# OpenShift Container Platform 4.7

## CI/CD

OpenShift Container Platform のビルド、パイプライン、および GitOps に関する情報

Last Updated: 2022-11-29

# OpenShift Container Platform 4.7 CI/CD

OpenShift Container Platform のビルド、パイプライン、および GitOps に関する情報

Enter your first name here. Enter your surname here.
Enter your organisation's name here. Enter your organisational division here.
Enter your email address here.

## 法律上の通知



## 概要

OpenShift Container Platform 向けの CI/CD

# 目次

# 第1章 OPENSHIFT CONTAINER PLATFORM CI/CD の概要

OpenShift Container Platform は、開発者向けのエンタープライズ対応の Kubernetes プラットフォームであり、組織は継続的インテグレーション (CI) や継続的デリバリー (CD) などの DevOps プラクティスを通じてアプリケーションデリバリープロセスを自動化できます。組織のニーズを満たすために、OpenShift Container Platform は以下の CI/CD ソリューションを提供します。

- OpenShift Builds
- OpenShift Pipeline
- OpenShift GitOps

## 1.1. OPENSHIFT BUILDS

OpenShift Builds を使用すると、宣言型ビルドプロセスを使用してクラウドネイティブアプリを作成できます。BuildConfig オブジェクトの作成に使用する YAML ファイルでビルドプロセスを定義できます。この定義には、ビルドトリガー、入力パラメーター、ソースコードなどの属性が含まれます。デプロイされると、BuildConfig オブジェクトは通常、実行可能なイメージをビルドし、それをコンテナーイメージレジストリーにプッシュします。

OpenShift Builds は、ビルドストラテジーに対して以下の拡張可能なサポートを提供します。

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

詳細は、イメージビルドについて を参照してください。

## 1.2. OPENSHIFT PIPELINE

OpenShift Pipelines は、Kubernetes ネイティブの CI/CD フレームワークを提供して、CI/CD パイプラインの各ステップを独自のコンテナーで設計および実行します。独立して拡張し、予測可能な結果を伴うオンデマンドパイプラインに対応できます。

詳細は、OpenShift Pipelines について を参照してください。

## 1.3. OPENSHIFT GITOPS

OpenShift GitOps は、宣言型 GitOps エンジンとして Argo CD を使用するオペレーターです。これにより、マルチクラスター OpenShift および Kubernetes インフラストラクチャー全体で GitOps ワークフローが可能になります。管理者は、OpenShift GitOps を使用して、クラスターおよび開発ライフサイクル全体で Kubernetes ベースのインフラストラクチャーとアプリケーションを一貫して設定およびデプロイできます。

OpenShift GitOps について を参照してください。

## 1.4. JENKINS

Jenkins は、アプリケーションとプロジェクトの構築、テスト、およびデプロイのプロセスを自動化します。OpenShift Developer Tools は、OpenShift Container Platform と直接統合する Jenkins イメージを提供します。Jenkins は、Samples Operator テンプレートまたは認定 Helm チャートを使用して

OpenShift にデプロイできます。

# 第2章 ビルド

## 2.1. イメージビルドについて

### 2.1.1. ビルド

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

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

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

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

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

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

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

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

- 継続的インテグレーション
- 継続的デプロイメント

#### 2.1.1.1. Docker ビルド

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

**ヒント**

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

#### 2.1.1.2. Source-to-Image ビルド

Source-to-Image (S2I) は再現可能なコンテナーイメージをビルドするためのツールです。これはアプ

リケーションソースをコンテナーイメージに挿入し、新規イメージをアセンブルして実行可能なイメージを生成します。新規イメージはベースイメージ、ビルダーおよびビルドされたソースを組み込み、**buildah run** コマンドで使用することができます。S2I は増分ビルドをサポートします。これは以前にダウンロードされた依存関係や、以前にビルドされたアーティファクトなどを再利用します。

#### 2.1.1.3. カスタムビルド

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

カスタムビルダーイメージは、RPM またはベースイメージの構築など、ビルドプロセスのロジックに組み込まれるプレーンなコンテナーイメージです。

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

#### 2.1.1.4. パイプラインビルド

**重要**

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

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

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

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

## 2.2. 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。

### 2.2.1. BuildConfig

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

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

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

更する場合に役立ちます。

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

**BuildConfig のオブジェクト定義**

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

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

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

(3) 新規ビルドを作成するトリガーの一覧を指定できます。

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

(5) **strategy** セクションでは、ビルドの実行に使用するビルドストラテジーを記述します。ここでは **Source**、**Docker** または **Custom** ストラテジーを指定できます。上記の例では、Source-to-image (S2I) がアプリケーションのビルドに使用する **ruby-20-centos7** コンテナーイメージを使用します。

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

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

## 2.3. ビルド入力の作成

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

### 2.3.1. ビルド入力

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

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

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

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

ビルドを実行すると、以下が行われます。

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

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

```
source:
  git:
    uri: https://github.com/openshift/ruby-hello-world.git (1)
    ref: "master"
  images:
  - from:
      kind: ImageStreamTag
      name: myinputimage:latest
      namespace: mynamespace
    paths:
    - destinationDir: app/dir/injected/dir (2)
      sourcePath: /usr/lib/somefile.jar
  contextDir: "app/dir" (3)
  dockerfile: "FROM centos:7\nRUN yum install -y httpd" (4)
```

(1) 作業ディレクトリーにクローンされるビルド用のリポジトリー

(2) **myinputimage** の **/usr/lib/somefile.jar** は、**<workingdir>/app/dir/injected/dir** に保存されます。

(3) ビルドの作業ディレクトリーは **<original_workingdir>/app/dir** になります。

(4) このコンテンツを含む Dockerfile は **<original_workingdir>/app/dir** に作成され、この名前が指定された既存ファイルは上書きされます。

### 2.3.2. Dockerfile ソース

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

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

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

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

**関連情報**

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

### 2.3.3. イメージソース

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

ソースパスは、指定したイメージ内の絶対パスで指定してください。宛先は、相対ディレクトリーパス

でなければなりません。ビルド時に、イメージは読み込まれ、指定のファイルおよびディレクトリーはビルドプロセスのコンテキストディレクトリーにコピーされます。これは、ソースリポジトリーのコンテンツのクローンが作成されるディレクトリーと同じです。ソースパスの末尾は /. であり、ディレクトリーのコンテンツがコピーされますが、ディレクトリー自体は宛先で作成されません。

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

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

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

**注記**

クラスターが **ImageContentSourcePolicy** オブジェクトを使用してリポジトリーの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。

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

-

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

**注記**

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

### 2.3.4. Git ソース

ソースコードは、指定されている場合は指定先の場所からフェッチされます。

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

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

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

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

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

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

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

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

**警告**

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

#### 2.3.4.1. プロキシーの使用

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

**注記**

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

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

**注記**

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

**関連情報**

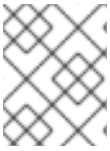
- Jenkins UI でのプロキシーの設定方法については、JenkinsBehindProxy を参照してください。

#### 2.3.4.2. ソースクローンのシークレット

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

以下は、サポートされているソースクローンのシークレット設定です。

- .gitconfig ファイル
- Basic 認証
- SSH キー認証
- 信頼されている認証局



**注記**

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

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

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

この機能を使用するには、Git リポジトリーの認証情報を含むシークレットが **BuildConfig** が後に作成される namespace になければなりません。このシークレットには、接頭辞 **build.openshift.io/source-secret-match-uri-** で開始するアノテーション 1 つ以上含まれている必要もあります。これらの各アノテーションの値には、以下で定義される URI (Uniform Resource Identifier) パターンを使用します。これは以下のように定義されます。ソースクローンのシークレット参照なしに **BuildConfig** が作成され、Git ソースの URI がシークレットのアノテーションの URI パターンと一致する場合に、OpenShift Container Platform はそのシークレットへの参照を **BuildConfig** に自動的に挿入します。

**前提条件**

URI パターンには以下を含める必要があります。

- 有効なスキーム: ***://**、 **git://**、**http://**、**https://** または **ssh://**
- ホスト: *` または有効なホスト名、あるいは ***.** が先頭に指定された IP アドレス
- パス: /* または、 / の後に * 文字などの文字がオプションで後に続きます。

上記のいずれの場合でも、* 文字はワイルドカードと見なされます。

**重要**

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

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

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

**手順**

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

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

```
kind: Secret
apiVersion: v1
metadata:
  name: matches-all-corporate-servers-https-only
  annotations:
```

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

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

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

#### 2.3.4.2.2. ソースクローンシークレットの手動による追加

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

```
apiVersion: "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
```

#### 2.3.4.2.3. .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
```

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

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

**前提条件**

- Git 認証情報が必要です。

**手順**

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

1. **client.crt**、**cacert.crt**、および **client.key** ファイルをアプリケーションソースコードの **/var/run/secrets/openshift.io/source/** フォルダーに追加します。
2. サーバーの **.gitconfig** ファイルに、以下のように **[http]** セクションを追加します。

```
# cat .gitconfig
```

**出力例**

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

3. シークレットを作成します。

```
$ oc create secret generic <secret_name> \
--from-literal=username=<user_name> \ 1
--from-literal=password=<password> \ 2
--from-file=.gitconfig=.gitconfig \
--from-file=client.crt=/var/run/secrets/openshift.io/source/client.crt \
--from-file=cacert.crt=/var/run/secrets/openshift.io/source/cacert.crt \
--from-file=client.key=/var/run/secrets/openshift.io/source/client.key
```

1. ユーザーの Git ユーザー名
2. このユーザーのパスワード

**重要**

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

**関連情報**

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

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

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

**前提条件**

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

**手順**

1. シークレットを先に作成してから、プライベートリポジトリーにアクセスするために **--username** および **--password** を使用してください。

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

2. トークンで Basic 認証のシークレットを作成します。

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

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

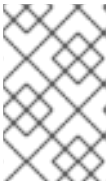
SSH キーベースの認証では、プライベート SSH キーが必要です。

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

**手順**

1. SSH キーの認証情報を生成します。

```
$ ssh-keygen -t ed25519 -C "your_email@example.com"
```



**注記**

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

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

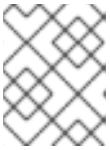
2. SSH キーを使用してプライベートリポジトリーにアクセスする前に、シークレットを作成します。

```
$ oc create secret generic <secret_name> \
    --from-file=ssh-privatekey=<path/to/ssh/private/key> \
    --from-file=<path/to/known_hosts> \ 1
    --type=kubernetes.io/ssh-auth
```

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

**警告**

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



**注記**

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

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

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

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

**手順**

CA 証明書ファイルでシークレットを作成します。

1. CA が中間認証局を使用する場合には、**ca.crt** ファイルにすべての CA の証明書を統合します。以下のコマンドを入力します。

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

   a. シークレットを作成します。

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

   1 **ca.crt** というキーの名前を使用する必要があります。

#### 2.3.4.2.8. ソースシークレットの組み合わせ

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

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

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

**前提条件**

- SSH 認証
- .gitconfig ファイル

**手順**

- **.gitconfig** ファイルを使って SSH ベースの認証シークレットを作成するには、以下を実行します。

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

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

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

**前提条件**

- .gitconfig ファイル

- CA 証明書

**手順**

- **.gitconfig** ファイルと CA 証明書を組み合わせてシークレットを作成するには、以下を実行します。

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

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

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

**前提条件**

- Basic 認証の認証情報
- CA 証明書

**手順**

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

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

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

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

**前提条件**

- Basic 認証の認証情報
- **.gitconfig** ファイル

**手順**

- **.gitconfig** ファイルで Basic 認証のシークレットを作成するには、以下を実行します。

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

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

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

**前提条件**

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

**手順**

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

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

### 2.3.5. バイナリー (ローカル) ソース

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

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

**注記**

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

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

- **--from-file**: 指定したファイルのコンテンツはバイナリーストリームとしてビルダーに送信されます。ファイルに URL を指定することもできます。次に、ビルダーはそのデータをビルドコンテキストの上に、同じ名前のファイルに保存します。
- **--from-dir** および **--from-repo**: コンテンツはアーカイブされて、バイナリーストリームとしてバイナリーに送信されます。次に、ビルダーはビルドコンテキストディレクトリー内にアーカイブのコンテンツを展開します。**--from-dir** を使用して、展開されるアーカイブに URL を指定することもできます。

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

上記のそれぞれの例では、以下のようになります。

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

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

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

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

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

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

バイナリーストリームが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。

### 2.3.6. 入力シークレットおよび設定マップ

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

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

1. ミラーの URL および接続の設定が含まれる **settings.xml** ファイル。

2. ~/**.ssh/id_rsa** などの、設定ファイルで参照されるプライベートキー。

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

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

#### 2.3.6.1. シークレットの概要

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

**YAML シークレットオブジェクト定義**

```
apiVersion: v1
kind: Secret
metadata:
  name: test-secret
  namespace: my-namespace
type: Opaque (1)
data: (2)
  username: dmFsdWUtMQ0K (3)
  password: dmFsdWUtMg0KDQo=
stringData: (4)
  hostname: myapp.mydomain.com (5)
```

(1) シークレットにキー名および値の構造を示しています。

(2) **data** フィールドでキーに使用できる形式は、Kubernetes identifiers glossary の **DNS_SUBDOMAIN** 値のガイドラインに従う必要があります。

(3) **data** マップのキーに関連付けられる値は base64 でエンコーディングされている必要があります。

(4) **stringData** マップのエントリーが base64 に変換され、このエントリーは自動的に **data** マップに移動します。このフィールドは書き込み専用です。値は **data** フィールドによってのみ返されます。

(5) **stringData** マップのキーに関連付けられた値は単純なテキスト文字列で設定されます。

##### 2.3.6.1.1. シークレットのプロパティー

キーのプロパティーには以下が含まれます。

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

##### 2.3.6.1.2. 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**

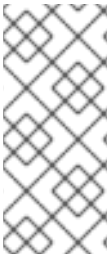
**example.com/my-secret-type** などの他の任意のタイプを指定できます。これらのタイプはサーバー側では実行されませんが、シークレットの作成者がその種類のキー/値の要件に従う意図があることを示します。

#### 2.3.6.1.3. シークレットの更新

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

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

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



**注記**

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

### 2.3.6.2. シークレットの作成

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

シークレットの作成時に以下を実行します。

- シークレットデータでシークレットオブジェクトを作成します。
- Pod のサービスアカウントをシークレットの参照を許可するように更新します。

- シークレットを環境変数またはファイルとして使用する Pod を作成します (**secret** ボリュームを使用)。

**手順**

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

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

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

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

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

**YAML の不透明なシークレットオブジェクトの定義**

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

1 **opaque** シークレットを指定します。

**Docker 設定の JSON ファイルシークレットオブジェクトの定義**

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

.dockerconfigjson:bm5ubm5ubm5ubm5ubm5ubm5ubm5ubmdnZ2dnZ2dnZ2dnZ2dnZ2dnZ2cg
YXV0aCBrZXIzCg== 2
```

1 シークレットが docker 設定の JSON ファイルを使用することを指定します。

2 docker 設定 JSON ファイルを base64 でエンコードした出力

### 2.3.6.3. シークレットの使用

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

**手順**

1. シークレットを参照する Pod を作成します。

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

2. ログを取得します。

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

3. Pod を削除します。

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

**関連情報**

- シークレットデータを含む YAML ファイルのサンプル

**4 つのファイルを作成する YAML シークレット**

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

(1) デコードされる値が含まれるファイル

(2) デコードされる値が含まれるファイル

(3) 提供される文字列が含まれるファイル

(4) 提供されるデータが含まれるファイル

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

```
apiVersion: v1
kind: Pod
metadata:
  name: secret-example-pod
spec:
  containers:
```

```
    - name: secret-test-container
      image: busybox
      command: [ "/bin/sh", "-c", "cat /etc/secret-volume/*" ]
      volumeMounts:
          # name must match the volume name below
          - name: secret-volume
            mountPath: /etc/secret-volume
            readOnly: true
  volumes:
    - name: secret-volume
      secret:
        secretName: test-secret
  restartPolicy: Never
```

**シークレットデータと共に環境変数が設定された Pod の YAML**

```
apiVersion: v1
kind: Pod
metadata:
  name: secret-example-pod
spec:
  containers:
    - name: secret-test-container
      image: busybox
      command: [ "/bin/sh", "-c", "export" ]
      env:
        - name: TEST_SECRET_USERNAME_ENV_VAR
          valueFrom:
            secretKeyRef:
              name: test-secret
              key: username
  restartPolicy: Never
```

**シークレットデータと環境変数を設定するビルド設定の YAML**

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

#### 2.3.6.4. 入力シークレットおよび設定マップの追加

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

**手順**

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

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

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

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

2. **Secret** オブジェクトがない場合はこれを作成します。

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

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

3. 設定マップおよびシークレットを既存の **BuildConfig** オブジェクトの **source** セクションに追加します。

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

シークレットおよび設定マップを新規の **BuildConfig** オブジェクトに追加するには、以下のコマンドを実行します。

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

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

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

```
      destinationDir: ".m2"
  secrets:
    - secret:
        name: secret-mvn
      destinationDir: ".ssh"
```

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

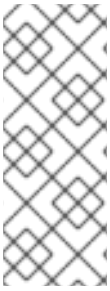
```
$ oc new-build \
    openshift/wildfly-101-centos7~https://github.com/wildfly/quickstart.git \
    --context-dir helloworld --build-secret “secret-mvn:.ssh” \
    --build-config-map "settings-mvn:.m2"
```

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

#### 2.3.6.5. Source-to-Image ストラテジー

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

**destinationDir** が相対パスの場合に同じルールが使用されます。シークレットは、イメージの作業ディレクトリーに相対的なパスに配置されます。**destinationDir** パスの最終ディレクトリーは、ビルダーイメージにない場合に作成されます。**destinationDir** の先行するすべてのディレクトリーは存在している必要があり、そうでない場合にはエラーが生じます。



**注記**

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

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

#### 2.3.6.6. Docker ストラテジー

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

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

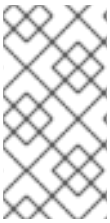
**シークレットおよび設定マップデータを参照する Dockerfile の例**

```
FROM centos/ruby-22-centos7

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

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

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



**注記**

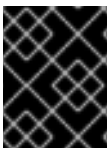
通常はシークレットがイメージから実行するコンテナーに置かれないように、入力シークレットを最終的なアプリケーションイメージから削除します。ただし、シークレットは追加される階層のイメージ自体に存在します。この削除は、Dockerfile の一部として組み込まれます。

#### 2.3.6.7. カスタムストラテジー

Custom ストラテジーを使用する場合、定義された入力シークレットおよび設定マップはすべて、**/var/run/secrets/openshift.io/build** ディレクトリー内のビルダーコンテナーで入手できます。カスタムのビルドイメージは、これらのシークレットおよび設定マップを適切に使用する必要があります。Custom ストラテジーでは、Custom ストラテジーのオプションで説明されているようにシークレットを定義できます。

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

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



**重要**

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

### 2.3.7. 外部アーティファクト

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

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

**.s2i/bin/assemble ファイル**

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

**.s2i/bin/run ファイル**

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

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

### Dockerfile の抜粋

```
FROM jboss/base-jdk:8

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

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

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

環境変数の定義には複数の方法があり、いずれかの方法を選択できます。

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

## 2.3.8. プライベートレジストリーでの docker 認証情報の使用

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

### 注記

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

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

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

1. レジストリーの URL
2. 暗号化されたパスワード

3 ログイン用のメールアドレス

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

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

**前提条件**

- **.docker/config.json** ファイルが必要です。

**手順**

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

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

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

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

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

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

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

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

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

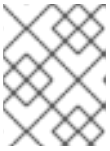
3. ビルドストラテジー定義に含まれる **pullSecret** を指定して、プライベートコンテナーイメージレジストリーからビルダーコンテナーイメージをプルします。

```
strategy:
  sourceStrategy:
    from:
```

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

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

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



**注記**

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

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

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

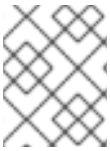
**注記**

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

### 2.3.9. ビルド環境

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

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



**注記**

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

#### 2.3.9.1. 環境変数としてのビルドフィールドの使用

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

**注記**

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

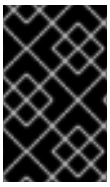
**手順**

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

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

#### 2.3.9.2. 環境変数としてのシークレットの使用

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



**重要**

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

**手順**

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

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

**関連情報**

- 入力シークレットおよび設定マップ

### 2.3.10. サービス提供証明書のシークレット

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

**手順**

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

- 値をシークレットに使用する名前に設定し、**service.beta.openshift.io/serving-cert-secret-name** アノテーションをサービスに設定します。

次に、**PodSpec** はそのシークレットをマウントできます。これが利用可能な場合、Pod が実行されます。この証明書は内部サービス DNS 名、**<service.name>.<service.namespace>.svc** に適しています。

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

**注記**

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

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

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

### 2.3.11. シークレットの制限

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

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

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

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

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

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

## 2.4. ビルド出力の管理

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

### 2.4.1. ビルド出力

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

出力の種類が **ImageStreamTag** の場合は、イメージが統合された OpenShift Container Platform 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"
```

### 2.4.2. アウトプットイメージの環境変数

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

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

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

### 2.4.3. アウトプットイメージのラベル

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

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

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

**ビルドイメージに適用されるカスタムラベル**

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

## 2.5. 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。

### 2.5.1. Docker ビルド

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

**ヒント**

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

#### 2.5.1.1. Dockerfile FROM イメージの置き換え

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

**手順**

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

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

#### 2.5.1.2. Dockerfile パスの使用

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

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

**手順**

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

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

#### 2.5.1.3. docker 環境変数の使用

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

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

**手順**

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

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

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

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

#### 2.5.1.4. docker ビルド引数の追加

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

**ヒント**

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

**手順**

docker ビルドの引数を設定するには、以下のように **buildArgs** 配列にエントリーを追加します。これは、**BuildConfig** オブジェクトの **dockerStrategy** 定義の中にあります。以下に例を示します。

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

**注記**

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

#### 2.5.1.5. 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
```

### 2.5.2. Source-to-Image ビルド

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

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

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

**手順**

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

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

(1) 増分ビルドをサポートするイメージを指定します。この動作がサポートされているか判断するには、ビルダーイメージのドキュメントを参照してください。

(2) このフラグでは、増分ビルドを試行するかどうかを制御します。ビルダーイメージで増分ビルドがサポートされていない場合は、ビルドは成功しますが、**save-artifacts** スクリプトがないため、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については、S2I 要件について参照してください。

