicated 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かを確認します。一致する場合には、OpenShift Dedicated ビルドの Webhook イベントに記載されているのと全く同じコミット参照がチェックアウトされます。一致しない場合には、ビルドはトリガーされません。



注記

oc new-app および **oc new-build** は GitHub および Generic Webhook トリガーを自動的に作成しますが、それ以外の Webhook トリガーが必要な場合には手動で追加する必要があります。トリガーを設定して、トリガーを手動で追加できます。

すべての Webhook に対して、**WebHookSecretKey** という名前のキーと、Webhook の呼び出し時に指定する値を含むシークレット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** フィールドに必要な点に注意してください。

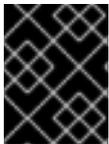
```
- kind: Secret
  apiVersion: v1
  metadata:
    name: mysecret
    creationTimestamp:
  data:
    WebHookSecretKey: c2VjcmV0dmFsdWUx
```

7.1.1.1. 認証されていないユーザーをシステムに追加する:Webhook ロールバインディング

クラスター管理者は、特定の namespace の OpenShift Dedicated の **system:webhook** ロールバインディングに認証されていないユーザーを追加できます。**system:webhook** ロールバインディングを使用すると、ユーザーは OpenShift Dedicated 認証メカニズムを使用しない外部システムからビルドをトリガーできます。認証されていないユーザーは、デフォルトでは非パブリックロールバインディングにアクセスできません。これは、OpenShift Dedicated バージョン 4.16 より前のバージョンからの変更点です。

GitHub、GitLab、Bitbucket からの builds を正常にトリガーするには、認証されていないユーザーを **system:webhook** ロールバインディングに追加する必要があります。

認証されていないユーザーにクラスターへのアクセスを許可する必要がある場合は、必要な各 namespace の **system:webhook** ロールバインディングに認証されていないユーザーを追加することでこれを実行できます。この方法は、認証されていないユーザーを **system:webhook** クラスターロールバインディングに追加するよりもセキュアです。ただし、namespace の数が多い場合は、認証されていないユーザーを **system:webhook** クラスターロールバインディングに追加して、すべての namespace に変更を適用することができます。



重要

認証されていないアクセスを変更するときは、常に組織のセキュリティ標準に準拠していることを確認してください。

前提条件

- **cluster-admin** ロールを持つユーザーとしてクラスターにアクセスできる。
- OpenShift CLI (**oc**) がインストールされている。

手順

1. **add-webhooks-unauth.yaml** という名前の YAML ファイルを作成し、次のコンテンツを追加します。

```
apiVersion: rbac.authorization.k8s.io/v1
kind: RoleBinding
metadata:
  annotations:
    rbac.authorization.kubernetes.io/autoupdate: "true"
  name: webhook-access-unauthenticated
  namespace: <namespace> ①
roleRef:
  apiGroup: rbac.authorization.k8s.io
  kind: ClusterRole
  name: "system:webhook"
```

```
subjects:
- apiGroup: rbac.authorization.k8s.io
  kind: Group
  name: "system:unauthenticated"
```

1 **BuildConfig** の namespace。

2. 以下のコマンドを実行して設定を適用します。

```
$ oc apply -f add-webhooks-unauth.yaml
```

関連情報

- [認証されていないグループのクラスターロールバインディング](#)

7.1.1.2. GitHub Webhook の使用

GitHub webhook は、リポジトリの更新時に 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 トリガー設定のシークレットにより、Webhook URL が一意になり、予測が困難になります。GitHub UI で設定されるシークレットは、**X-Hub-Signature** ヘッダーとして送信される本文の HMAC 16 進ダイジェストを作成するために使用されるオプションの文字列フィールドです。

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 リポジトリから **BuildConfig** を作成します。
- **system:unauthenticated** は、必要な namespace 内の **system:webhook** ロールにアクセスできます。または、**system:unauthenticated** は **system:webhook** クラスターロールにアクセスできます。

手順

1. GitHub Webhook を設定します。

- a. GitHub リポジトリから **BuildConfig** オブジェクトを作成した後、次のコマンドを実行します。

```
$ oc describe bc/<name_of_your_BuildConfig>
```

このコマンドは、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/<name>/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 ペイロードを含むファイルを指定すると、次の **curl** コマンドを使用して 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>/buildconfigs/<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](#)