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

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

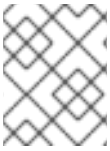
**手順**

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

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

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

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



注記

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

#### 2.5.2.3. Source-to-Image 環境変数

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

##### 2.5.2.3.1. Source-to-Image 環境ファイルの使用

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

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

**手順**

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

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

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

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

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

##### 2.5.2.3.2. 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。

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

#### 2.5.2.5. Source-to-Image によるソースコードからのイメージの作成

Source-to-Image (S2I) は、アプリケーションのソースコードを入力として取り、アセンブルされたアプリケーションを出力として実行する新規イメージを生成するイメージを簡単に作成できるようにするフレームワークです。

再生成可能なコンテナーイメージのビルドに S2I を使用する主な利点として、開発者の使い勝手の良さが挙げられます。ビルダーイメージの作成者は、イメージが最適な S2I パフォーマンスを実現できるように、ビルドプロセスと S2I スクリプトの基本的なコンセプト 2 点を理解する必要があります。

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

ビルドプロセスは、以下の 3 つの要素で設定されており、これら 3 つを組み合わせて最終的なコンテナーイメージが作成されます。

- ソース
- Source-to-Image (S2I) スクリプト
- ビルダーイメージ

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

##### 2.5.2.5.2. 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2.1 S2I スクリプト**

| スクリプト | 説明 |
|---|---|
| **assemble** | **assemble** スクリプトは、ソースからアプリケーションアーティファクトをビルドし、イメージ内の適切なディレクトリーに配置します。このスクリプトが必要です。このスクリプトのワークフローは以下のとおりです。<br>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)。<br>2. 任意の場所に、アプリケーションソースを配置します。<br>3. アプリケーションのアーティファクトをビルドします。<br>4. 実行に適した場所に、アーティファクトをインストールします。 |
| **run** | **run** スクリプトはアプリケーションを実行します。このスクリプトが必要です。 |
| **save-artifacts** | **save-artifacts** スクリプトは、次に続くビルドプロセスを加速できるようにすべての依存関係を収集します。このスクリプトはオプションです。以下に例を示します。<br>- Ruby の場合は、Bundler でインストールされる **gems**<br>- Java の場合は、**.m2** のコンテンツ<br>これらの依存関係は **tar** ファイルに集められ、標準出力としてストリーミングされます。 |
| **usage** | **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。 |

| スクリプト | 説明 |
| --- | --- |
| **test/run** | **test/run** スクリプトでは、イメージが正しく機能しているかどうかを確認するためのプロセスを作成できます。このスクリプトはオプションです。このプロセスの推奨フローは以下のとおりです。<br>1. イメージをビルドします。<br>2. イメージを実行して **usage** スクリプトを検証します。<br>3. **s2i build** を実行して **assemble** スクリプトを検証します。<br>4. 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。<br>5. イメージを実行して、テストアプリケーションが機能していることを確認します。<br>  <br>**注記**<br>**test/run** スクリプトでビルドしたテストアプリケーションを配置するための推奨される場所は、イメージリポジトリーの **test/test-app** ディレクトリーです。 |

### S2I スクリプトの例

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

**assemble スクリプト:**

```
#!/bin/bash

# restore build artifacts
if [ "$(ls /tmp/s2i/artifacts/ 2>/dev/null)" ]; then
    mv /tmp/s2i/artifacts/* $HOME/.
fi

# move the application source
mv /tmp/s2i/src $HOME/src

# build application artifacts
pushd ${HOME}
make all

# install the artifacts
make install
popd
```

**run スクリプト:**

```
#!/bin/bash
```

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

**save-artifacts スクリプト:**

```
#!/bin/bash

pushd ${HOME}
if [ -d deps ]; then
    # all deps contents to tar stream
    tar cf - deps
fi
popd
```

**usage スクリプト:**

```
#!/bin/bash

# inform the user how to use the image
cat <<EOF
This is a S2I sample builder image, to use it, install
https://github.com/openshift/source-to-image
EOF
```

**関連情報**

- S2I イメージ作成のチュートリアル

### 2.5.3. カスタムビルド

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

カスタムビルダーイメージは、RPM またはベースイメージの構築など、ビルドプロセスのロジックに組み込まれるプレーンなコンテナーイメージです。

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

#### 2.5.3.1. カスタムビルドの FROM イメージの使用

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

**手順**

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

```
strategy:
  customStrategy:
    from:
```

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

#### 2.5.3.2. カスタムビルドでのシークレットの使用

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

**手順**

- 各シークレットを特定の場所にマウントするには、**strategy** YAML ファイルの **secretSource** および **mountPath** フィールドを編集します。

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

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

#### 2.5.3.3. カスタムビルドの環境変数の使用

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

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

**手順**

1. ビルド時に使用されるカスタムの HTTP プロキシーを定義します。

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

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

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

#### 2.5.3.4. カスタムビルダーイメージの使用

OpenShift Container Platform のカスタムビルドストラテジーにより、ビルドプロセス全体を対象とす

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

カスタムビルダーイメージは、RPM またはベースのコンテナーイメージの構築など、ビルドプロセスのロジックに組み込まれるプレーンなコンテナーイメージです。

さらに、カスタムビルダーは、単体または統合テストを実行する CI/CD フローなどの拡張ビルドプロセスを実装できます。

#### 2.5.3.4.1. カスタムビルダーイメージ

呼び出し時に、カスタムのビルダーイメージは、ビルドの続行に必要な情報が含まれる以下の環境変数を受け取ります。

**表2.2 カスタムビルダーの環境変数**

| 変数名 | 説明 |
|---|---|
| **BUILD** | **Build** オブジェクト定義のシリアル化された JSON すべて。シリアル化した中で固有の API バージョンを使用する必要がある場合は、ビルド設定のカスタムストラテジーの仕様で、**buildAPIVersion** パラメーターを設定できます。 |
| **SOURCE_REPOSITORY** | ビルドするソースが含まれる Git リポジトリーの URL |
| **SOURCE_URI** | **SOURCE_REPOSITORY** と同じ値を仕様します。どちらでも使用できます。 |
| **SOURCE_CONTEXT_DIR** | ビルド時に使用する Git リポジトリーのサブディレクトリーを指定します。定義された場合にのみ表示されます。 |
| **SOURCE_REF** | ビルドする Git 参照 |
| **ORIGIN_VERSION** | このビルドオブジェクトを作成した OpenShift Container Platform のマスターのバージョン |
| **OUTPUT_REGISTRY** | イメージをプッシュするコンテナーイメージレジストリー |
| **OUTPUT_IMAGE** | ビルドするイメージのコンテナーイメージタグ名 |
| **PUSH_DOCKERCFG_PATH** | **podman push** 操作を実行するためのコンテナーレジストリー認証情報へのパス |

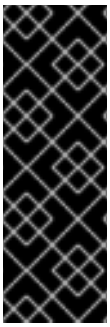
#### 2.5.3.4.2. カスタムビルダーのワークフロー

カスタムビルダーイメージの作成者は、ビルドプロセスを柔軟に定義できますが、ビルダーイメージは、OpenShift Container Platform 内でビルドを実行するために必要な以下の手順に従う必要があります。

1. **Build** オブジェクト定義に、ビルドの入力パラメーターの必要情報をすべて含める。
2. ビルドプロセスを実行する。

3. ビルドでイメージが生成される場合には、ビルドの出力場所が定義されていれば、その場所にプッシュする。他の出力場所には環境変数を使用して渡すことができます。

### 2.5.4. パイプラインビルド



**重要**

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

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

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

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

#### 2.5.4.1. OpenShift Container Platform Pipeline について

**重要**

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

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

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

**OpenShift Container Platform Jenkins 同期プラグイン**

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

- Jenkins での動的なジョブおよび実行の作成。
- イメージストリーム、イメージストリームタグまたは設定マップからのエージェント Pod テンプレートの動的作成。
- 環境変数の挿入。
- OpenShift Container Platform Web コンソールでのパイプラインの可視化。

- Jenkins Git プラグインとの統合。これにより、OpenShift Container Platform ビルドからの Jenkins Git プラグインにコミット情報が渡されます。
- シークレットを Jenkins 認証情報エントリーに同期。

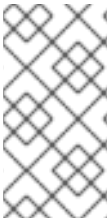
**OpenShift Container Platform Jenkins クライアントプラグイン**

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

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

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

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



**注記**

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

### 2.5.4.2. パイプラインビルド用の Jenkins ファイルの提供

**重要**

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

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

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

**jenkinsfile** は以下のいずれかの方法で指定できます。

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

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

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

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

### 手順

Jenkins ファイルを指定するには、以下のいずれかを実行できます。

- ビルド設定に Jenkins ファイルを埋め込む
- Jenkins ファイルを含む Git リポジトリーへの参照をビルド設定に追加する

### 埋め込み定義

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

### Git リポジトリーへの参照

```
kind: "BuildConfig"
apiVersion: "v1"
metadata:
  name: "sample-pipeline"
spec:
  source:
    git:
      uri: "https://github.com/openshift/ruby-hello-world"
  strategy:
    jenkinsPipelineStrategy:
      jenkinsfilePath: some/repo/dir/filename 1
```

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

#### 2.5.4.3. Pipeline ビルドの環境変数の使用

**重要**

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

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

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

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

**手順**

- ビルド時に使用される環境変数を定義するには、YAML ファイルを編集します。

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

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

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

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

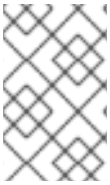
Jenkins ジョブの初回作成後に、パラメーターを Jenkins コンソールからジョブに追加できます。パラメーター名は、ビルド設定の環境変数名とは異なります。上記の Jenkins ジョブ用にビルドを開始すると、これらのパラメーターが使用されます。

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

- **oc start-build** で開始された場合には、ビルド設定の環境変数が対応するジョブインスタンスに設定するパラメーターになります。Jenkins コンソールからパラメーターのデフォルト値に変更を加えても無視されます。ビルド設定値が優先されます。
- **oc start-build -e** で開始する場合、**-e** オプションで指定される環境変数の値が優先されます。
    - ビルド設定に一覧表示されていない環境変数を指定する場合、それらは Jenkins ジョブパラメーター定義として追加されます。

- Jenkins コンソールから環境変数に対応するパラメーターに加える変更は無視されます。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。

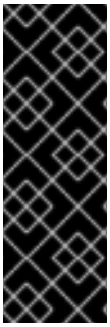
- Jenkins コンソールで Jenkins ジョブを開始した場合には、ジョブのビルドを開始する操作の一環として、Jenkins コンソールを使用してパラメーターの設定を制御できます。



**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。

#### 2.5.4.4. Pipeline ビルドのチュートリアル



**重要**

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

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

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

**手順**

1. Jenkins マスターを作成するには、以下を実行します。

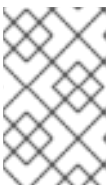
```
$ oc project <project_name>
```

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

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

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

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



**注記**

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

```
kind: "BuildConfig"
apiVersion: "v1"
metadata:
  name: "nodejs-sample-pipeline"
```

```
spec:
  strategy:
    jenkinsPipelineStrategy:
      jenkinsfile: <pipeline content from below>
    type: JenkinsPipeline
```

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

**注記**

この例では、アプリケーションに Git リポジトリーを設定しません。

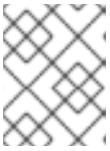
以下の **jenkinsfile** の内容は、OpenShift Container Platform DSL を使用して Groovy で記述されています。ソースリポジトリーに **jenkinsfile** を追加することが推奨される方法ですが、この例では YAML Literal Style を使用して **BuildConfig** にインラインコンテンツを追加しています。

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

```
    }
  }
  stage('create') {
    steps {
      script {
          openshift.withCluster() {
              openshift.withProject() {
                openshift.newApp(templatePath) 7
              }
          }
      }
    }
  }
  stage('build') {
    steps {
      script {
          openshift.withCluster() {
              openshift.withProject() {
                def builds = openshift.selector("bc", templateName).related('builds')
                timeout(5) { 8
                  builds.untilEach(1) {
                    return (it.object().status.phase == "Complete")
                  }
                }
              }
          }
      }
    }
  }
  stage('deploy') {
    steps {
      script {
          openshift.withCluster() {
              openshift.withProject() {
                def rm = openshift.selector("dc", templateName).rollout()
                timeout(5) { 9
                  openshift.selector("dc", templateName).related('pods').untilEach(1) {
                    return (it.object().status.phase == "Running")
                  }
                }
              }
          }
      }
    }
  }
  stage('tag') {
    steps {
      script {
          openshift.withCluster() {
              openshift.withProject() {
                openshift.tag("${templateName}:latest", "${templateName}-staging:latest") 10
              }
          }
      }
    }
  }
```

```
    }
  }
}
```

1. 使用するテンプレートへのパス
2. 作成するテンプレート名
3. このビルドを実行する **node.js** のエージェント Pod をスピンアップします。
4. この Pipeline に 20 分間のタイムアウトを設定します。
5. このテンプレートラベルが指定されたものすべてを削除します。
6. このテンプレートラベルが付いたシークレットをすべて削除します。
7. **templatePath** から新規アプリケーションを作成します。
8. ビルドが完了するまで最大 5 分待機します。
9. デプロイメントが完了するまで最大 5 分待機します。
10. すべてが正常に完了した場合は、**$ {templateName}:latest** イメージに **$ {templateName}-staging:latest** のタグを付けます。ステージング環境向けのパイプラインのビルド設定は、変更する **$ {templateName}-staging:latest** イメージがないかを確認し、このイメージをステージング環境にデプロイします。



**注記**

以前の例は、宣言型のパイプラインスタイルを使用して記述されていますが、以前のスクリプト化されたパイプラインスタイルもサポートされます。

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

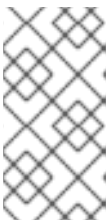
```
$ oc create -f nodejs-sample-pipeline.yaml
```

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

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

5. Pipeline を起動します。

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



**注記**

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

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

- ジョブインスタンスが Jenkins サーバー上で作成される
- パイプラインで必要な場合には、エージェント Pod が起動される
- Pipeline がエージェント Pod で実行されるか、またはエージェントが必要でない場合には master で実行される
  - **template=nodejs-mongodb-example** ラベルの付いた以前に作成されたリソースは削除されます。
  - 新規アプリケーションおよびそれに関連するすべてのリソースは、**nodejs-mongodb-example** テンプレートで作成されます。
  - ビルドは **nodejs-mongodb-example BuildConfig** を使用して起動されます。
    - Pipeline は、ビルドが完了して次のステージをトリガーするまで待機します。
  - デプロイメントは、**nodejs-mongodb-example** のデプロイメント設定を使用して開始されます。
    - パイプラインは、デプロイメントが完了して次のステージをトリガーするまで待機します。
  - ビルドとデプロイに成功すると、**nodejs-mongodb-example:latest** イメージが **nodejs-mongodb-example:stage** としてトリガーされます。
- パイプラインで以前に要求されていた場合には、スレーブ Pod が削除される

**注記**

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

### 2.5.5. Web コンソールを使用したシークレットの追加

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

**手順**

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

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

### 2.5.6. プルおよびプッシュの有効化

プライベートレジストリーへのプルを実行できるようにするには、ビルド設定にプルシークレットを設定し、プッシュします。

**手順**

プライベートレジストリーへのプルを有効にするには、以下を実行します。

- ビルド設定にプルシークレットを設定します。

プッシュを有効にするには、以下を実行します。

- ビルド設定にプッシュシークレットを設定します。

## 2.6. BUILDAH によるカスタムイメージビルド

OpenShift Container Platform 4.7 では、docker ソケットはホストノードに表示されません。これは、カスタムビルドの **mount docker socket** オプションがカスタムビルドイメージ内で使用できる docker ソケットを提供しない可能性がゼロではないことを意味します。

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

**注記**

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

### 2.6.1. 前提条件

- カスタムビルドパーミッションを付与する 方法について確認してください。

### 2.6.2. カスタムビルドアーティファクトの作成

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

**手順**

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

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

```
# /usr/bin/build.sh contains the actual custom build logic that will be run when
# this custom builder image is run.
ENTRYPOINT ["/usr/bin/build.sh"]
```

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

```
FROM registry.access.redhat.com/ubi8/ubi
RUN touch /tmp/build
```

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

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


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


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


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

### 2.6.3. カスタムビルダーイメージのビルド

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

**前提条件**

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

**手順**

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

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

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

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

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

### 2.6.4. カスタムビルダーイメージの使用

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

**前提条件**

- 新規カスタムビルダーイメージに必要なすべての入力を定義します。
- カスタムビルダーイメージをビルドします。

**手順**

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

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

1 プロジェクト名を指定します。

2. **BuildConfig** を作成します。

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

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

```
kind: ImageStream
```

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

4. imagestream を作成します。

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

5. カスタムビルドを実行します。

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

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

## 2.7. 基本的なビルドの実行

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

### 2.7.1. ビルドの開始

現在のプロジェクトに既存のビルド設定から新規ビルドを手動で起動できます。

**手順**

手動でビルドを開始するには、以下のコマンドを入力します。

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

#### 2.7.1.1. ビルドの再実行

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

**手順**

- 手動でビルドを再実行するには、以下のコマンドを入力します。

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

#### 2.7.1.2. ビルドログのストリーミング

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

**手順**

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

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

#### 2.7.1.3. ビルド開始時の環境変数の設定

**--env** フラグを指定して、ビルドの任意の環境変数を設定します。

**手順**

- 必要な環境変数を指定するには、以下のコマンドを実行します。

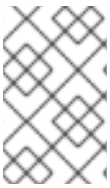
```
$ oc start-build <buildconfig_name> --env=<key>=<value>
```

#### 2.7.1.4. ソースを使用したビルドの開始

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

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

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



**注記**

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

**手順**

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

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

### 2.7.2. ビルドの中止

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

**手順**

- 手動でビルドを取り消すには、以下のコマンドを入力します。

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

#### 2.7.2.1. 複数ビルドのキャンセル

以下の CLI コマンドを使用して複数ビルドを中止できます。

**手順**

- 複数ビルドを手動で取り消すには、以下のコマンドを入力します。

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

#### 2.7.2.2. すべてのビルドのキャンセル

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

**手順**

- すべてのビルドを取り消すには、以下のコマンドを実行します。

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

#### 2.7.2.3. 指定された状態のすべてのビルドのキャンセル

特定の状態にあるビルドをすべて取り消すことができます (例: **new** または **pending**)。この際、他の状態のビルドは無視されます。

**手順**

- 特定の状態のすべてのビルドを取り消すには、以下のコマンドを入力します。

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

### 2.7.3. BuildConfig の削除

以下のコマンドで **BuildConfig** を削除します。

**手順**

- **BuildConfig** を削除するには、以下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>
```

### 2.7.4. ビルドの詳細表示

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

これにより、以下のような情報が表示されます。

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

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

**手順**

- ビルドの詳細を表示するには、以下のコマンドを入力します。

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

### 2.7.5. ビルドログへのアクセス

Web コンソールまたは CLI を使用してビルドログにアクセスできます。

**手順**

- ビルドを直接使用してログをストリーミングするには、以下のコマンドを入力します。

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

#### 2.7.5.1. BuildConfig ログへのアクセス

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

**手順**

- **BuildConfig** の最新ビルドのログをストリーミングするには、以下のコマンドを入力します。

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

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

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

**手順**

- **BuildConfig** の特定バージョンのビルドのログをストリームするには、以下のコマンドを入力します。

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

### 2.7.5.3. ログの冗長性の有効化

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

**注記**

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

ソースビルドで利用できるログレベルは以下のとおりです。

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

**手順**

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

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

1 この値を任意のログレベルに調整します。

## 2.8. ビルドのトリガーおよび変更

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

### 2.8.1. ビルドトリガー

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

- Webhook
- イメージの変更
- 設定の変更

#### 2.8.1.1. Webhook のトリガー

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

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

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

**注記**

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

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

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

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

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

```
- kind: Secret
```

```
apiVersion: v1
metadata:
  name: mysecret
  creationTimestamp:
data:
  WebHookSecretKey: c2VjcmV0dmFsdWUx
```

#### 2.8.1.1.1. GitHub Webhook の使用

GitHub webhook は、リポジトリーの更新時に GitHub からの呼び出しを処理します。トリガーを定義する際に、シークレットを指定する必要があります。このシークレットは、Webhook の設定時に GitHub に指定する URL に追加されます。

GitHub Webhook の定義例:

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

**注記**

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

**oc describe** コマンドは、ペイロード URL を GitHub Webhook URL として返します (Webhook URL の表示を参照)。 ペイロード URL は以下のように設定されます。

**出力例**

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

**前提条件**

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

**手順**

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

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

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

      **出力例**

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

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

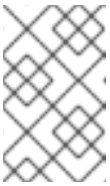
c. GitHub リポジトリーで、**Settings → Webhooks** から **Add Webhook** を選択します。

d. **Payload URL** フィールドに、URL の出力を貼り付けます。

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

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

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



**注記**

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

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

```
$ curl -H "X-GitHub-Event: push" -H "Content-Type: application/json" -k -X POST --data-
binary @payload.json
https://<openshift_api_host:port>/apis/build.openshift.io/v1/namespaces/<namespace>/buildcon
gs/<name>/webhooks/<secret>/github
```

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

**関連情報**

- Gogs

#### 2.8.1.1.2. GitLab Webhook の使用

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

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

**oc describe** コマンドは、ペイロード URL を GitLab Webhook URL として返します。 ペイロード URL は以下のように設定されます。

**出力例**

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

**手順**

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

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

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

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

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

#### 2.8.1.1.3. Bitbucket Webhook の使用

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

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

**oc describe** コマンドは、ペイロード URL を Bitbucket Webhook URL として返します。ペイロード URL は以下のように設定されます。

**出力例**

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

**手順**

1. Bitbucket Webhook を設定するには以下を実行します。
   a. 'BuildConfig' を記述して Webhook URL を取得します。

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

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

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

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

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

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

#### 2.8.1.1.4. Generic Webhook の使用

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

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

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

**手順**

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

   **出力例**

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

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

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

```
$ curl -X POST -k
https://<openshift_api_host:port>/apis/build.openshift.io/v1/namespaces/<namespace>/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。

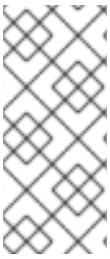
```
git:
  uri: "<url to git repository>"
  ref: "<optional git reference>"
  commit: "<commit hash identifying a specific git commit>"
  author:
    name: "<author name>"
    email: "<author e-mail>"
  committer:
    name: "<committer name>"
    email: "<committer e-mail>"
  message: "<commit message>"
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す。これらの変数が **BuildConfig** の環境変数と競合する場合には、これらの変数が優先されます。デフォルトでは、webhook 経由で渡された環境変数は無視されます。Webhook 定義の **allowEnv** フィールドを **true** に設定して、この動作を有効にします。

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

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

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



**注記**

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

#### 2.8.1.1.5. Webhook URL の表示

以下のコマンドを使用して、ビルド設定に関連する webhook URL を表示できます。コマンドが Webhook URL を表示しない場合、そのビルド設定に定義される Webhook トリガーはありません。

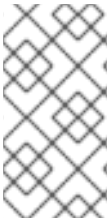
**手順**

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

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

#### 2.8.1.2. イメージ変更トリガーの使用

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



**注記**

v1 コンテナーレジストリー のコンテナーイメージを参照するイメージストリームは、イメージストリームタグが利用できるようになった時点でビルドが 1 度だけトリガーされ、後続のイメージ更新ではトリガーされません。これは、v1 コンテナーレジストリーに一意で識別可能なイメージがないためです。

**手順**

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

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

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

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

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

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

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

3. **ImageStreams** を参照する 1 つまたは複数のトリガーでビルドを定義します。

```
type: "ImageChange" (1)
imageChange: {}
type: "ImageChange" (2)
imageChange:
  from:
    kind: "ImageStreamTag"
    name: "custom-image:latest"
```

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

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

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

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

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

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

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

参照されるイメージストリームで複数の変更を可能にするためにイメージ変更トリガーを一時停止してからビルドを開始できます。また、ビルドがすぐにトリガーされるのを防ぐために、最初に **ImageChangeTrigger** を **BuildConfig** に追加する際に、**paused** 属性を true に設定することもできます。

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

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

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

**関連情報**

- v1 コンテナーレジストリー

#### 2.8.1.3. 設定変更のトリガー

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

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。

#### 2.8.1.3.1. トリガーの手動設定

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

**手順**

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

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

- イメージ変更トリガーを設定するには、以下を使用します。

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

- トリガーを削除するには **--remove** を追加します。

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

**注記**

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

詳細情報は、以下を実行してヘルプドキュメントを参照してください。

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

### 2.8.2. ビルドフック

ビルドフックを使用すると、ビルドプロセスに動作を挿入できます。

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

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

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

ビルドフックは、ビルドが完了とマークされ、イメージがレジストリーに公開される前に、単体テストを実行してイメージを検証するために使用できます。すべてのテストに合格し、テストランナーにより

終了コード **0** が返されると、ビルドは成功とマークされます。テストに失敗すると、ビルドは失敗とマークされます。すべての場合に、ビルドログにはテストランナーの出力が含まれるので、失敗したテストを特定するのに使用できます。

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

#### 2.8.2.1. コミット後のビルドフックの設定

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

**手順**

- シェルスクリプト:

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

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

**注記**

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

- イメージエントリーポイントとしてのコマンド:

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

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

- 引数のあるコマンド:

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

この形式は **command** に引数を追加するのと同じです。

**注記**

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

#### 2.8.2.2. CLI を使用したコミット後のビルドフックの設定

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

**手順**

1. コミット後のビルドフックとしてコマンドを設定します。

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

2. コミット後のビルドフックとしてスクリプトを設定します。

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

## 2.9. 高度なビルドの実行

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

### 2.9.1. ビルドリソースの設定

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

**手順**

リソースの使用を制限する方法は 2 つあります。

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

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

1. **cpu** は CPU のユニットで、 **100m** は 0.1 CPU ユニット (100 * 1e-3) を表します。
2. **memory** はバイト単位です。**256Mi** は 268435456 バイトを表します (256 * 2 ^ 20)。

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

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

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

1 **requests** オブジェクトは、クォータ内のリソース一覧に対応するリソース一覧を含みます。

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

### 2.9.2. 最長期間の設定

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

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

**手順**

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

```
spec:
  completionDeadlineSeconds: 1800
```

**注記**

この設定は、パイプラインストラテジーオプションではサポートされていません。

### 2.9.3. 特定のノードへのビルドの割り当て

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

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

**注記**

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

**手順**

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

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

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

### 2.9.4. チェーンビルド

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

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

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

**注記**

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

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

```
apiVersion: build.openshift.io/v1
kind: BuildConfig
metadata:
  name: artifact-build
spec:
  output:
    to:
      kind: ImageStreamTag
      name: artifact-image:latest
  source:
    git:
      uri: https://github.com/openshift/openshift-jee-sample.git
```

```
      ref: "master"
  strategy:
    sourceStrategy:
      from:
        kind: ImageStreamTag
        name: wildfly:10.1
        namespace: openshift
```

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

```
apiVersion: build.openshift.io/v1
kind: BuildConfig
metadata:
  name: image-build
spec:
  output:
    to:
      kind: ImageStreamTag
      name: image-build:latest
  source:
    dockerfile: |-
      FROM jee-runtime:latest
      COPY ROOT.war /deployments/ROOT.war
    images:
    - from: 1
        kind: ImageStreamTag
        name: artifact-image:latest
      paths: 2
      - sourcePath: /wildfly/standalone/deployments/ROOT.war
        destinationDir: "."
  strategy:
    dockerStrategy:
      from: 3
        kind: ImageStreamTag
        name: jee-runtime:latest
  triggers:
  - imageChange: {}
    type: ImageChange
```

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

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

### 2.9.5. ビルドのプルーニング

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

**手順**

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

```
apiVersion: "v1"
kind: "BuildConfig"
metadata:
  name: "sample-build"
spec:
  successfulBuildsHistoryLimit: 2 (1)
  failedBuildsHistoryLimit: 2 (2)
```

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

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

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

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

**注記**

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

### 2.9.6. ビルド実行ポリシー

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

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

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

## 2.10. ビルドでの RED HAT サブスクリプションの使用

以下のセクションを使用して、OpenShift Container Platform でエンタイトルメントが適用されたビルドを実行します。

### 2.10.1. Red Hat Universal Base Image へのイメージストリームタグの作成

ビルド内で Red Hat サブスクリプションを使用するには、Universal Base Image (UBI) を参照するイメージストリームを作成します。

UBI をクラスター内の **すべてのプロジェクトで** 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。それ以外の場合は、これを **特定のプロジェクトで** 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。

このようにイメージストリームタグを使用すると、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の **registry.redhat.io** 認証情報に基づいて UBI へのアクセスを付与することができます。これは、各開発者が各プロジェクトで **registry.redhat.io** 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ことが必要になる場合よりも便利です。

**手順**

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

```
$ oc tag --source=docker registry.redhat.io/ubi7/ubi:latest ubi:latest -n openshift
```

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

```
$ oc tag --source=docker registry.redhat.io/ubi7/ubi:latest ubi:latest
```

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

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

**前提条件**

サブスクリプションを使用して Red Hat エンタイトルメントにアクセスできる必要があり、エンタイトルメントには別個のパブリックキーおよびプライベートキーファイルがなければなりません。

**ヒント**

Red Hat Enterprise Linux (RHEL) 7 を使用してエンタイトルメントビルドを実行する場合、**yum** コマンドを実行する前に、Dockerfile に次の手順を含める必要があります。

```
RUN rm /etc/rhsm-host
```

**手順**

1. エンタイトルメントを含むシークレットを作成し、パブリックキーとプライベートキーが含まれる別々のファイルがあることを確認します。

```
$  oc create secret generic etc-pki-entitlement --from-file /path/to/entitlement/{ID}.pem \
> --from-file /path/to/entitlement/{ID}-key.pem ...
```

2. シークレットをビルド入力としてビルド設定に追加します。

```
source:
  secrets:
  - secret:
      name: etc-pki-entitlement
    destinationDir: etc-pki-entitlement
```

## 2.10.3. Subscription Manager を使用したビルドの実行

### 2.10.3.1. Subscription Manager を使用した Docker ビルド

Docker ストラテジービルドは Subscription Manager 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、Subscription Manager の設定、および Subscription Manager の認証局は、ビルド入力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/rhel7:latest
USER root
# Copy entitlements
COPY ./etc-pki-entitlement /etc/pki/entitlement
# Copy subscription manager configurations
COPY ./rhsm-conf /etc/rhsm
COPY ./rhsm-ca /etc/rhsm/ca
# Delete /etc/rhsm-host to use entitlements from the build container
RUN rm /etc/rhsm-host && \
    # Initialize /etc/yum.repos.d/redhat.repo
    # See https://access.redhat.com/solutions/1443553
    yum repolist --disablerepo=* && \
    subscription-manager repos --enable <enabled-repo> && \
    yum -y update && \
    yum -y install <rpms> && \
    # Remove entitlements and Subscription Manager configs
    rm -rf /etc/pki/entitlement && \
    rm -rf /etc/rhsm
# OpenShift requires images to run as non-root by default
USER 1001
ENTRYPOINT ["/bin/bash"]
```

## 2.10.4. Red Hat Satellite サブスクリプションを使用したビルドの実行

### 2.10.4.1. Red Hat Satellite 設定のビルドへの追加

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

**前提条件**

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

  **サンプルリポジトリーの設定**

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

**手順**

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

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

2. Satellite リポジトリー設定を **BuildConfig** に追加します。

```
source:
  configMaps:
  - configMap:
      name: yum-repos-d
    destinationDir: yum.repos.d
```

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

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

**前提条件**

- エンタイトルメントキーと Satellite リポジトリー設定がビルド入力として追加される必要があります。

**手順**

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

```
FROM registry.redhat.io/rhel7:latest
USER root
# Copy entitlements
COPY ./etc-pki-entitlement /etc/pki/entitlement
# Copy repository configuration
COPY ./yum.repos.d /etc/yum.repos.d
# Delete /etc/rhsm-host to use entitlements from the build container
RUN sed -i".org" -e "s#^enabled=1#enabled=0#g" /etc/yum/pluginconf.d/subscription-manager.conf
(1)
#RUN cat /etc/yum/pluginconf.d/subscription-manager.conf
RUN yum clean all
#RUN yum-config-manager
```

```
RUN rm /etc/rhsm-host && \
    # yum repository info provided by Satellite
    yum -y update && \
    yum -y install <rpms> && \
    # Remove entitlements
    rm -rf /etc/pki/entitlement
# OpenShift requires images to run as non-root by default
USER 1001
ENTRYPOINT ["/bin/bash"]
```

1 **enabled=1** を使用して Satellite 設定をビルドに追加できない場合は、 **RUN sed -i".org" -e "s#^enabled=1#enabled=0#g" /etc/yum/pluginconf.d/subscription-manager.conf** を Dockerfile に追加します。

### 2.10.5. 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

## 2.11. ストラテジーによるビルドのセキュリティー保護

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

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

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

**表2.3 ビルドストラテジーのサブリソースおよびロール**

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

| ストラテジー | サブリソース | ロール |
| --- | --- | --- |
| JenkinsPipeline | ビルド/jenkinspipeline | system:build-strategy-jenkinspipeline |

### 2.11.1. ビルドストラテジーへのアクセスのグローバルな無効化

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

**手順**

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

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

**出力例**

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

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

2. ロールを削除します。

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

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

```
$ oc edit clusterrole admin
```

```
$ oc edit clusterrole edit
```

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

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

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

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

### 2.11.2. ユーザーへのビルドストラテジーのグルーバルな制限

一連の特定ユーザーのみが特定のストラテジーでビルドを作成できます。

**前提条件**

- ビルドストラテジーへのグローバルアクセスを無効にします。

**手順**

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

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

**警告**

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

### 2.11.3. プロジェクト内でのユーザーへのビルドストラテジーの制限

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

**前提条件**

- ビルドストラテジーへのグローバルアクセスを無効にします。

**手順**

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

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

## 2.12. ビルド設定リソース

以下の手順でビルドを設定します。

### 2.12.1. ビルドコントローラー設定パラメーター

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

| パラメーター | 説明 |
|---|---|
| **Build** | ビルドの処理方法についてのクラスター全体の情報を保持します。正規名および唯一の有効な名前となるのは **cluster** です。<br><br>**spec**: ビルドコントローラー設定のユーザーが設定できる値を保持します。 |

| パラメーター | 説明 |
| --- | --- |
| **buildDefaults** | ビルドのデフォルト情報を制御します。<br><br>**defaultProxy**: イメージのプルまたはプッシュ、およびソースのダウンロードを含む、ビルド操作のデフォルトのプロキシー設定が含まれます。<br><br>**BuildConfig** ストラテジーに **HTTP_PROXY**、**HTTPS_PROXY**、および **NO_PROXY** 環境変数を設定することで、値を上書きできます。<br><br>**gitProxy**: Git 操作のプロキシー設定のみが含まれます。設定されている場合、これは **git clone** などの Git コマンドのプロキシー設定を上書きします。<br><br>ここで設定されていない値は DefaultProxy から継承されます。<br><br>**env**: 指定される変数がビルドに存在しない場合にビルドに適用される一連のデフォルト環境変数。<br><br>**imageLabels**: 結果として生成されるイメージに適用されるラベルの一覧。**BuildConfig** に同じ名前のラベルを指定することでデフォルトのラベルを上書きできます。<br><br>**resources**: ビルドを実行するためのリソース要件を定義します。 |
| **ImageLabel** | **name**: ラベルの名前を定義します。ゼロ以外の長さを持つ必要があります。 |
| **buildOverrides** | ビルドの上書き設定を制御します。<br><br>**imageLabels**: 結果として生成されるイメージに適用されるラベルの一覧。表にあるものと同じ名前のラベルを **BuildConfig** に指定する場合、ラベルは上書きされます。<br><br>**nodeSelector**: セレクター。 ビルド Pod がノードに適合させるには True である必要があります。<br><br>**tolerations**: ビルド Pod に設定された既存の容認を上書きする容認の一覧。 |
| **BuildList** | **items**: 標準オブジェクトのメタデータ。 |

### 2.12.2. ビルド設定の設定

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

**手順**

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

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

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

```
apiVersion: config.openshift.io/v1
kind: Build 1
```

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

1. **Build**: ビルドの処理方法についてのクラスター全体の情報を保持します。正規名および唯一の有効な名前となるのは **cluster** です。
2. **buildDefaults**: ビルドのデフォルト情報を制御します。
3. **defaultProxy**: イメージのプルまたはプッシュ、およびソースのダウンロードを含む、ビルド操作のデフォルトのプロキシー設定が含まれます。
4. **env**: 指定される変数がビルドに存在しない場合にビルドに適用される一連のデフォルト環境変数。
5. **gitProxy**: Git 操作のプロキシー設定のみが含まれます。設定されている場合、これは **git**

clone などの Git コマンドのプロキシー設定を上書きします。

6. **imageLabels**: 結果として生成されるイメージに適用されるラベルの一覧。**BuildConfig** に同じ名前のラベルを指定することでデフォルトのラベルを上書きできます。
7. **resources**: ビルドを実行するためのリソース要件を定義します。
8. **buildOverrides**: ビルドの上書き設定を制御します。
9. **imageLabels**: 結果として生成されるイメージに適用されるラベルの一覧。表にあるものと同じ名前のラベルを **BuildConfig** に指定する場合、ラベルは上書きされます。
10. **nodeSelector**: セレクター。 ビルド Pod がノードに適合させるには True である必要があります。
11. **tolerations**: ビルド Pod に設定された既存の容認を上書きする容認の一覧。

## 2.13. ビルドのトラブルシューティング

ビルドの問題をトラブルシューティングするために、以下を使用します。

### 2.13.1. リソースへのアクセスのための拒否の解決

リソースへのアクセス要求が拒否される場合:

**問題**

ビルドが以下のエラーで失敗します。

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

**解決策**

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

```
$ oc describe quota
```

### 2.13.2. サービス証明書の生成に失敗

リソースへのアクセス要求が拒否される場合:

**問題**

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

**出力例**

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

**解決策**

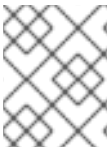
証明書を生成したサービスがすでに存在しないか、またはサービスに異なる **serviceUID** があります。古いシークレットを削除し、サービスのアノテーション (**service.beta.openshift.io/serving-**

**cert-generation-error** および **service.beta.openshift.io/serving-cert-generation-error-num**) をクリアして証明書の再生成を強制的に実行する必要があります。

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

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

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



**注記**

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

## 2.14. ビルドの信頼される認証局の追加設定

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

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

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

### 2.14.1. クラスターへの認証局の追加

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

**前提条件**

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

**手順**

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

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

2. クラスターイメージの設定を更新します。

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

### 2.14.2. 関連情報

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

# 第3章 パイプライン

## 3.1. RED HAT OPENSHIFT PIPELINES リリースノート

Red Hat OpenShift Pipelines は、以下を提供する Tekton プロジェクトをベースとするクラウドネイティブの CI/CD エクスペリエンスです。

- 標準の Kubernetes ネイティブパイプライン定義 (CRD)
- CI サーバー管理のオーバーヘッドのないサーバーレスのパイプライン。
- S2I、Buildah、JIB、Kaniko などの Kubernetes ツールを使用してイメージをビルドするための拡張性。
- Kubernetes ディストリビューションでの移植性。
- パイプラインと対話するための強力な CLI。
- OpenShift Container Platform Web コンソールの **Developer** パースペクティブと統合されたユーザーエクスペリエンス。

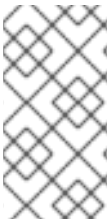
Red Hat OpenShift Pipelines の概要については、OpenShift Pipelines について を参照してください。

### 3.1.1. 多様性を受け入れるオープンソースの強化

Red Hat では、コード、ドキュメント、Web プロパティーにおける配慮に欠ける用語の置き換えに取り組んでいます。まずは、マスター (master)、スレーブ (slave)、ブラックリスト (blacklist)、ホワイトリスト (whitelist) の 4 つの用語の置き換えから始めます。この取り組みは膨大な作業を要するため、今後の複数のリリースで段階的に用語の置き換えを実施して参ります。詳細は、弊社の CTO、Chris Wright のメッセージ を参照してください。

### 3.1.2. Red Hat OpenShift Pipelines General Availability (GA) 1.4 のリリースノート

Red Hat OpenShift Pipelines General Availability (GA) 1.4 が OpenShift Container Platform 4.7 で利用可能になりました。



**注記**

stable および preview Operator チャネルのほかに、Red Hat OpenShift Pipelines Operator 1.4.0 には ocp-4.6、ocp-4.5、および ocp-4.4 の非推奨チャネルが同梱されます。これらの非推奨チャネルおよびそれらのサポートは、Red Hat OpenShift Pipelines の以下のリリースで削除されます。

#### 3.1.2.1. 互換性およびサポート表

現在、今回のリリースに含まれる機能にはテクノロジープレビューのものがあります。これらの実験的機能は、実稼働環境での使用を目的としていません。

テクノロジープレビュー機能のサポート範囲

以下の表では、機能は以下のステータスでマークされています。

- **TP**: **テクノロジープレビュー**

- GA: 一般公開機能

これらの機能に関しては、Red Hat カスタマーポータルの以下のサポート範囲を参照してください。

**表3.1 互換性およびサポート表**

| 機能 | バージョン | サポートステータス |
|---|---|---|
| パイプライン | 0.22 | GA |
| CLI | 0.17 | GA |
| カタログ | 0.22 | GA |
| トリガー | 0.12 | TP |
| パイプラインリソース | - | TP |

質問やフィードバックについては、製品チームに pipelines-interest@redhat.com 宛のメールを送信してください。

### 3.1.2.2. 新機能

以下のセクションでは、修正および安定性の面での改善点に加え、OpenShift Pipelines 1.4 の主な新機能について説明します。

- カスタムタスクには、以下の機能強化が含まれます。
  - パイプラインの結果として、カスタムタスクで生成される結果を参照できるようになりました。
  - カスタムタスクはワークスペース、サービスアカウント、および Pod テンプレートを使用して、より複雑なカスタムタスクをビルドできるようになりました。
- **finally** タスクには、以下の機能強化が含まれます。
  - **when** 式は **最後** のタスクでサポートされます。これにより、効率的に保護された実行が可能になり、タスクの再利用性が向上します。
  - **finally** タスクは、同じパイプライン内のタスクの結果を使用するように設定できます。

**注記**

**when** 式および **finally** タスクのサポートは OpenShift Container Platform 4.7 Web コンソールでは利用できません。

- **dockercfg** または **dockerconfigjson** タイプの複数のシークレットのサポートがランタイム時に認証用に追加されました。
- **git-clone** タスクでスパースチェックをサポートする機能が追加されました。これにより、ローカルコピーとしてリポジトリーのサブセットのみをクローンすることができ、これはクローン作成したリポジトリーのサイズを制限するのに便利です。

- 実際に起動せずに、パイプライン実行を保留中の状態で作成できます。負荷が大きいクラスターでは、これにより、Operator はパイプライン実行の開始時間を制御することができます。
- コントローラー用に **SYSTEM_NAMESPACE** 環境変数を手動で設定していることを確認します。これは以前はデフォルトで設定されていました。
- root 以外のユーザーがパイプラインのビルドベースイメージに追加され、**git-init** がリポジトリーのクローンを root 以外のユーザーとして作成できるようになりました。
- パイプライン実行の開始前に解決されたリソース間で依存関係を検証するサポートが追加されています。パイプラインのすべての結果変数は有効でなければならず、パイプラインからのオプションのワークスペースは、パイプライン実行の開始に使用することが予想されているタスクにのみ渡すことができます。
- コントローラーおよび Webhook は root 以外のグループとして実行され、それらの必要以上の機能は削除され、よりセキュアになりました。
- **tkn pr logs** コマンドを使用して、再試行されたタスク実行のログストリームを表示できます。
- **tkn tr delete** コマンドで **--clustertask** オプションを使用して、特定のクラスタータスクに関連付けられたすべてのタスク実行を削除できます。
- **EventListener** リソースでの Knative サービスのサポートは、新規の **customResource** フィールドを導入して追加されます。
- イベントペイロードが JSON 形式を使用しない場合にエラーメッセージが表示されます。
- GitLab、BitBucket、GitHub などのソース制御インターセプターは、新規の **InterceptorRequest** または **InterceptorResponse** を使用できるようになりました。
- 新しい CEL 関数の **marshalJSON** が実装され、JSON オブジェクトまたは配列を文字列にエンコードできます。
- CEL およびソース制御コアインターセプターを提供する HTTP ハンドラーが追加されました。これは、**tekton-pipelines** namespace にデプロイされる単一の HTTP サーバーに 4 つのコアインターセプターをパッケージ化します。**EventListener** オブジェクトは、HTTP サーバー経由でイベントをインターセプターに転送します。それぞれのインターセプターは異なるパスで利用できます。たとえば、CEL インターセプターは **/cel** パスで利用できます。
- **pipelines-scc** SCC (Security Context Constraint) は、パイプラインのデフォルト **pipeline** サービスアカウントで使用されます。この新規サービスアカウントは **anyuid** と似ていますが、OpenShift Container Platform 4.7 の SCC について YAML に定義されるように若干の違いがあります。

```
fsGroup:
  type: MustRunAs
```