7.1.1.3. GitLab Webhook の使用

GitLab Webhook は、リポジトリの更新時の 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
```

前提条件

- **system:unauthenticated** は、必要な namespace 内の **system:webhook** ロールにアクセスできます。または、**system:unauthenticated** は **system:webhook** クラスターロールにアクセスできます。

手順

1. GitLab Webhook を設定します。
 - a. 次のコマンドを入力して、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. [GitLab の設定手順](#) に従い、GitLab リポジトリの設定に Webhook URL を貼り付けます。
2. **payload.json** などの有効な JSON ペイロードを含むファイルを指定すると、次の **curl** コマンドを使用して 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。

7.1.1.4. Bitbucket Webhook の使用

Bitbucket webhook は、リポジトリの更新時の Bitbucket による呼び出しを処理します。GitHub および GitLab 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
```

前提条件

- **system:unauthenticated** は、必要な namespace 内の **system:webhook** ロールにアクセスできます。または、**system:unauthenticated** は **system:webhook** クラスターロールにアクセスできます。

手順

1. Bitbucket Webhook を設定します。
 - a. 次のコマンドを入力して、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 ペイロードを含むファイルがある場合は、次の **curl** コマンドを入力して 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。

7.1.1.5. 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。

一般的な Webhook エンドポイント URL の例

```

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

```

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

2. Webhook を手動で呼び出すには、次の **curl** コマンドを入力します。

```

$ curl -X POST -k
https://<openshift_api_host:port>/apis/build.openshift.io/v1/namespaces/<namespace>/buildconfigs/<name>/webhooks/<secret>/generic

```

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

エンドポイントは、以下の形式で任意のペイロードを受け入れることができます。

```

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

```

- ① **BuildConfig** 環境変数と同様に、ここで定義されている環境変数はビルドで利用できません。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で定義し、次のコマンドを実行します。

```
$ 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 Dedicated では、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、無効なリクエストペイロードが提示された場合でも、汎用 Webhook によってビルドがトリガーされることが許可されます。この動作は、後方互換性を確保するために継続されています。無効な要求ペイロードがある場合には、OpenShift Dedicated は、**HTTP 200 OK** 応答の一部として JSON 形式で警告を返します。

7.1.1.6. Webhook URL の表示

oc describe コマンドを使用して、ビルド設定に関連付けられた Webhook URL を表示できます。コマンドが Webhook URL を表示しない場合、そのビルド設定に現在定義される Webhook トリガーはありません。

手順

- **BuildConfig** に関連付けられているすべての Webhook URL を表示するには、次のコマンドを実行します。

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

7.1.2. イメージ変更トリガーの使用

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 コンテナレジストリー のコンテナ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イメージリポジトリに紐付けられます。`<system-registry>` は、OpenShift Dedicated で実行する `docker-registry` の名前、サービスとして定義されます。

- 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。

- `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" ❶
imageChange: {}
type: "ImageChange" ❷
imageChange:
  from:
    kind: "ImageStreamTag"
    name: "custom-image:latest"
```

- 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。この `imageChange` オブジェクトは空でなければなりません。
- 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、ストラテジーに関する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
```

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 builds が実行されるようになります。

関連情報

- [v1 コンテナレジストリー](#)

7.1.3. 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@sha256:0f88ffbeb9d25525720bfa3524cb2ce0908b7f791057cf1acfae917b11266a69

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. **status.imageChangeTriggers** の **BuildConfig** CR で、最新のタイムスタンプを持つ **lastTriggerTime** を特定します。
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には示されていませんが、OpenShift Dedicated 4.8 では非推奨となり、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、次の注目すべきフィールドがあります。

- **kind:** 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 コンテナレジストリー](#)

7.1.4. 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。

7.1.4.1. 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、次のコマンドを入力します。

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

- イメージ変更トリガーを設定するには、次のコマンドを入力します。

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

- トリガーを削除するには、次のコマンドを入力します。