#### 3.1.2.3. 非推奨の機能

- パイプラインリソースストレージの **build-gcs** サブタイプ、および **gcs-fetcher** イメージは、サポートされていません。
- クラスタータスクの **taskRun** フィールドで、**tekton.dev/task** ラベルが削除されます。
- Webhook の場合、フィールド **admissionReviewVersions** に対応する値 **v1beta1** は削除されます。

- ビルドおよびデプロイ用の **creds-init** ヘルパーイメージが削除されます。
- トリガー仕様およびバインディングでは、**template.ref** が優先されるため、非推奨フィールドの **template.name** が削除されます。**ref** フィールドを使用するには、**eventListener** のすべての定義を更新する必要があります。

**注記**

**template.name** フィールドが利用できないため、Pipelines 1.3.x 以前のバージョンから Pipelines 1.4.0 へのアップグレードにより、イベントリスナーが破損します。このような場合には、Pipelines 1.4.1 を使用して、復元された **template.name** フィールドを利用します。

- **EventListener** カスタムリソース/オブジェクトの場合、**Resource** が優先されるために、**PodTemplate** および **ServiceType** フィールドは非推奨になりました。
- 非推奨の仕様スタイルの埋め込みバインディングは削除されています。
- **spec** フィールドは **triggerSpecBinding** から削除されています。
- イベント ID 表現は、5 文字のランダムな文字列から UUID に変更されています。

#### 3.1.2.4. 既知の問題

- **Developer** パースペクティブでは、Pipeline メトリクスおよびトリガー機能は OpenShift Container Platform 4.7.6 以降のバージョンでのみ利用できます。
- IBM Power Systems、IBM Z、および LinuxONE では、**tkn hub** コマンドはサポートされません。
- IBM Power Systems (ppc64le)、IBM Z、および LinuxONE (s390x) クラスターで Maven および Jib Maven クラスタータスクを実行する場合、**MAVEN_IMAGE** パラメーターの値を **maven:3.6.3-adoptopenjdk-11** に設定します。
- トリガーは、トリガーバインディングに以下の設定がある場合は、JSON 形式の正しくない処理によって生じるエラーを出力します。

```
params:
  - name: github_json
    value: $(body)
```

この問題を解決するには、以下を実行します。

  - トリガー v0.11.0 以降を使用している場合、**marshalJSON** 関数を使用して JSON オブジェクトまたは配列を取得し、そのオブジェクトまたは配列の JSON エンコーディングを文字列として返します。
  - 古いバージョンのトリガーを使用している場合は、以下のアノテーションをトリガーテンプレートに追加します。

```
annotations:
  triggers.tekton.dev/old-escape-quotes: "true"
```

- Pipelines 1.3.x から 1.4.x にアップグレードする際に、ルートを再作成する必要があります。

#### 3.1.2.5. 修正された問題

- 以前のバージョンでは、**tekton.dev/task** ラベルがクラスタータスクのタスク実行から削除され、**tekton.dev/clusterTask ラベル** が導入されました。この変更により生じる問題は 、**clustertask describe** および **delete** コマンドを修正して解決されています。さらに、タスクの **lastrun** 機能は変更され、古いバージョンのパイプラインでタスクとクラスタータスクの両方のタスク実行に適用される **tekton.dev/task** ラベルの問題を修正できるになりました。

- 対話的な **tkn pipeline start pipelinename** を実行する場合、**PipelineResource** が対話的に作成されます。**tkn p start** コマンドは、リソースのステータスが **nil** ではない場合にリソースのステータスを出力します。

- 以前のバージョンでは、**tekton.dev/task=name** ラベルは、クラスタータスクから作成されるタスク実行から削除されました。今回の修正により、**--last** フラグの指定される **tkn clustertask start** コマンドが変更され、作成されたタスク実行で **tekton.dev/task=name** ラベルの有無がチェックされるようになりました。

- タスクがインラインのタスク仕様を使用する場合、対応するタスク実行は **tkn pipeline describe** コマンドの実行時にパイプラインに組み込まれ、タスク名は埋め込まれた状態で返されます。

- **tkn version** コマンドは、設定された **kubeConfiguration namespace** やクラスターへのアクセスなしに、インストールされた Tekton CLI ツールのバージョンを表示するように修正されています。

- 引数が予期せずに使用されるか、または複数の引数が使用される場合、**tkn completion** コマンドでエラーが発生します。

- 以前のバージョンでは、パイプライン仕様でネスト化された **finally** タスクのあるパイプライン実行は、**v1alpha1** バージョンに変換され、**v1beta1** バージョンに戻されると、それらの **finally** タスクを失うことがあります。変換中に発生するこのエラーは修正され、潜在的データ損失を防ぐことができます。**finally** タスクがパイプライン仕様でネスト化されたパイプライン実行はシリアライズされ、アルファバージョンに保存されてデシリアライズは後に実行されるようになりました。

- 以前のバージョンでは、サービスアカウントで **secrets** フィールドに **{}** があると、Pod の生成でエラーが発生しました。空のシークレット名を持つ GET 要求がエラーがリソース名が空ではないことを示すエラーを返すため、タスク実行は **CouldntGetTask** で失敗しました。この問題は、**kubeclient** GET 要求で空のシークレット名を使用しないことで解決されています。

- **v1beta1** API バージョンのあるパイプラインは、**finally** タスクを失うことなく、**v1alpha1** バージョンと共に要求できるようになりました。返される **v1alpha1** バージョンを適用すると、リソースが **v1beta1** として保存され、**finally** セクションがその元の状態に戻ります。

- 以前のバージョンでは、コントローラーの **selfLink** フィールドが設定されていないと、Kubernetes v1.20 クラスターでエラーが発生しました。一時的な修正として、**CloudEvent** ソースフィールドは、自動設定される **selfLink** フィールドの値なしに現在のソース URI に一致する値に設定されます。

- 以前のバージョンでは、**gcr.io** などのドットの付いたシークレット名により、タスク実行の作成が失敗しました。これは、シークレット名がボリュームマウント名の一部として内部で使用されるために生じました。ボリュームマウント名は RFC1123 DNS ラベルに準拠し、名前の一部として使用されるドットを許可しません。この問題は、ドットをダッシュに置き換えることで解決し、これにより名前の読み取りが可能になりました。

- コンテキスト変数は、**finally** タスクで検証されるようになりました。

- 以前のバージョンでは、タスク実行リコンサイラーが渡され、作成した Pod の名前を含む直前のステータス更新を持たないタスク実行があると、タスク実行リコンサイラーはタスク実行に関連付けられた Pod を一覧表示しました。タスク実行リコンサイラーは、Pod を検索するために、Pod に伝播されるタスク実行のラベルを使用しました。タスク実行の実行中にこれらのラベルを変更すると、コードが既存の Pod を見つけることができませんでした。その結果、重複した Pod が作成されました。この問題は、Pod の検索時に **tekton.dev/taskRun** の Tekton で制御されるラベルのみを使用するようにタスク実行リコンサイラーを変更することで修正されています。

- 以前のバージョンでは、パイプラインがオプションのワークスペースを受け入れ、これをパイプラインタスクに渡すと、パイプライン実行リコンサイラーは、ワークスペースが提供されておらず、欠落しているワークスペースのバインディングがオプションのワークスペースについて有効な場合でも、エラーを出して停止しました。この問題は、オプションのワークスペースが指定されていない場合でも、パイプライン実行リコンサイラーがタスク実行の作成に失敗しないようにすることで修正されています。

- ステップのステータスの並び順は、ステップコンテナーの順序と一致します。

- 以前のバージョンでは、Pod で **CreateContainerConfigError** の理由が出されると、タスク実行のステータスは **unknown** に設定されました。これは、タスクおよびパイプラインが Pod がタイムアウトするまで実行されることを意味しました。この問題は、Pod で **CreateContainerConfigError** の理由が出される際にタスクを失敗 (failed) として設定できるようにタスク実行ステータスを **false** に設定することで解決されています。

- 以前のバージョンでは、パイプライン実行の完了後に、パイプラインの結果は最初の調整で解決されました。これにより解決が失敗し、パイプライン実行の **Succeeded** 状態が上書きされる可能性がありました。その結果、最終のステータス情報が失われ、パイプライン実行の状態を監視するすべてのサービスに混乱を生じさせる可能性がありました。この問題は、パイプライン実行が **Succeeded** または **True** 状態になる際に、パイプラインの結果の解決を調整の最後に移行することにで解決されました。

- 実行ステータス変数が検証されるようになりました。これにより、実行ステータスにアクセスするためのコンテキスト変数の検証中に、タスク結果が検証されることを防ぐことができます。

- 以前のバージョンでは、無効な変数を含むパイプラインの結果は、変数のリテラル式はそのままの状態でパイプライン実行に追加されます。そのため、結果が正しく設定されているかどうかを評価することは容易ではありませんでした。この問題は、失敗したタスク実行を参照するパイプライン実行結果でフィルターリングすることで解決されています。無効な変数を含むパイプラインの結果は、パイプライン実行によって出されなくなりました。

- **tkn eventlistener describe** コマンドは、テンプレートなしでクラッシュを回避できるように修正されています。また、トリガーの参照に関する情報も表示します。

- **template.name** が利用できないため、Pipelines 1.3.x 以前のバージョンから Pipelines 1.4.0 へのアップグレードにより、イベントリスナーが破損します。Pipelines 1.4.1 では、トリガーでイベントリスナーが破損しないように、**template.name** が復元されています。

- Pipelines 1.4.1 では、**ConsoleQuickStart** カスタムリソースが OpenShift Container Platform 4.7 の機能および動作に合わせて更新されました。

## 3.1.3. Red Hat OpenShift Pipelines テクノロジープレビュー 1.3 のリリースノート

### 3.1.3.1. 新機能

Red Hat OpenShift Pipelines テクノロジープレビュー (TP) 1.3 が OpenShift Container Platform 4.7 で利用可能になりました。Red Hat OpenShift Pipelines TP 1.3 が以下をサポートするように更新されています。

- Tekton Pipelines 0.19.0
- Tekton **tkn** CLI 0.15.0
- Tekton Triggers 0.10.2
- Tekton Catalog 0.19.0 をベースとするクラスタータスク
- OpenShift Container Platform 4.7 での IBM Power Systems
- OpenShift Container Platform 4.7 での IBM Z および LinuxONE

以下のセクションでは、修正および安定性の面での改善点に加え、OpenShift Pipelines 1.3 の主な新機能について説明します。

##### 3.1.3.1.1. パイプライン

- S2I や Buildah タスクなどのイメージをビルドするタスクが、イメージの SHA を含むビルドされたイメージの URL を生成するようになりました。
- **Condition** カスタムリソース定義 (CRD) が非推奨となっているため、カスタムタスクを参照するパイプラインタスクの条件は許可されません。
- **spec.steps[].imagePullPolicy** および **spec.sidecar[].imagePullPolicy** フィールドの **Task** CRD に変数の拡張が追加されました。
- **disable-creds-init** feature-flag を **true** に設定すると、Tekton のビルトイン認証情報メカニズムを無効にすることができます。
- 解決済みの When 式は、**PipelineRun** 設定の **Status** フィールドの **Skipped Tasks** および **Task Runs** セクションに一覧表示されるようになりました。
- **git init** コマンドが、再帰的なサブモジュールのクローンを作成できるようになりました。
- **Task** CR の作成者は、 **Task** 仕様のステップのタイムアウトを指定できるようになりました。
- エントリーポイントイメージを **distroless/static:nonroot** イメージにベースとして作成し、ベースイメージに存在する **cp** コマンドを使用せずに、これを宛先にコピーするモードを許可できるようになりました。
- Git SSH シークレットの既知のホストの省略を許可しないように、設定フラグ **require-git-ssh-secret-known-hosts** を使用できるようになりました。フラグ値が **true** に設定されている場合には、Git SSH シークレットに **known_host** フィールドを含める必要があります。フラグのデフォルト値は **false** です。
- オプションのワークスペースの概念が導入されました。タスクまたはパイプラインはワークスペースオプションを宣言し、その存在に基づいて動作を条件的に変更する可能性があります。タスク実行またはパイプライン実行により、そのワークスペースが省略され、タスクまたはパイプラインの動作が変更される可能性があります。デフォルトのタスク実行ワークスペースは、省略されたオプションのワークスペースの代わりに追加されることはありません。

- Tekton の認証情報の初期化により、SSH 以外の URL で使用する SSH 認証情報が検出されるほか、Git パイプラインリソースでは SSH URL で使用する http 認証情報が検出され、Step コンテナーで警告がログに記録されるようになりました。

- タスク実行コントローラーは、Pod テンプレートで指定されたアフィニティーがアフィニティーアシスタントによって上書きされる場合に警告イベントを生成します。

- タスク実行リコンサイラーは、タスク実行が完了すると生成されるクラウドイベントのメトリクスを記録するようになりました。これには再試行が含まれます。

#### 3.1.3.1.2. Pipelines CLI

- **--no-headers flag** のサポートが、次のコマンドに追加されました: **tkn condition list**、**tkn triggerbinding list**、**tkn eventlistener list**、**tkn clustertask list**、**tkn clustertriggerbinding list**

- 併用した場合、**--last** または **--use** オプションは、**--prefix-name** および **--timeout** オプションを上書きします。

- **tkn eventlistener logs** コマンドが追加され、**EventListener** ログが表示されるようになりました。

- **tekton hub** コマンドは **tkn** CLI に統合されるようになりました。

- **--nocolour** オプションは **--no-color** に変更されました。

- **--all-namespaces** フラグは、次のコマンドに追加されました: **tkn triggertemplate list**、**tkn condition list**、**tkn triggerbinding list**、**tkn eventlistener list**

#### 3.1.3.1.3. トリガー

- **EventListener** テンプレートでリソース情報を指定できるようになりました。

- すべてのトリガーリソースの **get** 動詞に加えて、**EventListener** サービスアカウントに **list** および **watch** 動詞が設定されることが必須になりました。これにより、**Listers** を使用して **EventListener**、 **Trigger**、**TriggerBinding**、**TriggerTemplate**、および **ClusterTriggerBinding** リソースからデータを取得することができます。この機能を使用して、複数のインフォーマーを指定するのではなく **Sink** オブジェクトを作成し、API サーバーを直接呼び出すことができます。

- イミュータブルな入力イベント本体をサポートする新たな **Interceptor** インターフェイスが追加されました。インターセプターはデータまたはフィールドを新しい **extensions** フィールドに追加できるようになり、入力本体を変更できなくなったことでイミュータブルとなりました。CEL インターセプターはこの新たな **Interceptor** インターフェイスを使用します。

- **namespaceSelector** フィールドは **EventListener** リソースに追加されます。これを使用して、**EventListener** リソースがイベント処理用に **Trigger** オブジェクトを取得できる namespace を指定します。**namespaceSelector** フィールドを使用するには、**EventListener** のサービスアカウントにクラスターロールが必要です。

- トリガー **EventListener** リソースは、**eventlistener** Pod へのエンドツーエンドのセキュアな接続をサポートするようになりました。

- **"** を **\"** に置き換えることで、**TriggerTemplates** リソースのエスケープパラメーター動作が削除されました。

- Kubernetes リソースをサポートする新規 **resources** フィールドは、**EventListener** 仕様の一部として導入されます。
- ASCII 文字列の大文字と小文字へのサポートが含まれる CEL インターセプターの新機能が追加されました。
- **TriggerBinding** リソースは、トリガーの **name** および **value** フィールドを使用するか、またはイベントリスナーを使用して埋め込むことができます。
- **PodSecurityPolicy** 設定は、制限された環境で実行されるように更新されます。これにより、コンテナーは root 以外のユーザーとして実行する必要があります。さらに、Pod セキュリティーポリシーを使用するためのロールベースのアクセス制御は、クラスタースコープから namespace スコープに移行されます。これにより、トリガーは namespace に関連しない他の Pod セキュリティーポリシーを使用することができません。
- 埋め込みトリガーテンプレートのサポートが追加されました。**name** フィールドを使用して埋め込みテンプレートを参照するか、または **spec** フィールド内にテンプレートを埋め込むことができます。

#### 3.1.3.2. 非推奨の機能

- **PipelineResources** CRD を使用する Pipeline テンプレートは非推奨となり、今後のリリースで削除されます。
- **template.ref** フィールドが優先されるため、**template.name** フィールドは非推奨となり、今後のリリースで削除されます。
- **--check** コマンドの短縮形である **-c** が削除されました。さらに、グローバル **tkn** フラグが **version** コマンドに追加されます。

#### 3.1.3.3. 既知の問題

- CEL オーバーレイは、受信イベント本体を変更する代わりに、フィールドを新しい最上位の **extensions** 関数に追加します。**TriggerBinding** リソースは、**$(extensions.<key>)** 構文を使用して、この新しい **extensions** 関数内の値にアクセスできます。**$(body.<overlay-key>)** の代わりに **$(extensions.<key>)** 構文を使用するようにバインディングを更新します。
- **"** を **\"** に置き換えることで、エスケープパラメーター動作が削除されました。古いエスケープパラメーターの動作を保持する必要がある場合は、**tekton.dev/old-escape-quotes: true"** アノテーションを **TriggerTemplate** 仕様に追加します。
- **TriggerBinding** リソースは、トリガーまたはイベントリスナー内の **name** および **value** フィールドを使用して組み込みことができます。ただし、単一のバインディングに **name** および **ref** フィールドの両方を指定することはできません。**ref** フィールドを使用して **TriggerBinding** リソースおよび埋め込みバインディングの **name** フィールドを参照します。
- インターセプターは、**EventListener** リソースの namespace 外で **secret** の参照を試行することはできません。シークレットを `EventListener` の namespace に含める必要があります。
- Trigger 0.9.0 以降では、本体またはヘッダーベースの **TriggerBinding** パラメーターが見つからないか、またはイベントペイロードで形式が正しくない場合に、エラーを表示する代わりにデフォルト値が使用されます。
- JSON アノテーションを修正するには、Tekton および Pipelines 0.16.x を使用して **WhenExpression** オブジェクトで作成されたタスクおよびパイプラインを再適用する必要があります。

- パイプラインがオプションのワークスペースを受け入れ、これをタスクに付与すると、ワークスペースが指定されていない場合はパイプライン実行が停止します。

- 非接続環境で Buildah クラスタータスクを使用するには、Dockerfile が内部イメージストリームをベースイメージとして使用していることを確認してから、これを S2I クラスタータスクと同じ方法で使用します。

#### 3.1.3.4. 修正された問題

- CEL インターセプターによって追加された拡張機能は、イベント本体内に **Extensions** フィールドを追加して Webhook インターセプターに渡されます。

- ログリーダーのアクティビティータイムアウトは、**LogOptions** フィールドを使用して設定できるようになりました。ただし、10 秒のタイムアウトのデフォルト動作は保持されます。

- **log** コマンドは、タスク実行またはパイプライン実行が完了したときに **--follow** フラグを無視し、ライブログではなく利用可能なログを読み取ります。

- 以下の Tekton リソースへの参照:
**EventListener**、**TriggerBinding**、**ClusterTriggerBinding**、**Condition**、および **TriggerTemplate** は、**tkn** コマンドのすべてのユーザーに表示されるメッセージで標準化され、一貫性を保つようになりました。

- 以前は、**--use-taskrun <canceled-task-run-name>**、**--use-pipelinerun <canceled-pipeline-run-name>** または **--last** フラグを使用してキャンセルされたタスク実行またはパイプライン実行を開始した場合、新規の実行はキャンセルされました。このバグは修正されています。

- **tkn pr desc** コマンドが強化され、パイプラインが各種の状態で実行された場合に失敗しなくなりました。

- **--task** オプションで **tkn tr delete** コマンドを使用してタスク実行を削除し、クラスタータスクが同じ名前で存在する場合、クラスタータスクのタスク実行も削除されます。回避策として、**TaskRefKind** フィールドを使用して、タスク実行をフィルターリングします。

- **tkn triggertemplate describe** コマンドは、出力内の **apiVersion** 値の一部のみを表示します。たとえば、**triggers.tekton.dev/v1alpha1** ではなく、**triggers.tekton.dev** のみが表示されました。このバグは修正されています。

- 特定の条件下で Webhook はリースの取得に失敗し、正常に機能しません。このバグは修正されています。

- v0.16.3 で作成した When 式を持つパイプラインは、v0.17.1 以降で実行できるようになりました。アップグレード後に、アノテーションの最初の大文字と小文字の両方がサポートされるようになったため、以前のバージョンで作成されたパイプライン定義を再適用する必要はありません。

- デフォルトでは、**leader-election-ha** フィールドが高可用性に対して有効にされるようになりました。コントローラーフラグ **disable-ha** を **true** に設定すると、高可用性サポートが無効になります。

- 重複したクラウドイベントに関する問題が修正されています。クラウドイベントは、条件が状態、理由、またはメッセージを変更する場合にのみ送信されるようになりました。

- サービスアカウント名が **PipelineRun** または **TaskRun** 仕様にない場合、コントローラーは **config-defaults** 設定マップからサービスアカウント名を使用します。サービスアカウント名が **config-defaults** 設定マップにもない場合、コントローラーはこれを仕様で **default** に設定するようになりました。

- アフィニティーアシスタントとの互換性の検証は、同じ永続ボリューム要求 (PVC) が複数のワークスペースに使用される場合にサポートされるようになりましたが、サブパスは異なります。

### 3.1.4. Red Hat OpenShift Pipelines テクノロジープレビュー 1.2 のリリースノート

#### 3.1.4.1. 新機能

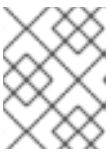
Red Hat OpenShift Pipelines テクノロジープレビュー (TP) 1.2 が OpenShift Container Platform 4.6 で利用可能になりました。Red Hat OpenShift Pipelines TP 1.2 が以下をサポートするように更新されています。

- Tekton Pipelines 0.16.3
- Tekton **tkn** CLI 0.13.1
- Tekton Triggers 0.8.1
- Tekton Catalog 0.16 をベースとするクラスタータスク
- OpenShift Container Platform 4.6 での IBM Power Systems
- OpenShift Container Platform 4.6 での IBM Z および LinuxONE

以下では、修正および安定性の面での改善点に加え、OpenShift Pipelines 1.2 の主な新機能について説明します。

##### 3.1.4.1.1. パイプライン

- Red Hat OpenShift Pipelines のリリースでは、非接続インストールのサポートが追加されました。



**注記**

制限された環境でのインストールは現時点で、IBM Power Systems、IBM Z、および LinuxONE ではサポートされていません。

- **conditions** リソースの代わりに **when** フィールドを使用して、特定の条件が満たされる場合にのみタスクを実行できるようになりました。**WhenExpression** の主なコンポーネントは **Input**、 **Operator**、および **Values** です。すべての When 式が **True** に評価されると、タスクが実行されます。When 式のいずれかが **False** に評価されると、タスクはスキップされます。
- ステップのステータスは、タスクの実行がキャンセルまたはタイムアウトすると更新されるようになりました。
- **git-init** が使用するベースイメージをビルドするために、Git Large File Storage (LFS) のサポートが利用できるようになりました。
- **taskSpec** フィールドを使用して、タスクがパイプラインに組み込まれる際に、ラベルやアノテーションなどのメタデータを指定できるようになりました。
- クラウドイベントがパイプラインの実行でサポートされるようになりました。**backoff** を使用した再試行が、クラウドイベントパイプラインリソースによって送信されるクラウドイベントに対して有効になりました。

- **Task** リソースが宣言するものの、**TaskRun** リソースが明示的に指定しないワークスペースのデフォルトの **Workspace** 設定を設定できるようになりました。

- サポートは、**PipelineRun** namespace および **TaskRun** namespace の namespace 変数の補間に利用できます。

- **TaskRun** オブジェクトの検証が追加され、**TaskRun** リソースが Affinity Assistant に関連付けられる際に複数の永続ボリューム要求 (PVC) ワークスペースが使用されていないことを確認するようになりました。複数の永続ボリューム要求 (PVC) ワークスペースが使用されていると、タスクの実行は **TaskRunValidationFailed** の状態で失敗します。デフォルトで、Affinity Assistant は Red Hat OpenShift Pipelines で無効にされているため、これを使用できるように有効にする必要があります。

#### 3.1.4.1.2. Pipelines CLI

- **tkn task describe**、**tkn taskrun describe**、**tkn clustertask describe**、**tkn pipeline describe**、および **tkn pipelinerun describe** コマンドが以下を実行するようになりました。

  - **Task**、**TaskRun**、**ClusterTask**、**Pipeline** および **PipelineRun** リソースのいずれかが 1 つしかない場合、それぞれを自動的に選択します。

  - 出力に **Task**、**TaskRun**、**ClusterTask**、**Pipeline** および **PipelineRun** リソースの結果をそれぞれ表示します。

  - 出力に **Task**、**TaskRun**、**ClusterTask**、**Pipeline** および **PipelineRun** リソースで宣言されたワークスペースをそれぞれ表示します。

- **tkn clustertask start** コマンドに **--prefix-name** オプションを指定して、タスク実行の名前に接頭辞を指定できるようになりました。

- インタラクティブモードのサポートが **tkn clustertask start** コマンドに提供されるようになりました。

- **TaskRun** および **PipelineRun** オブジェクトのローカルまたはリモートファイル定義を使用して、パイプラインでサポートされる **PodTemplate** プロパティーを指定できるようになりました。

- **--use-params-defaults** オプションを **tkn clustertask start** コマンドに指定して、**ClusterTask** 設定に設定したデフォルト値を使用して、タスク実行を作成できるようになりました。

- **tkn pipeline start** コマンドの **--use-param-defaults** フラグで、デフォルトの値が一部のパラメーターに指定されていない場合に対話モードをプロンプトで表示するようになりました。

#### 3.1.4.1.3. トリガー

- YAML 文字列を文字列のマップに解析するために、**parseYAML** という名前の Common Expression Language (CEL) 関数が追加されました。

- 式を評価する際や、評価環境を作成するためにフック本体を解析する際に、CEL 式の解析を行うエラーメッセージの詳細度が上がりました。

- ブール値とマップが CEL オーバーレイメカニズムで式の値として使用されている場合に、それらをマーシャリングするためのサポートが利用できるようになりました。

- 以下のフィールドが **EventListener** オブジェクトに追加されました。

- **replicas** フィールドは、YAML ファイルのレプリカ数を指定して、イベントリスナーが複数の Pod を実行できるようにします。
- **NodeSelector** フィールドでは、**EventListener** オブジェクトがイベントリスナー Pod を特定のノードにスケジュールできるようにします。

- Webhook インターセプターは **EventListener-Request-URL** ヘッダーを解析し、イベントリスナーによって処理される元のリクエスト URL からパラメーターを抽出できるようになりました。
- イベントリスナーからのアノテーションがデプロイメント、サービス、およびその他の Pod に伝播できるようになりました。サービスまたはデプロイメントのカスタムアノテーションは上書きされるため、イベントリスナーアノテーションに追加して伝播できるようにする必要があります。
- **EventListener** 仕様のレプリカの適切な検証が、ユーザーが **spec.replicas** 値を **negative** または **zero** として指定する場合に利用できるようになりました。
- **TriggerCRD** オブジェクトを、**TriggerRef** フィールドを使用して参照として **EventListener** 仕様内に指定し、**TriggerCRD** オブジェクトを別個に作成してから、これを **EventListener** 仕様内でバインドできるようになりました。
- **TriggerCRD** オブジェクトの検証およびデフォルト値が利用可能になりした。

#### 3.1.4.2. 非推奨の機能

- **$(params)** パラメーターは **triggertemplate** リソースから削除され、**$(tt.params)** に置き換えられ、これにより **resourcetemplate** と **triggertemplate** パラメーター間の混乱が生じなくなります。
- オプションの **EventListenerTrigger** ベースの認証レベルの **ServiceAccount** 参照が **ServiceAccountName** 文字列へのオブジェクト参照から変更されました。これにより、**ServiceAccount** 参照が **EventListenerTrigger** オブジェクトと同じ namespace に置かれるようになりました。
- **Conditions** カスタムリソース定義 (CRD) は非推奨となり、代わりに **WhenExpressions** CRD が使用されます。
- **PipelineRun.Spec.ServiceAccountNames** オブジェクトは非推奨となり、**PipelineRun.Spec.TaskRunSpec[].ServiceAccountName** オブジェクトによって置き換えられます。

#### 3.1.4.3. 既知の問題

- Red Hat OpenShift Pipelines のリリースでは、非接続インストールのサポートが追加されました。ただし、クラスタータスクで使用される一部のイメージは、非接続クラスターで動作するようにミラーリングする必要があります。
- **openshift** namespace のパイプラインは、Red Hat OpenShift Pipelines Operator のアンインストール後に削除されません。**oc delete pipelines -n openshift --all** コマンドを使用してパイプラインを削除します。
- Red Hat OpenShift Pipelines Operator をアンインストールしても、イベントリスナーは削除されません。
  回避策として、**EventListener** および **Pod** CRD を削除するには、以下を実行します。

1. **EventListener** オブジェクトを **foregroundDeletion** ファイナライザーで編集します。

```
$ oc patch el/<eventlistener_name> -p '{"metadata":{"finalizers":["foregroundDeletion"]}}'
--type=merge
```

以下に例を示します。

```
$ oc patch el/github-listener-interceptor -p '{"metadata":{"finalizers":
["foregroundDeletion"]}}' --type=merge
```

2. **EventListener** CRD を削除します。

```
$ oc patch crd/eventlisteners.triggers.tekton.dev -p '{"metadata":{"finalizers":[]}}' --
type=merge
```

- IBM Power Systems (ppc64le) または IBM Z (s390x) クラスターでコマンド仕様なしにマルチアーキテクチャーコンテナーイメージタスクを実行すると、**TaskRun** リソースは以下のエラーを出して失敗します。

```
Error executing command: fork/exec /bin/bash: exec format error
```

回避策として、アーキテクチャー固有のコンテナーイメージを使用するか、または正しいアーキテクチャーを参照する sha256 ダイジェストを指定します。sha256 ダイジェストを取得するには、以下を実行します。

```
$ skopeo inspect --raw <image_name>| jq '.manifests[] | select(.platform.architecture == "
<architecture>") | .digest'
```

#### 3.1.4.4. 修正された問題

- CEL フィルター、Webhook バリデーターのオーバーレイ、およびインターセプターの式を確認するための簡単な構文検証が追加されました。
- Trigger は、基礎となるデプロイメントおよびサービスオブジェクトに設定されたアノテーションを上書きしなくなりました。
- 以前のバージョンでは、イベントリスナーはイベントの受け入れを停止しました。今回の修正により、この問題を解決するために **EventListener** シンクの 120 秒のアイドルタイムアウトが追加されました。
- 以前のバージョンでは、**Failed(Canceled)** 状態でパイプラインの実行を取り消すと、成功のメッセージが表示されました。これは、代わりにエラーが表示されるように修正されました。
- **tkn eventlistener list** コマンドが一覧表示されたイベントリスナーのステータスを提供するようになり、利用可能なイベントリスナーを簡単に特定できるようになりました。
- トリガーがインストールされていない場合や、リソースが見つからない場合に、**triggers list** および **triggers describe** コマンドについて一貫性のあるエラーメッセージが表示されるようになりました。
- 以前のバージョンでは、多くのアイドル接続がクラウドイベントの配信時に増大しました。この問題を修正するために、**DisableKeepAlives: true** パラメーターが **cloudeventclient** 設定に追加されました。新規の接続がすべてのクラウドイベントに設定されます。

- 以前のバージョンでは、特定のタイプの認証情報が指定されていない場合であっても、**creds-init** コードが空のファイルをディスクに書き込みました。今回の修正により、**creds-init** コードが変更され、正しくアノテーションが付けられたシークレットから実際にマウントされた認証情報のみのファイルを書き込むようになりました。

### 3.1.5. Red Hat OpenShift Pipelines テクノロジープレビュー 1.1 のリリースノート

#### 3.1.5.1. 新機能

Red Hat OpenShift Pipelines テクノロジープレビュー (TP) 1.1 が OpenShift Container Platform 4.5 で利用可能になりました。Red Hat OpenShift Pipelines TP 1.1 が以下をサポートするように更新されています。

- Tekton Pipelines 0.14.3
- Tekton **tkn** CLI 0.11.0
- Tekton Triggers 0.6.1
- Tekton Catalog 0.14 をベースとするクラスタータスク

以下では、修正および安定性の面での改善点に加え、OpenShift Pipelines 1.1 の主な新機能について説明します。

##### 3.1.5.1.1. パイプライン

- ワークスペースをパイプラインリソースの代わりに使用できるようになりました。パイプラインリソースはデバッグが容易ではなく、スコープの制限があり、タスクの再利用を可能にしないため、OpenShift Pipelines ではワークスペースを使用することが推奨されます。ワークスペースの詳細は、OpenShift Pipelines のセクションを参照してください。
- ボリューム要求テンプレートのワークスペースのサポートが追加されました。
  - パイプライン実行およびタスク実行のボリューム要求テンプレートがワークスペースのボリュームソースとして追加できるようになりました。次に、tekton-controller はパイプラインのすべてのタスク実行の PVC として表示されるテンプレートを使用して永続ボリューム要求 (PVC) を作成します。したがって、複数のタスクにまたがるワークスペースをバインドするたびに PVC 設定を定義する必要はありません。
  - ボリューム要求テンプレートがボリュームソースとして使用される場合の PVC の名前検索のサポートが、変数の置換を使用して利用できるようになりました。
- 監査を強化するサポート:
  - **PipelineRun.Status** フィールドには、パイプラインのすべてのタスク実行のステータスと、パイプライン実行の進捗をモニターするためにパイプライン実行をインスタンス化する際に使用するパイプライン仕様が含まれるようになりました。
  - Pipeline の結果が Pipeline 仕様および **PipelineRun** ステータスに追加されました。
  - **TaskRun.Status** フィールドには、**TaskRun** リソースのインスタンス化に使用される実際のタスク仕様が含まれるようになりました。
- デフォルトパラメーターを各種の状態に適用するサポート。

- クラスタータスクを参照して作成されるタスク実行は、**tekton.dev/task** ラベルではなく **tekton.dev/clusterTask** ラベルを追加するようになりました。

- kube config writer は、kubeconfig-creator タスクでパイプラインリソースタイプクラスターの置き換えを有効にするために **ClientKeyData** および **ClientCertificateData** 設定をリソース構造に追加できるようになりました。

- **feature-flags** および **config-defaults** 設定マップの名前はカスタマイズ可能になりました。

- タスク実行で使用される Pod テンプレートのホストネットワークのサポートが追加されました。

- Affinity Assistant が、ワークスペースボリュームを共有するタスク実行のノードのアフィニティーをサポートするようになりました。デフォルトで、これは OpenShift Pipelines で無効にされます。

- Pod テンプレートは、Pod の起動時にコンテナーイメージのプルを許可するためにコンテナーランタイムが使用するシークレットを特定するために **imagePullSecrets** を指定するように更新されました。

- コントローラーがタスク実行の更新に失敗した場合にタスク実行コントローラーから警告イベントを出すためのサポート。

- アプリケーションまたはコンポーネントに属するリソースを特定するために、すべてのリソースに標準または推奨される k8s ラベルが追加されました。

- **Entrypoint** プロセスがシグナルについて通知されるようになり、これらのシグナルは **Entrypoint** プロセスの専用の PID グループを使用して伝播されるようになりました。

- Pod テンプレートはタスク実行仕様を使用してランタイム時にタスクレベルで設定できるようになりました。

- Kubernetes イベントを生成するサポート。

  - コントローラーは、追加のタスク実行ライフサイクルイベント (**taskrun started** および **taskrun running**) のイベントを生成するようになりました。

  - パイプライン実行コントローラーは、パイプラインの起動時に毎回イベントを生成するようになりました。

- デフォルトの Kubernetes イベントのほかに、タスク実行のクラウドイベントのサポートが利用可能になりました。コントローラーは、クラウドイベントとして create、started、および failed などのタスク実行イベントを送信するように設定できます。

- パイプライン実行およびタスク実行の場合に適切な名前を参照するための **$context.<taskRun|pipeline|pipelineRun>.name** 変数を使用するサポート。

- パイプライン実行パラメーターの検証が、パイプラインで必要なすべてのパラメーターがパイプライン実行によって提供できるようにするために利用可能になりました。これにより、パイプライン実行は必要なパラメーターに加えて追加のパラメーターを指定することもできます。

- パイプライン YAML ファイルの **finally** フィールドを使用して、すべてのタスクが正常に終了するか、またはパイプラインのタスクの失敗後、パイプラインが終了する前に常に実行されるパイプライン内でタスクを指定できるようになりました。

- **git-clone** クラスタータスクが利用できるようになりました。

#### 3.1.5.1.2. Pipelines CLI

- 組み込まれた Trigger バインディングのサポートが、**tkn evenlistener describe** コマンドで利用できるようになりました。
- 正しくないサブコマンドが使用される場合にサブコマンドを推奨し、提案するためのサポート。
- **tkn task describe** コマンドは、1つのタスクのみがパイプラインに存在する場合にタスクを自動的に選択できるようになりました。
- **--use-param-defaults** フラグを **tkn task start** コマンドに指定することにより、デフォルトのパラメーター値を使用してタスクを起動できるようになりました。
- **--workspace** オプションを **tkn pipeline start** または **tkn task start** コマンドで使用して、パイプライン実行またはタスク実行のボリューム要求テンプレートを指定できるようになりました。
- **tkn pipelinerun logs** コマンドに、**finally** セクションに一覧表示される最終タスクのログが表示されるようになりました。
- インタラクティブモードのサポートが、以下の **tkn** リソース向けに **tkn task start** コマンドおよび **describe** サブコマンドに追加されました: **pipeline**、**pipelinerun**、**task**、**taskrun**、**clustertask** および **pipelineresource**。
- **tkn version** コマンドで、クラスターにインストールされているトリガー のバージョンが表示されるようになりました。
- **tkn pipeline describe** コマンドで、パイプラインで使用されるタスクに指定されたパラメーター値およびタイムアウトが表示されるようになりました。
- 最近のパイプライン実行またはタスク実行をそれぞれ記述できるように、**tkn pipelinerun describe** および **tkn taskrun describe** コマンドの **--last** オプションのサポートが追加されました。
- **tkn pipeline describe** コマンドに、パイプラインのタスクに適用される各種の状態が表示されるようになりました。
- **--no-headers** および **--all-namespaces** フラグを **tkn resource list** コマンドで使用できるようになりました。

#### 3.1.5.1.3. トリガー

- 以下の Common Expression Language (CEL) 機能が利用できるようになりました。
  - **parseURL**: URL の一部を解析し、抽出します。
  - **parseJSON**: **deployment** webhook の **payload** フィールドの文字列に埋め込まれた JSON 値タイプを解析します。
- Bitbucket からの Webhook の新規インターセプターが追加されました。
- イベントリスナーは、**kubectl get** コマンドで一覧表示される際の追加フィールドとして **Address URL** および **Available status** を表示します。

- トリガーテンプレートパラメーターは、**$(params.<paramName>)** ではなく **$(tt.params.<paramName>)** 構文を使用するようになり、トリガーテンプレートとリソーステンプレートパラメーター間で生じる混乱が軽減されました。
- **EventListener** CRD に **tolerations** を追加し、セキュリティーや管理上の問題によりすべてのノードにテイントのマークが付けられる場合でもイベントリスナーが同じ設定でデプロイされるようにできるようになりました。
- イベントリスナー Deployment の Readiness Probe を **URL/live** に追加できるようになりました。
- イベントリスナートリガーでの **TriggerBinding** 仕様の埋め込みのサポート。
- Trigger リソースに推奨される **app.kubernetes.io** ラベルでアノテーションが付けられるようになりました。

#### 3.1.5.2. 非推奨の機能

本リリースでは、以下の項目が非推奨になりました。

- **clustertask** コマンドおよび **clustertriggerbinding** コマンドを含む、クラスター全体のすべてのコマンドの **--namespace** または **-n** フラグが非推奨になりました。これは今後のリリースで削除されます。
- **ref** フィールドが優先されるため、イベントリスナー内の **triggers.bindings** の **name** フィールドは非推奨となり、今後のリリースで削除されます。
- **$(tt.params)** が優先されるため、**$(params)** を使用したトリガーテンプレートの変数の補間が非推奨となり、これにより、パイプライン変数の補間構文に関連した混乱が軽減されました。**$(params.<paramName>)** 構文は今後のリリースで削除されます。
- **tekton.dev/task** ラベルはクラスタータスクで非推奨になりました。
- **TaskRun.Status.ResourceResults.ResourceRef** フィールドは非推奨となり、今後削除されます。
- **tkn pipeline create**、 **tkn task create**、および **tkn resource create -f** サブコマンドが削除されました。
- namespace の検証が **tkn** コマンドから削除されました。
- **tkn ct start** コマンドのデフォルトタイムアウトの **1h** および **-t** フラグが削除されました。
- **s2i** クラスタータスクが非推奨になりました。

#### 3.1.5.3. 既知の問題

- 各種の状態はワークスペースには対応しません。
- **--workspace** オプションとおよびインタラクティブモードは **tkn clustertask start** コマンドではサポートされていません。
- **$(params.<paramName>)** 構文の後方互換性のサポートにより、トリガーテンプレートがパイプライン固有のパラメーターで強制的に使用されます。 トリガー webhook がトリガーパラメーターとパイプラインパラメーターを区別できないためです。

- Pipeline メトリクスは、**tekton_taskrun_count** および **tekton_taskrun_duration_seconds_count** の promQL を実行する際に正しくない値を報告します。

- パイプライン実行およびタスク実行は、存在しない PVC 名がワークスペースに指定されている場合でも、それぞれ **Running** および **Running(Pending)** の状態のままになります。

#### 3.1.5.4. 修正された問題

- 以前のバージョンでは、タスクおよびクラスタータスクの名前が同じ場合、**tkn task delete <name> --trs** コマンドは、タスクとクラスタータスクの両方を削除しました。今回の修正により、コマンドはタスク **<name>** で作成されるタスク実行のみを削除するようになりました。

- 以前のバージョンでは、**tkn pr delete -p <name> --keep 2** コマンドは、**--keep** フラグと共に使用する場合に **-p** フラグを無視し、最新の 2 つのパイプライン実行を除きすべてのパイプライン実行を削除しました。今回の修正により、コマンドは最新の 2 つのパイプライン実行を除き、パイプライン **<name>** で作成されるパイプライン実行のみを削除するようになりました。

- **tkn triggertemplate describe** 出力には、YAML 形式ではなくテーブル形式でリソーステンプレートが表示されるようになりました。

- 以前のバージョンでは、**buildah** クラスタータスクは、新規ユーザーがコンテナーに追加されると失敗していました。今回の修正により、この問題は解決されています。

### 3.1.6. Red Hat OpenShift Pipelines テクノロジープレビュー 1.0 のリリースノート

#### 3.1.6.1. 新機能

Red Hat OpenShift Pipelines テクノロジープレビュー (TP) 1.0 が OpenShift Container Platform 4.4 で利用可能になりました。Red Hat OpenShift Pipelines TP 1.0 が以下をサポートするように更新されています。

- Tekton Pipelines 0.11.3

- Tekton **tkn** CLI 0.9.0

- Tekton Triggers 0.4.0

- Tekton Catalog 0.11 をベースとするクラスタータスク

以下では、修正および安定性の面での改善点に加え、OpenShift Pipelines 1.0 の主な新機能について説明します。

##### 3.1.6.1.1. パイプライン

- v1beta1 API バージョンのサポート。

- 改善された制限範囲のサポート。以前のバージョンでは、制限範囲はタスク実行およびパイプライン実行に対してのみ指定されていました。制限範囲を明示的に指定する必要がなくなりました。namespace 間で最小の制限範囲が使用されます。

- タスク結果およびタスクパラメーターを使用してタスク間でデータを共有するためのサポート。

- パイプラインは、**HOME** 環境変数および各ステップの作業ディレクトリーを上書きしないように設定できるようになりました。

- タスクステップと同様に、**sidecars** がスクリプトモードをサポートするようになりました。
- タスク実行 **podTemplate** リソースに別のスケジューラーの名前を指定できるようになりました。
- Star Array Notation を使用した変数置換のサポート。
- Tekton コントローラーは、個別の namespace を監視するように設定できるようになりました。
- パイプライン、タスク、クラスタータスク、リソース、および状態 (condition) の仕様に新規の説明フィールドが追加されました。
- Git パイプラインリソースへのプロキシーパラメーターの追加。