```
$ 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。

詳細は、次のコマンドを入力してヘルプドキュメントを参照してください。

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

7.2. ビルドフック

ビルドフックを使用すると、ビルドプロセスに動作を注入できます。

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

7.2.1. コミット後のビルドフックの設定

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

手順

- ビルド後のフックを設定するには、次のいずれかのオプションを使用します。

オプション	説明
シェルスクリプト	<pre>postCommit: script: "bundle exec rake test --verbose"</pre> <p>script の値は、<code>/bin/sh -ic</code> 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、このオプションを使用します。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 <code>/bin/sh</code> がない場合は、command または args、もしくは両方を使用します。</p> <div data-bbox="868 788 975 1016" style="float: left; width: 60px; height: 100px; background: repeating-linear-gradient(45deg, transparent, transparent 2px, #ccc 2px, #ccc 4px);"></div> <p>注記</p> <p>CentOS や RHEL イメージでの作業を改善するために、追加で <code>-i</code> 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。</p>
イメージエントリーポイントとしてのコマンド:	<pre>postCommit: command: ["/bin/bash", "-c", "bundle exec rake test --verbose"]</pre> <p>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オーバーライドします。これはイメージに <code>/bin/sh</code> がない場合、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。</p>
引数のあるコマンド:	<pre>postCommit: command: ["bundle", "exec", "rake", "test"] args: ["--verbose"]</pre> <p>この形式は command に引数を追加するのと同じです。</p>



注記

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

7.2.2. CLI を使用したコミット後のビルドフックの設定

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

手順

1. 以下のアクションの1つを完了します。

- コマンドをコミット後のビルドフックとして設定するには、次のコマンドを入力します。

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

- スクリプトをコミット後のビルドフックとして設定するには、次のコマンドを入力します。

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

第8章 高度なビルドの実行

ビルドリソースと最大期間を設定したり、ビルドをノードに割り当てたり、ビルドをチェーンしたり、ビルドを削減したり、ビルド実行ポリシーを設定したりすることができます。

8.1. 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制限でリソース制限を指定して、リソースを制限します。
- ビルド設定の一部としてリソース制限を指定して、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** の各パラメーターはオプションです。

```
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力され、ビルド Pod の作成は失敗します。

8.2. 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。指定されたタイムアウトに達すると、OpenShift Dedicated によってビルドが終了します。

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、Pipeline Strategy オプションではサポートされていません。

8.3. 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、値をオーバーライドできます。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。値をオーバーライドすると、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。

8.4. 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。

8.5. 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**、**canceled** 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。

8.6. 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9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 Dedicated 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。

9.1. 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

- 単一のプロジェクトで **ImageStreamTag** リソースを作成するには、次のコマンドを入力します。

```
$ 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** リソースを作成できます。

```
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
```

9.2. 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 builds 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

- **cluster-admin** ロールを持つユーザーとしてクラスターにアクセスできる必要があります。または、**openshift-config-managed** プロジェクト内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. 次のコマンドを実行して、**openshift-config-managed** namespace からビルドの namespace にエンタイトルメントシークレットをコピーします。

```
$ 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
```

9.3. SUBSCRIPTION MANAGER を使用した BUILDS の実行

9.3.1. Subscription Manager を使用した Docker builds

Docker ストラテジー builds 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 ❶
RUN yum --enablerepo=codeready-builder-for-rhel-9-x86_64-rpms install \ ❷
    nss_wrapper \
    uid_wrapper -y && \
    yum clean all -y
RUN ln -s /run/secrets/rhsm /etc/rhsm-host ❸
```

- ❶ **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。
- ❷ **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を見つけます。
- ❸ イメージと他の Red Hat コンテナ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。

9.4. RED HAT SATELLITE サブスクリプションを使用した BUILDS の実行

9.4.1. Red Hat Satellite 設定の builds 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 builds は、Satellite リポジトリ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

- 次のコマンドを入力して、Satellite リポジトリ設定ファイルを含む **ConfigMap** オブジェクトを作成します。

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

- Satellite リポジトリ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

```

9.4.2. Red Hat Satellite サブスクリプションを使用した Docker builds

Docker ストラテジー builds は、Red Hat Satellite リポジトリ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 リポジトリ設定がビルドボリュームとして追加しておく。

手順

- 次の例を使用して、Satellite を使用してコンテンツをインストールするための **Dockerfile** を作成します。

```

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

- [How to use builds with Red Hat Satellite subscriptions and which certificate to use](#)

9.5. 関連情報

- [イメージストリームの管理](#)
- [ビルドストラテジー](#)

第10章 BUILDS 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。

10.1. 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
```

10.2. 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。アノテーションをクリアするには、次のコマンドを入力します。

```
$ 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。