#### 3.1.6.1.2. Pipelines CLI

- **describe** サブコマンドが以下の **tkn** リソースについて追加されました。**EventListener**、**Condition**、**TriggerTemplate**、**ClusterTask**、および **TriggerSBinding**。
- **v1beta1** についてのサポートが、**v1alpha1** の後方互換性と共に以下のコマンドに追加されました。**ClusterTask**、**Task**、 **Pipeline**、**PipelineRun**、および **TaskRun**。
- 以下のコマンドは、**--all-namespaces** フラグオプションを使用してすべての namespace からの出力を一覧表示できるようになりました。これらは、**tkn task list**、**tkn pipeline list**、**tkn taskrun list**、**tkn pipelinerun list** です。
  これらのコマンドの出力は、**--no-headers** フラグオプションを使用してヘッダーなしで情報を表示するように強化されています。
- **--use-param-defaults** フラグを **tkn pipelines start** コマンドに指定することにより、デフォルトのパラメーター値を使用してパイプラインを起動できるようになりました。
- ワークスペースのサポートが **tkn pipeline start** および **tkn task start** コマンドに追加されるようになりました。
- 新規の **clustertriggerbinding** コマンドが以下のサブコマンドと共に追加されました。**describe**、**delete**、および **list**。
- ローカルまたはリモートの **yaml** ファイルを使用してパイプラインの実行を直接開始できるようになりました。
- **describe** サブコマンドには、強化され、詳細化した出力が表示されるようになりました。**description**、**timeout**、 **param description**、および **sidecar status** などの新規フィールドの追加により、コマンドの出力に特定の **tkn** リソースについてのより詳細な情報が提供されるようになりました。
- **tkn task log** コマンドには、1 つのタスクが namespace に存在する場合にログが直接表示されるようになりました。

#### 3.1.6.1.3. トリガー

- Trigger は **v1alpha1** および **v1beta1** の両方のパイプラインリソースを作成できるようになりました。
- 新規 Common Expression Language (CEL) インターセプター機能 **compareSecret** のサポート。この機能は、文字列と CEL 式のシークレットを安全な方法で比較します。

- イベントリスナーのトリガーレベルでの認証および認可のサポート。

#### 3.1.6.2. 非推奨の機能

本リリースでは、以下の項目が非推奨になりました。

- 環境変数 **$HOME**、および **Steps** 仕様の変数 **workingDir** が非推奨となり、今後のリリースで変更される可能性があります。現時点で **Step** コンテナーでは、**HOME** および **workingDir** 変数が **/tekton/home** および **/workspace** 変数にそれぞれ上書きされます。
今後のリリースでは、これらの 2 つのフィールドは変更されず、コンテナーイメージおよび **Task** YAML で定義される値に設定されます。本リリースでは、**disable-home-env-overwrite** および **disable-working-directory-overwrite** フラグを使用して、**HOME** および **workingDir** 変数の上書きを無効にします。

- 以下のコマンドは非推奨となり、今後のリリースで削除される可能性があります。**tkn pipeline create**、 **tkn task create**。

- **tkn resource create** コマンドの **-f** フラグは非推奨になりました。これは今後のリリースで削除される可能性があります。

- **tkn clustertask create** コマンドの **-t** フラグおよび **--timeout** フラグ (秒単位の形式) は非推奨になりました。期間タイムアウトの形式のみがサポートされるようになりました (例: **1h30s**)。これらの非推奨のフラグは今後のリリースで削除される可能性があります。

#### 3.1.6.3. 既知の問題

- 以前のバージョンの Red Hat OpenShift Pipelines からアップグレードする場合は、既存のデプロイメントを削除してから Red Hat OpenShift Pipelines バージョン 1.0 にアップグレードする必要があります。既存のデプロイメントを削除するには、まずカスタムリソースを削除してから Red Hat OpenShift Pipelines Operator をアンインストールする必要があります。詳細は、Red Hat OpenShift Pipelines のアンインストールについてのセクションを参照してください。

- 同じ **v1alpha1** タスクを複数回送信すると、エラーが発生します。**v1alpha1** タスクの再送信時に、**oc apply** ではなく **oc replace** コマンドを使用します。

- **buildah** クラスタータスクは、新規ユーザーがコンテナーに追加されると機能しません。Operator がインストールされると、**buildah** クラスタータスクの **--storage-driver** フラグが指定されていないため、フラグはデフォルト値に設定されます。これにより、ストレージドライバーが正しく設定されなくなることがあります。新規ユーザーが追加されると、storage-driver が間違っている場合に、**buildah** クラスタータスクが以下のエラーを出して失敗します。

```
useradd: /etc/passwd.8: lock file already used
useradd: cannot lock /etc/passwd; try again later.
```

回避策として、**buildah-task.yaml** ファイルで **--storage-driver** フラグの値を **overlay** に手動で設定します。

1. **cluster-admin** としてクラスターにログインします。

```
$ oc login -u <login> -p <password> https://openshift.example.com:6443
```

2. **oc edit** コマンドを使用して **buildah** クラスタータスクを編集します。

```
$ oc edit clustertask buildah
```

**buildah** clustertask YAML ファイルの現行バージョンが **EDITOR** 環境変数で設定されたエディターで開かれます。

3. **Steps** フィールドで、以下の **command** フィールドを見つけます。

```
 command: ['buildah', 'bud', '--format=$(params.FORMAT)', '--tls-
verify=$(params.TLSVERIFY)', '--layers', '-f', '$(params.DOCKERFILE)', '-t',
'$(resources.outputs.image.url)', '$(params.CONTEXT)']
```

4. **command** フィールドを以下に置き換えます。

```
 command: ['buildah', '--storage-driver=overlay', 'bud', '--format=$(params.FORMAT)', '--
tls-verify=$(params.TLSVERIFY)', '--no-cache', '-f', '$(params.DOCKERFILE)', '-t',
'$(params.IMAGE)', '$(params.CONTEXT)']
```

5. ファイルを保存して終了します。

または、**Pipelines → Cluster Tasks → buildah** に移動して、**buildah** クラスタータスク YAML ファイルを Web コンソール上で直接変更することもできます。**Actions** メニューから **Edit Cluster Task** を選択し、直前の手順のように **command** フィールドを置き換えます。

#### 3.1.6.4. 修正された問題

- 以前のリリースでは、**DeploymentConfig** タスクは、イメージのビルドがすでに進行中であっても新規デプロイメントビルドをトリガーしていました。これにより、パイプラインのデプロイメントが失敗していました。今回の修正により、**deploy task** コマンドが **oc rollout status** コマンドに置き換えられ、進行中のデプロイメントが終了するまで待機するようになりました。
- **APP_NAME** パラメーターのサポートがパイプラインテンプレートに追加されました。
- 以前のバージョンでは、Java S2I のパイプラインテンプレートはレジストリーでイメージを検索できませんでした。今回の修正により、イメージはユーザーによって提供される **IMAGE_NAME** パラメーターの代わりに既存イメージのパイプラインリソースを使用して検索されるようになりました。
- OpenShift Pipelines イメージはすべて、Red Hat Universal Base Images (UBI) をベースにしています。
- 以前のバージョンでは、パイプラインが **tekton-pipelines** 以外の namespace にインストールされている場合、**tkn version** コマンドはパイプラインのバージョンを **unknown** と表示していました。今回の修正により、**tkn version** コマンドにより、正しいパイプラインのバージョンがすべての namespace で表示されるようになりました。
- **-c** フラグは **tkn version** コマンドでサポートされなくなりました。
- 管理者以外のユーザーがクラスタートリガーバインディングを一覧表示できるようになりました。
- イベントリスナーの **CompareSecret** 機能が、CEL インターセプターについて修正されました。
- タスクおよびクラスタータスクの **list**、**describe**、および **start** サブコマンドは、タスクおよびクラスタータスクが同じ名前を持つ場合に出力に正常に表示されるようになりました。

- 以前のバージョンでは、OpenShift Pipelines Operator は特権付き SCC (Security Context Constraints) を変更していました。これにより、クラスターのアップグレード時にエラーが発生しました。このエラーは修正されています。

- **tekton-pipelines** namespace では、設定マップを使用して、すべてのタスク実行およびパイプライン実行のタイムアウトが **default-timeout-minutes** フィールドの値に設定されるようになりました。

- 以前のバージョンでは、Web コンソールのパイプラインセクションは管理者以外のユーザーには表示されませんでした。この問題は解決されています。

## 3.2. OPENSHIFT PIPELINES について

Red Hat OpenShift Pipelines は、Kubernetes リソースをベースとしたクラウドネイティブの継続的インテグレーションおよび継続的デリバリー (CI/CD) ソリューションです。これは Tekton ビルディングブロックを使用し、基礎となる実装の詳細を抽象化することで、複数のプラットフォームでのデプロイメントを自動化します。Tekton では、Kubernetes ディストリビューション間で移植可能な CI/CD パイプラインを定義するための標準のカスタムリソース定義 (CRD) が多数導入されています。

### 3.2.1. 主な特長

- Red Hat OpenShift Pipelines は、分離されたコンテナーで必要なすべての依存関係と共にパイプラインを実行するサーバーレスの CI/CD システムです。

- Red Hat OpenShift Pipelines は、マイクロサービスベースのアーキテクチャーで機能する分散型チーム向けに設計されています。

- Red Hat OpenShift Pipelines は、拡張および既存の Kubernetes ツールとの統合を容易にする標準の CI/CD パイプライン定義を使用し、オンデマンドのスケーリングを可能にします。

- Red Hat OpenShift Pipelines を使用して、Kubernetes プラットフォーム全体で移植可能な S2I (Source-to-Image)、Buildah、Buildpacks、および Kaniko などの Kubernetes ツールを使用してイメージをビルドできます。

- OpenShift Container Platform Developer Console を使用して、Tekton リソースの作成、パイプライン実行のログの表示、OpenShift Container Platform namespace でのパイプラインの管理を実行できます。

### 3.2.2. OpenShift Pipelines の概念

本書では、パイプラインの各種概念を詳述します。

#### 3.2.2.1. タスク

**Task** は Pipeline のビルディングブロックであり、順次実行されるステップで設定されます。これは基本的に入出力の機能です。Task は個別に実行することも、パイプラインの一部として実行することもできます。これらは再利用可能であり、複数の Pipeline で使用することができます。

**Step** は、イメージのビルドなど、Task によって順次実行され、特定の目的を達成するための一連のコマンドです。各 Task は Pod として実行され、各 Step は同じ Pod 内のコンテナーとして実行されます。Step は同じ Pod 内で実行されるため、ファイル、設定マップ、およびシークレットをキャッシュするために同じボリュームにアクセスできます。

以下の例は、**apply-manifests** Task を示しています。

```
apiVersion: tekton.dev/v1beta1 (1)
kind: Task (2)
metadata:
  name: apply-manifests (3)
spec: (4)
  workspaces:
  - name: source
  params:
    - name: manifest_dir
      description: The directory in source that contains yaml manifests
      type: string
      default: "k8s"
  steps:
    - name: apply
      image: image-registry.openshift-image-registry.svc:5000/openshift/cli:latest
      workingDir: /workspace/source
      command: ["/bin/bash", "-c"]
      args:
        - |-
          echo Applying manifests in $(params.manifest_dir) directory
          oc apply -f $(params.manifest_dir)
          echo -----------------------------------
```

(1) Task API バージョン **v1beta1**。

(2) Kubernetes オブジェクトのタイプ **Task**。

(3) この Task の一意の名前。

(4) Task のパラメーターおよび Step と、Task によって使用される Workspace の一覧。

この Task は Pod を起動し、指定されたコマンドを実行するために指定されたイメージを使用して Pod 内のコンテナーを実行されます。

#### 3.2.2.2. TaskRun

**TaskRun** は、クラスター上の特定の入出力、および実行パラメーターで実行するために Task をインスタンス化します。これは独自に起動することも、パイプラインの各 Task の PipelineRun の一部として起動することもできます。

Task はコンテナーイメージを実行する 1 つ以上の Step で設定され、各コンテナーイメージは特定のビルド作業を実行します。TaskRun は、すべての Step が正常に実行されるか、または失敗が発生するまで、指定された順序で Task の Step を実行します。TaskRun は、Pipeline の各 Task について PipelineRun によって自動的に作成されます。

以下の例は、関連する入力パラメーターで **apply-manifests** Task を実行する TaskRun を示しています。

```
apiVersion: tekton.dev/v1beta1 (1)
kind: TaskRun (2)
metadata:
  name: apply-manifests-taskrun (3)
spec: (4)
  serviceAccountName: pipeline
```

```
  taskRef: 5
    kind: Task
    name: apply-manifests
  workspaces: 6
  - name: source
    persistentVolumeClaim:
      claimName: source-pvc
```

1 TaskRun API バージョン **v1beta1**

2 Kubernetes オブジェクトのタイプを指定します。この例では、**TaskRun** です。

3 この TaskRun を識別する一意の名前。

4 TaskRun の定義。この TaskRun には、Task と必要な Workspace を指定します。

5 この TaskRun に使用される Task 参照の名前。この TaskRun は Task **apply-manifests** Task を実行します。

6 TaskRun によって使用される Workspace。

#### 3.2.2.3. パイプライン

**Pipeline** は、特定の実行順序で編成される **Task** リソースのコレクションです。これらは、アプリケーションのビルド、デプロイメント、およびデリバリーを自動化する複雑なワークフローを構築するために実行されます。1 つ以上のタスクを含むパイプラインを使用して、アプリケーションの CI/CD ワークフローを定義できます。

**Pipeline** 定義は、多くのフィールドまたは属性で設定され、Pipeline が特定の目的を達成することを可能にします。各 **Pipeline** リソース定義には、特定の入力を取り込み、特定の出力を生成する **Task** が少なくとも 1 つ含まれる必要があります。パイプライン定義には、アプリケーション要件に応じて **Conditions**、**Workspaces**、 **Parameters**、または **Resources** をオプションで含めることもできます。

以下の例は、**buildah ClusterTask** を使用して Git リポジトリーからアプリケーションイメージをビルドする **build-and-deploy** パイプラインを示しています。

```
apiVersion: tekton.dev/v1beta1 1
kind: Pipeline 2
metadata:
  name: build-and-deploy 3
spec: 4
  workspaces: 5
  - name: shared-workspace
  params: 6
  - name: deployment-name
    type: string
    description: name of the deployment to be patched
  - name: git-url
    type: string
    description: url of the git repo for the code of deployment
  - name: git-revision
    type: string
    description: revision to be used from repo of the code for deployment
    default: "pipelines-1.4"
```

```
  - name: IMAGE
    type: string
    description: image to be built from the code
  tasks: (7)
  - name: fetch-repository
    taskRef:
      name: git-clone
      kind: ClusterTask
    workspaces:
    - name: output
      workspace: shared-workspace
    params:
    - name: url
      value: $(params.git-url)
    - name: subdirectory
      value: ""
    - name: deleteExisting
      value: "true"
    - name: revision
      value: $(params.git-revision)
  - name: build-image (8)
    taskRef:
      name: buildah
      kind: ClusterTask
    params:
    - name: TLSVERIFY
      value: "false"
    - name: IMAGE
      value: $(params.IMAGE)
    workspaces:
    - name: source
      workspace: shared-workspace
    runAfter:
    - fetch-repository
  - name: apply-manifests (9)
    taskRef:
      name: apply-manifests
    workspaces:
    - name: source
      workspace: shared-workspace
    runAfter: (10)
    - build-image
  - name: update-deployment
    taskRef:
      name: update-deployment
    workspaces:
    - name: source
      workspace: shared-workspace
    params:
    - name: deployment
      value: $(params.deployment-name)
    - name: IMAGE
      value: $(params.IMAGE)
    runAfter:
    - apply-manifests
```

1. Pipeline API バージョン **v1beta1**。
2. Kubernetes オブジェクトのタイプを指定します。この例では、**Pipeline** です。
3. この Pipeline の一意の名前。
4. Pipeline の定義および構造を指定します。
5. Pipeline のすべての Task で使用される Workspace。
6. Pipeline のすべての Task で使用されるパラメーター。
7. Pipeline で使用される Task の一覧を指定します。
8. Task **build-image**: **buildah** ClusterTask を使用して、所定の Git リポジトリーからアプリケーションイメージをビルドします。
9. Task **apply-manifests**: 同じ名前のユーザー定義 Task を使用します。
10. Task が Pipeline で実行されるシーケンスを指定します。この例では、**apply-manifests** Task は **build-image** Task の完了後にのみ実行されます。

#### 3.2.2.4. PipelineRun

**PipelineRun** は、Pipeline の実行中のインスタンスです。これは、クラスター上の特定の入力、出力、および実行パラメーターで実行される Pipeline をインスタンス化します。対応する TaskRun は、PipelineRun の Task ごとに自動的に作成されます。

Pipeline のすべての Task は、すべての Task が正常に実行されるか、または Task が失敗するまで定義されたシーケンスで実行されます。**status** フィールドは、監視および監査のために、PipelineRun で各 TaskRun の進捗を追跡し、保存します。

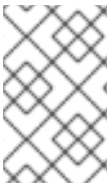
以下の例は、関連するリソースおよびパラメーターで **build-and-deploy** Pipeline を実行する PipelineRun を示しています。

```
apiVersion: tekton.dev/v1beta1 1
kind: PipelineRun 2
metadata:
  name: build-deploy-api-pipelinerun 3
spec:
  pipelineRef:
    name: build-and-deploy 4
  params: 5
  - name: deployment-name
    value: vote-api
  - name: git-url
    value: https://github.com/openshift-pipelines/vote-api.git
  - name: IMAGE
    value: image-registry.openshift-image-registry.svc:5000/pipelines-tutorial/vote-api
  workspaces: 6
  - name: shared-workspace
    volumeClaimTemplate:
      spec:
        accessModes:
          - ReadWriteOnce
```

```
    resources:
      requests:
        storage: 500Mi
```

1. PipelineRun API バージョン **v1beta1**。
2. Kubernetes オブジェクトのタイプを指定します。この例では、**PipelineRun** です。
3. この PipelineRun を識別する一意の名前。
4. 実行する Pipeline の名前。この例では、**build-and-deploy** です。
5. Pipeline の実行に必要なパラメーターの一覧を指定します。
6. PipelineRun によって使用される Workspace。

#### 3.2.2.5. Workspace



**注記**

PipelineResource はデバッグが容易ではなく、スコープの制限があり、Task を再利用可能にしないため、OpenShift Pipelines では PipelineResource の代わりに Workspace を使用することが推奨されます。

Workspace は、入力を受信し、出力を提供するために Pipeline の Task がランタイム時に必要とする共有ストレージボリュームを宣言します。Workspace では、ボリュームの実際の場所を指定する代わりに、ランタイム時に必要となるファイルシステムまたはファイルシステムの一部を宣言できます。Task または Pipeline は Workspace を宣言し、ボリュームの特定の場所の詳細を指定する必要があります。その後、これは TaskRun または PipelineRun の Workspace にマウントされます。ランタイムストレージボリュームからボリューム宣言を分離することで、Task を再利用可能かつ柔軟にし、ユーザー環境から切り離すことができます。

Workspace を使用すると、以下が可能になります。

- Task の入力および出力の保存
- Task 間でのデータの共有
- Secret に保持される認証情報のマウントポイントとして使用
- ConfigMap に保持される設定のマウントポイントとして使用
- 組織が共有する共通ツールのマウントポイントとして使用
- ジョブを高速化するビルドアーティファクトのキャッシュの作成

以下を使用して、TaskRun または PipelineRun で Workspace を指定できます。

- 読み取り専用 ConfigMap または Secret
- 他の Task と共有される既存の PersistentVolumeClaim
- 指定された VolumeClaimTemplate からの PersistentVolumeClaim
- TaskRun の完了時に破棄される emptyDir

以下の例は、Pipeline で定義される、**build-image** および **apply-manifests** Task の **shared-workspace** Workspace を宣言する **build-and-deploy** Pipeline のコードスニペットを示しています。

```
apiVersion: tekton.dev/v1beta1
kind: Pipeline
metadata:
  name: build-and-deploy
spec:
  workspaces: (1)
  - name: shared-workspace
  params:
...
  tasks: (2)
  - name: build-image
    taskRef:
      name: buildah
      kind: ClusterTask
    params:
    - name: TLSVERIFY
      value: "false"
    - name: IMAGE
      value: $(params.IMAGE)
    workspaces: (3)
    - name: source (4)
      workspace: shared-workspace (5)
    runAfter:
    - fetch-repository
  - name: apply-manifests
    taskRef:
      name: apply-manifests
    workspaces: (6)
    - name: source
      workspace: shared-workspace
    runAfter:
      - build-image
...
```

(1) Pipeline で定義される Task 間で共有される Workspace の一覧。Pipeline は、必要な数の Workspace を定義できます。この例では、**shared-workspace** という名前の 1 つの Workspace のみが宣言されます。

(2) Pipeline で使用される Task の定義。このスニペットは、共通の Workspace を共有する **build-image** および **apply-manifests** の 2 つの Task を定義します。

(3) **build-image** Task で使用される Workspace の一覧。Task 定義には、必要な数の Workspace を含めることができます。ただし、Task が最大 1 つの書き込み可能な Workspace を使用することが推奨されます。

(4) Task で使用される Workspace を一意に識別する名前。この Task は、**source** という名前の 1 つの Workspace を使用します。

(5) Task によって使用される Pipeline Workspace の名前。Workspace **source** は Pipeline Workspace の **shared-workspace** を使用することに注意してください。

(6) **apply-manifests** Task で使用される Workspace の一覧。この Task は、 **build-image** Task と

Workspace はタスクがデータを共有する際に使用でき、これにより、パイプラインの各タスクが実行時に必要となる 1 つまたは複数のボリュームを指定することができます。永続ボリューム要求 (PVC) を作成するか、または永続ボリューム要求 (PVC) を作成するボリューム要求テンプレートを指定できます。

以下の **build-deploy-api-pipelinerun** PipelineRun のコードスニペットは、**build-and-deploy** Pipeline で使用される **shared-workspace** Workspace のストレージボリュームを定義するための永続ボリューム要求 (PVC) を作成するために永続ボリュームテンプレートを使用します。

```
apiVersion: tekton.dev/v1beta1
kind: PipelineRun
metadata:
  name: build-deploy-api-pipelinerun
spec:
  pipelineRef:
    name: build-and-deploy
  params:
...

  workspaces: 1
  - name: shared-workspace 2
    volumeClaimTemplate: 3
      spec:
        accessModes:
          - ReadWriteOnce
        resources:
          requests:
            storage: 500Mi
```

1. PipelineRun にボリュームバインディングを提供する Pipeline Workspace の一覧を指定します。
2. ボリュームが提供されている Pipeline の Workspace の名前。
3. ワークスペースのストレージボリュームを定義するために永続ボリューム要求 (PVC) を作成するボリューム要求テンプレートを指定します。

#### 3.2.2.6. トリガー

**Trigger** をパイプラインと併用して、Kubernetes リソースで CI/CD 実行全体を定義する本格的な CI/CD システムを作成します。Trigger は、Git プルリクエストなどの外部イベントをキャプチャーし、それらのイベントを処理して情報の主要な部分を抽出します。このイベントデータを事前に定義されたパラメーターのセットにマップすると、Kubernetes リソースを作成およびデプロイし、パイプラインをインスタンス化できる一連のタスクがトリガーされます。

たとえば、アプリケーションの Red Hat OpenShift Pipelines を使用して CI/CD ワークフローを定義します。アプリケーションリポジトリーで新たな変更を有効にするには、パイプラインを開始する必要があります。トリガーは変更イベントをキャプチャーし、処理することにより、また新規イメージを最新の変更でデプロイするパイプライン実行をトリガーして、このプロセスを自動化します。

Trigger は、再利用可能で分離した自律型 CI/CD システムを設定するように連携する以下の主なリソースで設定されています。

- **TriggerBinding** リソースはイベントを検証し、イベントペイロードからフィールドを抽出し、それらをパラメーターとして保存します。

以下の例は、**TriggerBinding** リソースのコードスニペットを示しています。これは、受信イベントペイロードから Git リポジトリー情報を抽出します。

```
apiVersion: triggers.tekton.dev/v1alpha1 (1)
kind: TriggerBinding (2)
metadata:
  name: vote-app (3)
spec:
  params: (4)
  - name: git-repo-url
    value: $(body.repository.url)
  - name: git-repo-name
    value: $(body.repository.name)
  - name: git-revision
    value: $(body.head_commit.id)
```

(1) **TriggerBinding** リソースの API バージョン。この例では、**v1alpha1** です。

(2) Kubernetes オブジェクトのタイプを指定します。この例では、**TriggerBinding** です。

(3) この **TriggerBinding** を識別する一意の名前。

(4) 受信イベントペイロードから抽出され、 **TriggerTemplate** に渡されるパラメーターの一覧。この例では、Git リポジトリー URL、名前、およびリビジョンはイベントペイロードの本体から抽出されます。

- **TriggerTemplate** リソースは、リソースの作成方法の標準として機能します。これは、**TriggerBinding** リソースからのパラメーター化されたデータが使用される方法を指定します。トリガーテンプレートは、トリガーバインディングから入力を受信し、新規パイプラインリソースの作成および新規パイプライン実行の開始につながる一連のアクションを実行します。
以下の例は、**TriggerTemplate** リソースのコードスニペットを示しています。これは、作成した **TriggerBinding** リソースから受信される Git リポジトリー情報を使用してパイプライン実行を作成します。

```
apiVersion: triggers.tekton.dev/v1alpha1 (1)
kind: TriggerTemplate (2)
metadata:
  name: vote-app (3)
spec:
  params: (4)
  - name: git-repo-url
    description: The git repository url
  - name: git-revision
    description: The git revision
    default: pipelines-1.4
  - name: git-repo-name
    description: The name of the deployment to be created / patched

  resourcetemplates: (5)
  - apiVersion: tekton.dev/v1beta1
    kind: PipelineRun
    metadata:
```

```
    name: build-deploy-$(tt.params.git-repo-name)-$(uid)
  spec:
    serviceAccountName: pipeline
    pipelineRef:
      name: build-and-deploy
    params:
    - name: deployment-name
      value: $(tt.params.git-repo-name)
    - name: git-url
      value: $(tt.params.git-repo-url)
    - name: git-revision
      value: $(tt.params.git-revision)
    - name: IMAGE
      value: image-registry.openshift-image-registry.svc:5000/pipelines-
tutorial/$(tt.params.git-repo-name)
    workspaces:
    - name: shared-workspace
      volumeClaimTemplate:
       spec:
        accessModes:
         - ReadWriteOnce
        resources:
          requests:
            storage: 500Mi
```

1. **TriggerTemplate** リソースの API バージョン。この例では、**v1alpha1** です。
2. Kubernetes オブジェクトのタイプを指定します。この例では、**TriggerTemplate** です。
3. **TriggerTemplate** リソースを識別するための一意の名前。
4. **TriggerBinding** または **EventListerner** リソースによって提供されるパラメーター。
5. **TriggerBinding** または **EventListener** リソースを使用して受信されるパラメーターを使用してリソースを作成する必要がある方法を指定するテンプレートの一覧。

- **Trigger** リソースは **TriggerBinding** および **TriggerTemplate** リソースを接続し、この **Trigger** リソースは **EventListener** 仕様で参照されます。
以下の例は、**TriggerBinding** および **TriggerTemplate** リソースを接続する **vote-trigger** という名前の **Trigger** リソースのコードスニペットを示しています。

```
apiVersion: triggers.tekton.dev/v1alpha1 (1)
kind: Trigger (2)
metadata:
  name: vote-trigger (3)
spec:
  serviceAccountName: pipeline (4)
  bindings:
    - ref: vote-app (5)
  template: (6)
     ref: vote-app
```

1. **Trigger** リソースの API バージョン。この例では、**v1alpha1** です。

2 Kubernetes オブジェクトのタイプを指定します。この例では、**Trigger** です。

3 この **Trigger** リソースを識別するための一意の名前。

4 使用されるサービスアカウント名。

5 **TriggerTemplate** リソースに接続する **TriggerBinding** リソースの名前。

6 **TriggerBinding** リソースに接続するための **TriggerTemplate** リソースの名前。

- **EventListener** は、JSON ペイロードを含む受信 HTTP ベースイベントをリッスンするエンドポイントまたはイベントシンクを提供します。これは各 **TriggerBinding** リソースからイベントパラメーターを抽出し、次にこのデータを処理し、対応する **TriggerTemplate** リソースによって指定される Kubernetes リソースを作成します。**EventListener** リソースは、イベントの **interceptors** を使用してペイロードで軽量イベント処理または基本的なフィルターを実行します。これはペイロードのタイプを特定し、オプションでこれを変更します。現時点で、パイプライントリガーは **Webhook インターセプター**、 **GitHub インターセプター**、 **GitLab インターセプター**、および **Common Expression Language (CEL) インターセプター**の 4 種類のインターセプターをサポートします。
  以下の例は、**vote-trigger** という名前の **Trigger** リソースを参照する **EventListener** リソースを示しています。

```
apiVersion: triggers.tekton.dev/v1alpha1 1
kind: EventListener 2
metadata:
  name: vote-app 3
spec:
  serviceAccountName: pipeline 4
  triggers:
    - triggerRef: vote-trigger 5
```

1 **EventListener** リソースの API バージョン。この例では、**v1alpha1** です。

2 Kubernetes オブジェクトのタイプを指定します。この例では、**EventListener** です。

3 **EventListener** リソースを識別するための一意の名前。

4 使用されるサービスアカウント名。

5 **EventListener** リソースによって参照される **Trigger** リソースの名前。

Red Hat OpenShift Pipelines のトリガーは、**Eventlistener** リソースへの HTTP(非セキュア) および HTTPS (セキュアな HTTP) 接続の両方をサポートします。セキュアな HTTPS 接続を使用すると、クラスター内外のエンドツーエンドのセキュアな接続を得ることができます。namespace の作成後に、**operator.tekton.dev/enable-annotation=enabled** ラベルを namespace に追加し、次に **Trigger** リソースおよび re-encrypt TLS 終端を使用するセキュアなルートを作成して、**Eventlistener** についてこのセキュアな HTTPS 接続を有効にできます。

### 3.2.3. 関連情報

- パイプラインのインストールについての詳細は、OpenShift Pipelines のインストール を参照してください。

- カスタムの CI/CD ソリューションの作成についての詳細は、CI/CD パイプラインを使用したアプリケーションの作成 を参照してください。
- re-encrypt TLS 終端についての詳細は、 再暗号化終端 について参照してください。
- セキュリティー保護されたルートについての詳細は、セキュリティー保護されたルート についてのセクションを参照してください。

## 3.3. OPENSHIFT PIPELINES のインストール

以下では、クラスター管理者を対象に、Red Hat OpenShift Pipelines Operator の OpenShift Container Platform クラスターへのインストールプロセスについて説明します。

**前提条件**

- **cluster-admin** パーミッションを持つアカウントを使用して OpenShift Container Platform クラスターにアクセスできる。
- **oc** CLI がインストールされていること。
- OpenShift Pipelines (**tkn**) CLI がローカルシステムにインストールされていること。

### 3.3.1. Web コンソールでの Red Hat OpenShift Pipelines Operator のインストール

OpenShift Container Platform OperatorHub に一覧表示されている Operator を使用して Red Hat OpenShift Pipelines をインストールできます。Red Hat OpenShift Pipelines Operator をインストールする際に、パイプラインの設定に必要なカスタムリソース (CR) は Operator と共に自動的にインストールされます。

デフォルトの Operator カスタムリソース定義 (CRD) の **config.operator.tekton.dev** が **tektonconfigs.operator.tekton.dev** に置き換えられました。さらに Operator は、個別に管理される OpenShift Pipelines コンポーネントに追加の CRD (**tektonpipelines.operator.tekton.dev**、**tektontriggers.operator.tekton.dev** および **tektonaddons.operator.tekton.dev**) を提供します。

OpenShift Pipelines がクラスターにすでにインストールされている場合、既存のインストールはシームレスにアップグレードされます。Operator は必要に応じて、クラスターの **config.operator.tekton.dev** のインスタンスを **tektonconfigs.operator.tekton.dev** のインスタンスと、その他の CRD の追加オブジェクトに置き換えます。

**警告**

既存のインストールを手動で変更した場合 (**resource name - cluster** フィールドに変更を加えて **config.operator.tekton.dev** CRD インスタンスのターゲット namespace を変更する場合など) は、アップグレードパスはスムーズではありません。このような場合は、インストールをアンインストールし、Red Hat OpenShift Pipelines Operator を再インストールするワークフローが推奨されます。

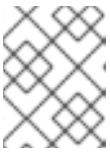
Red Hat OpenShift Pipelines Operator は、**TektonConfig** CR の一部としてプロファイルを指定して、インストールするコンポーネントを選択するオプションを提供するようになりました。**TektonConfig** CR は Operator のインストール時に自動的にインストールされます。サポートされるプロファイルは以

下のとおりです。

- Basic: これは Tekton パイプラインのみをインストールします。
- Default: これは Tekton パイプラインと Tekton トリガーをインストールします。
- All: これは **TektonConfig** CR のインストール時に使用されるデフォルトプロファイルです。このプロファイルは、Tekton Pipelines、Tekton Triggers、Tekton Addons (**ClusterTasks**、**ClusterTriggerBindings**、**ConsoleCLIDownload**、**ConsoleQuickStart** および **ConsoleYAMLSample** リソースを含む) のすべてをインストールします。

**手順**

1. Web コンソールの **Administrator** パースペクティブで、**Operators → OperatorHub** に移動します。
2. **Filter by keyword** ボックスを使用して、カタログで **Red Hat OpenShift Pipelines** Operator を検索します。**Red Hat OpenShift Pipelines** Operator タイルをクリックします。
3. **Red Hat OpenShift Pipelines** Operator ページで Operator についての簡単な説明を参照してください。**Install** をクリックします。
4. **Install Operator** ページで以下を行います。
   a. **Installation Mode** について **All namespaces on the cluster (default)**を選択します。このモードは、デフォルトの **openshift-operators** namespace で Operator をインストールし、Operator がクラスターのすべての namespace を監視し、これらの namespace に対して利用可能になるようにします。
   b. **Approval Strategy** について **Automatic** を選択します。これにより、Operator への今後のアップグレードは Operator Lifecycle Manager (OLM) によって自動的に処理されます。**Manual** 承認ストラテジーを選択すると、OLM は更新要求を作成します。クラスター管理者は、Operator を新規バージョンに更新できるように OLM 更新要求を手動で承認する必要があります。
   c. **Update Channel** を選択します。
      - **Stable** チャネルは、Red Hat OpenShift Pipelines Operator の最新の安定したサポートされているリリースのインストールを可能にします。
      - **preview** チャネルは、Red Hat OpenShift Pipelines Operator の最新プレビューバージョンのインストールを有効にします。これには、**Stable** チャネルでは利用できず、サポートされていない機能が含まれる場合があります。
5. **Install** をクリックします。Operator が **Installed Operators** ページに一覧表示されます。



**注記**

Operator は **openshift-operators** namespace に自動的にインストールされます。

6. **Status** が **Succeeded Up to date** に設定され、Red Hat OpenShift Pipelines Operator のインストールが正常に行われたことを確認します。

### 3.3.2. CLI を使用した OpenShift Pipelines Operator のインストール

CLI を使用して OperatorHub から Red Hat OpenShift Pipelines Operator をインストールできます。

**手順**

1. Subscription オブジェクトの YAML ファイルを作成し、namespace を Red Hat OpenShift Pipelines Operator にサブスクライブします (例: **sub.yaml**)。

   **Subscription の例**

```
apiVersion: operators.coreos.com/v1alpha1
kind: Subscription
metadata:
  name: openshift-pipelines-operator
  namespace: openshift-operators
spec:
  channel:  <channel name> 1
  name: openshift-pipelines-operator-rh 2
  source: redhat-operators 3
  sourceNamespace: openshift-marketplace 4
```

   1. Operator のサブスクライブ元のチャネル名を指定します。
   2. サブスクライブする Operator の名前。
   3. Operator を提供する CatalogSource の名前。
   4. CatalogSource の namespace。デフォルトの OperatorHub CatalogSource には **openshift-marketplace** を使用します。

2. Subscription オブジェクトを作成します。

```
$ oc apply -f sub.yaml
```

   Red Hat OpenShift Pipelines Operator がデフォルトのターゲット namespace **openshift-operators** にインストールされるようになりました。

### 3.3.3. 制限された環境での Red Hat OpenShift Pipelines Operator

Red Hat OpenShift Pipelines Operator は、ネットワークが制限された環境でのパイプラインのインストールに対してサポートを有効にします。

Operator は、**cluster** プロキシーオブジェクトに基づいて tekton-controllers によって作成される Pod のコンテナーにプロキシー環境変数を設定するプロキシー Webhook をインストールします。また、プロキシー環境変数を **TektonPipelines**、**TektonTriggers**、**Controllers**、**Webhooks**、および **Operator Proxy Webhook** リソースに設定します。

デフォルトで、プロキシー Webhook は **openshift-pipelines** namespace について無効にされます。他の namespace に対してこれを無効にするには、**operator.tekton.dev/disable-proxy: true** ラベルを **namespace** オブジェクトに追加します。

### 3.3.4. 関連情報

- Operator の OpenShift Container Platform へのインストール方法については、Operator のクラスターへの追加 セクションを参照してください。
- 制限された環境でパイプラインを使用する方法についての詳細は、以下を参照してください。
  - 制限された環境でパイプラインを実行するためのイメージのミラーリング
  - 制限されたクラスターの Samples Operator の設定
  - ミラーリングされたレジストリーでのクラスターの作成

## 3.4. OPENSHIFT PIPELINES のアンインストール

Red Hat OpenShift Pipelines Operator のアンインストールは 2 つの手順で実行されます。

1. Red Hat OpenShift Pipelines Operator のインストール時にデフォルトで追加されたカスタムリソース (CR) を削除します。
2. Red Hat OpenShift Pipelines Operator をアンインストールします。

Operator のみをアンインストールしても、Operator のインストール時にデフォルトで作成される Red Hat OpenShift Pipelines コンポーネントは削除されません。

### 3.4.1. Red Hat OpenShift Pipelines コンポーネントおよびカスタムリソースの削除

Red Hat OpenShift Pipelines Operator のインストール時にデフォルトで作成されるカスタムリソース (CR) を削除します。

**手順**

1. Web コンソールの **Administrator** パースペクティブで、**Administration → Custom Resource Definition** に移動します。
2. **Filter by name** ボックスに **config.operator.tekton.dev** を入力し、 Red Hat OpenShift Pipelines Operator CR を検索します。
3. **CRD Config** をクリックし、 **Custom Resource Definition Details**ページを表示します。
4. **Actions** ドロップダウンメニューをクリックし、**Delete Custom Resource Definition**を選択します。

**注記**

CR を削除すると Red Hat OpenShift Pipelines コンポーネントが削除され、クラスター上のすべての Task および Pipeline が失われます。

5. **Delete** をクリックし、CR の削除を確認します。

### 3.4.2. Red Hat OpenShift Pipelines Operator のアンインストール

**手順**

1. **Operators → OperatorHub** ページから、**Filter by keyword** ボックスを使用して **Red Hat OpenShift Pipelines Operator** を検索します。

2. **OpenShift Pipelines Operator** タイルをクリックします。Operator タイルはこれがインストールされていることを示します。

3. **OpenShift Pipelines Operator** 記述子ページで、**Uninstall** をクリックします。

**関連情報**

- Operator の OpenShift Container Platform でのアンインストール方法については、クラスターからの Operator の削除 セクションを参照してください。

## 3.5. OPENSHIFT PIPELINES を使用したアプリケーションの CI/CD ソリューションの作成

Red Hat OpenShift Pipelines を使用すると、カスタマイズされた CI/CD ソリューションを作成して、アプリケーションをビルドし、テストし、デプロイできます。

アプリケーション向けの本格的なセルフサービス型の CI/CD パイプラインを作成するには、以下のタスクを実行する必要があります。

- カスタムタスクを作成するか、または既存の再利用可能なタスクをインストールします。
- アプリケーションの配信パイプラインを作成し、定義します。
- 以下の方法のいずれかを使用して、パイプライン実行のためにワークスペースに接続されているストレージボリュームまたはファイルシステムを提供します。
  - 永続ボリューム要求 (PVC) を作成するボリューム要求テンプレートを指定します。
  - 永続ボリューム要求 (PVC) を指定します。
- **PipelineRun** オブジェクトを作成し、Pipeline をインスタンス化し、これを起動します。
- トリガーを追加し、ソースリポジトリーのイベントを取得します。

このセクションでは、**pipelines-tutorial** の例を使用して前述のタスクについて説明します。この例では、以下で設定される単純なアプリケーションを使用します。

- **pipelines-vote-ui** Git リポジトリーにソースコードがあるフロントエンドインターフェイス (**pipelines-vote-ui**)。
- **pipelines-vote-api** Git リポジトリーにソースコードがあるバックエンドインターフェイス (**pipelines-vote-api**)。
- **pipelines-tutorial** Git リポジトリーにある **apply-manifests** および **update-deployment** タスク。

### 3.5.1. 前提条件

- OpenShift Container Platform クラスターにアクセスできる。
- OpenShift OperatorHub に一覧表示されている Red Hat OpenShift Pipelines Operator を使用して OpenShift Pipelines をインストールしている。インストールが完了すると、クラスター全体に適用可能になります。
- OpenShift Pipelines CLI をインストールしている。

- GitHub ID を使用してフロントエンドの **ui-repo** およびバックエンドの **api-repo** Git リポジトリーをフォークしており、これらのリポジトリーに管理者権限でアクセスできる。
- オプション: **pipelines-tutorial** Git リポジトリーのクローンを作成している。

### 3.5.2. プロジェクトの作成およびパイプラインのサービスアカウントの確認

**手順**

1. OpenShift Container Platform クラスターにログインします。

```
$ oc login -u <login> -p <password> https://openshift.example.com:6443
```

2. サンプルアプリケーションのプロジェクトを作成します。このサンプルワークフローでは、**pipelines-tutorial** プロジェクトを作成します。

```
$ oc new-project pipelines-tutorial
```

**注記**

別の名前でプロジェクトを作成する場合は、サンプルで使用されているリソース URL をプロジェクト名で更新してください。

3. **pipeline** サービスアカウントを表示します。
Red Hat OpenShift Pipelines Operator は、イメージのビルドおよびプッシュを実行するのに十分なパーミッションを持つ **pipeline** という名前のサービスアカウントを追加し、設定します。このサービスアカウントは **PipelineRun** オブジェクトによって使用されます。

```
$ oc get serviceaccount pipeline
```

### 3.5.3. パイプラインタスクの作成

**手順**

1. **pipelines-tutorial** リポジトリーから **apply-manifests** および **update-deployment** タスクリソースをインストールします。これには、パイプラインの再利用可能なタスクの一覧が含まれます。

```
$ oc create -f https://raw.githubusercontent.com/openshift/pipelines-tutorial/pipelines-1.4/01_pipeline/01_apply_manifest_task.yaml
$ oc create -f https://raw.githubusercontent.com/openshift/pipelines-tutorial/pipelines-1.4/01_pipeline/02_update_deployment_task.yaml
```

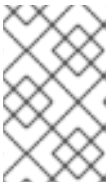
2. **tkn task list** コマンドを使用して、作成したタスクを一覧表示します。

```
$ tkn task list
```

出力では、**apply-manifests** および **update-deployment** タスクリソースが作成されていることを検証します。

```
NAME              DESCRIPTION   AGE
apply-manifests                 1 minute ago
update-deployment               48 seconds ago
```

3. **tkn clustertasks list** コマンドを使用して、**buildah** および **s2i-python-3** などの Operator でインストールされた追加のクラスタータスクを一覧表示します。



**注記**

制限された環境で **buildah** クラスタータスクを使用するには、Dockerfile が内部イメージストリームをベースイメージとして使用していることを確認する必要があります。

```
$ tkn clustertasks list
```

出力には、Operator でインストールされた **ClusterTask** リソースが一覧表示されます。

```
NAME                DESCRIPTION   AGE
buildah                           1 day ago
git-clone                         1 day ago
s2i-python                        1 day ago
tkn                               1 day ago
```

### 3.5.4. パイプラインのアセンブル

パイプラインは CI/CD フローを表し、実行するタスクによって定義されます。これは、複数のアプリケーションや環境で汎用的かつ再利用可能になるように設計されています。

パイプラインは、**from** および **runAfter** パラメーターを使用してタスクが相互に対話する方法および実行順序を指定します。これは **workspaces** フィールドを使用して、パイプラインの各タスクの実行中に必要な 1 つ以上のボリュームを指定します。

このセクションでは、GitHub からアプリケーションのソースコードを取り、これを OpenShift Container Platform にビルドし、デプロイするパイプラインを作成します。

パイプラインは、バックエンドアプリケーションの **vote-api** およびフロントエンドアプリケーション **vote-ui** について以下のタスクを実行します。

- **git-url** および **git-revision** パラメーターを参照して、Git リポジトリーからアプリケーションのソースコードのクローンを作成します。
- **buildah** クラスタータスクを使用してコンテナーイメージをビルドします。
- **image** パラメーターを参照してイメージを内部イメージレジストリーにプッシュします。
- **apply-manifests** および **update-deployment** タスクを使用して新規イメージを OpenShift Container Platform にデプロイします。

**手順**

1. 以下のサンプルのパイプライン YAML ファイルの内容をコピーし、保存します。

```
apiVersion: tekton.dev/v1beta1
```

```
kind: Pipeline
metadata:
  name: build-and-deploy
spec:
  workspaces:
  - name: shared-workspace
  params:
  - name: deployment-name
    type: string
    description: name of the deployment to be patched
  - name: git-url
    type: string
    description: url of the git repo for the code of deployment
  - name: git-revision
    type: string
    description: revision to be used from repo of the code for deployment
    default: "pipelines-1.4"
  - name: IMAGE
    type: string
    description: image to be built from the code
  tasks:
  - name: fetch-repository
    taskRef:
      name: git-clone
      kind: ClusterTask
    workspaces:
    - name: output
      workspace: shared-workspace
    params:
    - name: url
      value: $(params.git-url)
    - name: subdirectory
      value: ""
    - name: deleteExisting
      value: "true"
    - name: revision
      value: $(params.git-revision)
  - name: build-image
    taskRef:
      name: buildah
      kind: ClusterTask
    params:
    - name: IMAGE
      value: $(params.IMAGE)
    workspaces:
    - name: source
      workspace: shared-workspace
    runAfter:
    - fetch-repository
  - name: apply-manifests
    taskRef:
      name: apply-manifests
    workspaces:
    - name: source
      workspace: shared-workspace
    runAfter:
```

```
    - build-image
  - name: update-deployment
    taskRef:
      name: update-deployment
    params:
    - name: deployment
      value: $(params.deployment-name)
    - name: IMAGE
      value: $(params.IMAGE)
    runAfter:
    - apply-manifests
```

パイプライン定義は、Git ソースリポジトリーおよびイメージレジストリーの詳細を抽象化します。これらの詳細は、パイプラインのトリガーおよび実行時に **params** として追加されます。

2. パイプラインを作成します。

```
$ oc create -f <pipeline-yaml-file-name.yaml>
```

または、Git リポジトリーから YAML ファイルを直接実行することもできます。

```
$ oc create -f https://raw.githubusercontent.com/openshift/pipelines-tutorial/pipelines-1.4/01_pipeline/04_pipeline.yaml
```

3. **tkn pipeline list** コマンドを使用して、パイプラインがアプリケーションに追加されていることを確認します。

```
$ tkn pipeline list
```

この出力では、**build-and-deploy** パイプラインが作成されていることを検証します。

```
NAME              AGE            LAST RUN   STARTED   DURATION   STATUS
build-and-deploy  1 minute ago   ---        ---       ---        ---
```

### 3.5.5. 制限された環境でパイプラインを実行するためのイメージのミラーリング

OpenShift Pipelines を非接続のクラスターまたは制限された環境でプロビジョニングされたクラスターで実行するには、制限されたネットワークに Samples Operator が設定されているか、またはクラスター管理者がミラーリングされたレジストリーでクラスターを作成しているか確認する必要があります。

以下の手順では、**pipelines-tutorial** の例を使用して、ミラーリングされたレジストリーを持つクラスターを使用して、制限された環境でアプリケーションのパイプラインを作成します。**pipelines-tutorial** の例が制限された環境で機能することを確認するには、フロントエンドインターフェイス (**pipelines-vote-ui**)、バックエンドインターフェイス (**pipelines-vote-api**) および **cli** のミラーレジストリーからそれぞれのビルダーイメージをミラーリングする必要があります。

**手順**

1. フロントエンドインターフェイス (**pipelines-vote-ui**) のミラーレジストリーからビルダーイメージをミラーリングします。

   a. 必要なイメージタグがインポートされていないことを確認します。

```
$ oc describe imagestream python -n openshift
```

**出力例**

```
Name:   python
Namespace:  openshift
[...]

3.8-ubi8 (latest)
  tagged from registry.redhat.io/ubi8/python-38:latest
    prefer registry pullthrough when referencing this tag

  Build and run Python 3.8 applications on UBI 8. For more information about using this
builder image, including OpenShift considerations, see https://github.com/sclorg/s2i-
python-container/blob/master/3.8/README.md.
  Tags: builder, python
  Supports: python:3.8, python
  Example Repo: https://github.com/sclorg/django-ex.git

[...]
```

b. サポートされるイメージタグをプライベートレジストリーに対してミラーリングします。

```
$ oc image mirror registry.redhat.io/ubi8/python-38:latest <mirror-registry>:
<port>/ubi8/python-38
```

c. イメージをインポートします。

```
$ oc tag <mirror-registry>:<port>/ubi8/python-38 python:latest --scheduled -n openshift
```

イメージを定期的に再インポートする必要があります。**--scheduled** フラグは、イメージの自動再インポートを有効にします。

d. 指定されたタグを持つイメージがインポートされていることを確認します。

```
$ oc describe imagestream python -n openshift
```

**出力例**

```
Name:   python
Namespace:  openshift
[...]

latest
  updates automatically from registry <mirror-registry>:<port>/ubi8/python-38

  * <mirror-registry>:<port>/ubi8/python-
38@sha256:3ee3c2e70251e75bfeac25c0c33356add9cc4abcbc9c51d858f39e4dc29c5f58


[...]
```

2. バックエンドインターフェイス (**pipelines-vote-api**) のミラーレジストリーからビルダーイメージをミラーリングします。

   a. 必要なイメージタグがインポートされていないことを確認します。

   ```
   $ oc describe imagestream golang -n openshift
   ```

   **出力例**

   ```
   Name:   golang
   Namespace:  openshift
   [...]

   1.14.7-ubi8 (latest)
     tagged from registry.redhat.io/ubi8/go-toolset:1.14.7
       prefer registry pullthrough when referencing this tag

     Build and run Go applications on UBI 8. For more information about using this builder image, including OpenShift considerations, see https://github.com/sclorg/golang-container/blob/master/README.md.
     Tags: builder, golang, go
     Supports: golang
     Example Repo: https://github.com/sclorg/golang-ex.git

   [...]
   ```

   b. サポートされるイメージタグをプライベートレジストリーに対してミラーリングします。

   ```
   $ oc image mirror registry.redhat.io/ubi8/go-toolset:1.14.7 <mirror-registry>:<port>/ubi8/go-toolset
   ```

   c. イメージをインポートします。

   ```
   $ oc tag <mirror-registry>:<port>/ubi8/go-toolset golang:latest --scheduled -n openshift
   ```

   イメージを定期的に再インポートする必要があります。**--scheduled** フラグは、イメージの自動再インポートを有効にします。

   d. 指定されたタグを持つイメージがインポートされていることを確認します。

   ```
   $ oc describe imagestream golang -n openshift
   ```

   **出力例**

   ```
   Name:   golang
   Namespace:  openshift
   [...]

   latest
     updates automatically from registry <mirror-registry>:<port>/ubi8/go-toolset

     * <mirror-registry>:<port>/ubi8/go-toolset@sha256:59a74d581df3a2bd63ab55f7ac106677694bf612a1fe9e7e3e1487f55c421
   ```

```
b37

[...]
```

3. **cli** のミラーレジストリーからビルダーイメージをミラーリングします。

   a. 必要なイメージタグがインポートされていないことを確認します。

   ```
   $ oc describe imagestream cli -n openshift
   ```

   **出力例**

   ```
   Name:                   cli
   Namespace:              openshift
   [...]

   latest
     updates automatically from registry quay.io/openshift-release-dev/ocp-v4.0-art-
   dev@sha256:65c68e8c22487375c4c6ce6f18ed5485915f2bf612e41fef6d41cbfcdb143551


     * quay.io/openshift-release-dev/ocp-v4.0-art-
   dev@sha256:65c68e8c22487375c4c6ce6f18ed5485915f2bf612e41fef6d41cbfcdb143551


   [...]
   ```

   b. サポートされるイメージタグをプライベートレジストリーに対してミラーリングします。

   ```
   $ oc image mirror quay.io/openshift-release-dev/ocp-v4.0-art-
   dev@sha256:65c68e8c22487375c4c6ce6f18ed5485915f2bf612e41fef6d41cbfcdb143551
   <mirror-registry>:<port>/openshift-release-dev/ocp-v4.0-art-dev:latest
   ```

   c. イメージをインポートします。

   ```
   $ oc tag <mirror-registry>:<port>/openshift-release-dev/ocp-v4.0-art-dev cli:latest --
   scheduled -n openshift
   ```

   イメージを定期的に再インポートする必要があります。**--scheduled** フラグは、イメージの自動再インポートを有効にします。

   d. 指定されたタグを持つイメージがインポートされていることを確認します。

   ```
   $ oc describe imagestream cli -n openshift
   ```

   **出力例**

   ```
   Name:                   cli
   Namespace:              openshift
   [...]

   latest
     updates automatically from registry <mirror-registry>:<port>/openshift-release-dev/ocp-
   v4.0-art-dev
   ```

```
  * <mirror-registry>:<port>/openshift-release-dev/ocp-v4.0-art-
dev@sha256:65c68e8c22487375c4c6ce6f18ed5485915f2bf612e41fef6d41cbfcdb143551


[...]
```

**関連情報**

- 制限されたクラスターの Samples Operator の設定
- ミラーリングされたレジストリーでのクラスターの作成

### 3.5.6. パイプラインの実行

**PipelineRun** リソースはパイプラインを開始し、これを特定の呼び出しに使用する必要のある Git およびイメージリソースに関連付けます。これは、パイプラインの各タスクについて **TaskRun** を自動的に作成し、開始します。

**手順**

1. バックエンドアプリケーションのパイプラインを起動します。

```
$ tkn pipeline start build-and-deploy \
    -w name=shared-
workspace,volumeClaimTemplateFile=https://raw.githubusercontent.com/openshift/pipelines-
tutorial/pipelines-1.4/01_pipeline/03_persistent_volume_claim.yaml \
    -p deployment-name=pipelines-vote-api \
    -p git-url=https://github.com/openshift/pipelines-vote-api.git \
    -p IMAGE=image-registry.openshift-image-registry.svc:5000/pipelines-tutorial/pipelines-
vote-api
```

   直前のコマンドは、パイプライン実行の永続ボリューム要求 (PVC) を作成するボリューム要求テンプレートを使用します。

2. パイプライン実行の進捗を追跡するには、以下のコマンドを入力します。

```
$ tkn pipelinerun logs <pipelinerun_id> -f
```

   上記のコマンドの <pipelinerun_id> は、直前のコマンドの出力で返された **PipelineRun** の ID です。

3. フロントエンドアプリケーションのパイプラインを起動します。

```
$ tkn pipeline start build-and-deploy \
    -w name=shared-
workspace,volumeClaimTemplateFile=https://raw.githubusercontent.com/openshift/pipelines-
tutorial/pipelines-1.4/01_pipeline/03_persistent_volume_claim.yaml \
    -p deployment-name=pipelines-vote-ui \
    -p git-url=https://github.com/openshift/pipelines-vote-ui.git \
    -p IMAGE=image-registry.openshift-image-registry.svc:5000/pipelines-tutorial/pipelines-
vote-ui
```

4. パイプライン実行の進捗を追跡するには、以下のコマンドを入力します。

```
$ tkn pipelinerun logs <pipelinerun_id> -f
```

上記のコマンドの <pipelinerun_id> は、直前のコマンドの出力で返された **PipelineRun** の ID です。

5. 数分後に、**tkn pipelinerun list** コマンドを使用して、すべてのパイプライン実行を一覧表示してパイプラインが正常に実行されたことを確認します。

```
$ tkn pipelinerun list
```

出力には、パイプライン実行が一覧表示されます。

```
NAME                     STARTED      DURATION     STATUS
build-and-deploy-run-xy7rw   1 hour ago   2 minutes    Succeeded
build-and-deploy-run-z2rz8   1 hour ago   19 minutes   Succeeded
```

6. アプリケーションルートを取得します。

```
$ oc get route pipelines-vote-ui --template='http://{{.spec.host}}'
```

上記のコマンドの出力に留意してください。このルートを使用してアプリケーションにアクセスできます。

7. 直前のパイプラインのパイプラインリソースおよびサービスアカウントを使用して最後のパイプライン実行を再実行するには、以下を実行します。

```
$ tkn pipeline start build-and-deploy --last
```

### 3.5.7. トリガーのパイプラインへの追加

トリガーは、パイプラインがプッシュイベントやプル要求などの外部の GitHub イベントに応答できるようにします。アプリケーションのパイプラインをアセンブルし、起動した後に、**TriggerBinding**、**TriggerTemplate**、**Trigger**、および **EventListener** リソースを追加して GitHub イベントを取得します。

**手順**

1. 以下のサンプル **TriggerBinding** YAML ファイルの内容をコピーし、これを保存します。

```
apiVersion: triggers.tekton.dev/v1alpha1
kind: TriggerBinding
metadata:
  name: vote-app
spec:
  params:
  - name: git-repo-url
    value: $(body.repository.url)
  - name: git-repo-name
    value: $(body.repository.name)
  - name: git-revision
    value: $(body.head_commit.id)
```

2. **TriggerBinding** リソースを作成します。

```
$ oc create -f <triggerbinding-yaml-file-name.yaml>
```

または、**TriggerBinding** リソースを **pipelines-tutorial** Git リポジトリーから直接作成できます。

```
$ oc create -f https://raw.githubusercontent.com/openshift/pipelines-tutorial/pipelines-1.4/03_triggers/01_binding.yaml
```

3. 以下のサンプル **TriggerTemplate** YAML ファイルの内容をコピーし、これを保存します。

```
apiVersion: triggers.tekton.dev/v1alpha1
kind: TriggerTemplate
metadata:
  name: vote-app
spec:
  params:
  - name: git-repo-url
    description: The git repository url
  - name: git-revision
    description: The git revision
    default: pipelines-1.4
  - name: git-repo-name
    description: The name of the deployment to be created / patched

  resourcetemplates:
  - apiVersion: tekton.dev/v1beta1
    kind: PipelineRun
    metadata:
      generateName: build-deploy-$(tt.params.git-repo-name)-
    spec:
      serviceAccountName: pipeline
      pipelineRef:
        name: build-and-deploy
      params:
      - name: deployment-name
        value: $(tt.params.git-repo-name)
      - name: git-url
        value: $(tt.params.git-repo-url)
      - name: git-revision
        value: $(tt.params.git-revision)
      - name: IMAGE
        value: image-registry.openshift-image-registry.svc:5000/pipelines-tutorial/$(tt.params.git-repo-name)
      workspaces:
      - name: shared-workspace
        volumeClaimTemplate:
          spec:
            accessModes:
              - ReadWriteOnce
            resources:
              requests:
                storage: 500Mi
```

テンプレートは、ワークスペースのストレージボリュームを定義するために使用され

テンプレートは、ワークスペースのストレージボリュームを定義するための永続ボリューム要求 (PVC) を作成するためのボリューム要求テンプレートを指定します。そのため、データストレージを提供するために永続ボリューム要求 (PVC) を作成する必要はありません。

4. **TriggerTemplate** リソースを作成します。

```
$ oc create -f <triggertemplate-yaml-file-name.yaml>
```

または、**TriggerTemplate** リソースを **pipelines-tutorial** Git リポジトリーから直接作成できます。

```
$ oc create -f https://raw.githubusercontent.com/openshift/pipelines-tutorial/pipelines-1.4/03_triggers/02_template.yaml
```

5. 以下のサンプルの **Trigger** YAML ファイルの内容をコピーし、保存します。

```
apiVersion: triggers.tekton.dev/v1alpha1
kind: Trigger
metadata:
  name: vote-trigger
spec:
  serviceAccountName: pipeline
  bindings:
    - ref: vote-app
  template:
    ref: vote-app
```

6. **Trigger** リソースを作成します。

```
$ oc create -f <trigger-yaml-file-name.yaml>
```

または、**Trigger** リソースを **pipelines-tutorial** Git リポジトリーから直接作成できます。

```
$ oc create -f https://raw.githubusercontent.com/openshift/pipelines-tutorial/pipelines-1.4/03_triggers/03_trigger.yaml
```

7. 以下のサンプル **EventListener** YAML ファイルの内容をコピーし、これを保存します。

```
apiVersion: triggers.tekton.dev/v1alpha1
kind: EventListener
metadata:
  name: vote-app
spec:
  serviceAccountName: pipeline
  triggers:
    - triggerRef: vote-trigger
```

または、トリガーカスタムリソースを定義していない場合は、トリガーの名前を参照する代わりに、バインディングおよびテンプレート仕様を **EventListener** YAML ファイルに追加します。

```
apiVersion: triggers.tekton.dev/v1alpha1
kind: EventListener
metadata:
```

```
  name: vote-app
spec:
  serviceAccountName: pipeline
  triggers:
  - bindings:
    - ref: vote-app
    template:
      ref: vote-app
```

8. 以下のコマンドを実行して **EventListener** リソースを作成します。

- セキュアな HTTPS 接続を使用して **EventListener** リソースを作成するには、以下を実行します。

a. ラベルを追加して、**Eventlistener** リソースへのセキュアな HTTPS 接続を有効にします。

```
$ oc label namespace <ns-name> operator.tekton.dev/enable-annotation=enabled
```

b. **EventListener** リソースを作成します。

```
$ oc create -f <eventlistener-yaml-file-name.yaml>
```

または、**EvenListener** リソースを **pipelines-tutorial** Git リポジトリーから直接作成できます。

```
$ oc create -f https://raw.githubusercontent.com/openshift/pipelines-tutorial/pipelines-1.4/03_triggers/04_event_listener.yaml
```

c. re-encrypt TLS 終端でルートを作成します。

```
$ oc create route reencrypt --service=<svc-name> --cert=tls.crt --key=tls.key --ca-cert=ca.crt --hostname=<hostname>
```

または、re-encrypt TLS 終端 YAML ファイルを作成して、セキュアなルートを作成できます。

**セキュアなルートの re-encrypt TLS 終端 YAML の例**

```
apiVersion: route.openshift.io/v1
kind: Route
metadata:
  name: route-passthrough-secured 1
spec:
  host: <hostname>
  to:
    kind: Service
    name: frontend 2
  tls:
    termination: reencrypt 3
    key: [as in edge termination]
    certificate: [as in edge termination]
    caCertificate: [as in edge termination]
    destinationCACertificate: |- 4
```

```
-----BEGIN CERTIFICATE-----
[...]
-----END CERTIFICATE-----
```

1 2 オブジェクトの名前で、63 文字に制限されます。

3 **termination** フィールドは **reencrypt** に設定されます。これは、必要な唯一の **tls** フィールドです。

4 再暗号化に必要です。**destinationCACertificate** は CA 証明書を指定してエンドポイントの証明書を検証し、ルーターから宛先 Pod への接続のセキュリティーを保護します。サービスがサービス署名証明書を使用する場合または、管理者がデフォルトの CA 証明書をルーターに指定し、サービスにその CA により署名された証明書がある場合には、このフィールドは省略可能です。

他のオプションについては、**oc create route reencrypt --help** を参照してください。

- 非セキュアな HTTP 接続を使用して **EventListener** リソースを作成するには、以下を実行します。

  a. **EventListener** リソースを作成します。

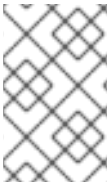
  b. **EventListener** サービスを OpenShift Container Platform ルートとして公開し、これをアクセス可能にします。

```
$ oc expose svc el-vote-app
```

### 3.5.8. Webhook の作成

**Webhook** は、設定されたイベントがリポジトリーで発生するたびにイベントリスナーよって受信される HTTP POST メッセージです。その後、イベントペイロードはトリガーバインディングにマップされ、トリガーテンプレートによって処理されます。トリガーテンプレートは最終的に 1 つ以上のパイプライン実行を開始し、Kubernetes リソースの作成およびデプロイメントを実行します。

このセクションでは、フォークされた Git リポジトリー **pipelines-vote-ui** および **pipelines-vote-api** で Webhook URL を設定します。この URL は、一般に公開されている **EventListener** サービスルートを参照します。



**注記**

Webhook を追加するには、リポジトリーへの管理者権限が必要です。リポジトリーへの管理者アクセスがない場合は、Webhook を追加できるようにシステム管理者に問い合わせてください。

**手順**

1. Webhook URL を取得します。

   - セキュアな HTTPS 接続の場合:

```
$ echo "URL: $(oc  get route el-vote-app --template='https://{{.spec.host}}')"
```

   - HTTP (非セキュアな) 接続の場合:

```
$ echo "URL: $(oc  get route el-vote-app --template='http://{{.spec.host}}')"
```

出力で取得した URL をメモします。

2. フロントエンドリポジトリーで Webhook を手動で設定します。
   a. フロントエンド Git リポジトリー **pipelines-vote-ui** をブラウザーで開きます。
   b. **Settings → Webhooks → Add Webhook** をクリックします。
   c. **Webhooks/Add Webhook** ページで以下を実行します。
      i. 手順 1 の Webhook URL を **Payload URL** フィールドに入力します。
      ii. **Content type** について **application/json** を選択します。
      iii. シークレットを **Secret** フィールドに指定します。
      iv. **Just the push event**が選択されていることを確認します。
      v. **Active** を選択します。
      vi. **Add Webhook** をクリックします。
3. バックエンドリポジトリー **pipelines-vote-api** について手順 2 を繰り返します。

### 3.5.9. パイプライン実行のトリガー

**push** イベントが Git リポジトリーで実行されるたびに、設定された Webhook はイベントペイロードを公開される **EventListener** サービスルートに送信します。アプリケーションの **EventListener** サービスはペイロードを処理し、これを関連する **TriggerBinding** および **TriggerTemplate** リソースのペアに渡します。**TriggerBinding** リソースはパラメーターを抽出し、**TriggerTemplate** リソースはこれらのパラメーターを使用して、リソースの作成方法を指定します。これにより、アプリケーションが再ビルドされ、再デプロイされる可能性があります。

このセクションでは、空のコミットをフロントエンドの **pipelines-vote-ui** リポジトリーにプッシュし、パイプライン実行をトリガーします。

**手順**

1. ターミナルから、フォークした Git リポジトリー **pipelines-vote-ui** のクローンを作成します。

```
$ git clone git@github.com:<your GitHub ID>/pipelines-vote-ui.git -b pipelines-1.4
```

2. 空のコミットをプッシュします。

```
$ git commit -m "empty-commit" --allow-empty && git push origin pipelines-1.4
```

3. パイプライン実行がトリガーされたかどうかを確認します。

```
$ tkn pipelinerun list
```

新規のパイプライン実行が開始されたことに注意してください。

### 3.5.10. 関連情報

- **Developer** パースペクティブのパイプラインについての詳細は、**Developer** パースペクティブでのパイプラインの使用 についてのセクションを参照してください。
- SCC (Security Context Constraints) の詳細は、Managing Security Context Constraints セクションを参照してください。
- 再利用可能なタスクの追加の例については、OpenShift Catalog リポジトリーを参照してください。さらに、Tekton プロジェクトで Tekton Catalog を参照することもできます。
- re-encrypt TLS 終端についての詳細は、 再暗号化終端 について参照してください。
- セキュリティー保護されたルートについての詳細は、セキュリティー保護されたルート についてのセクションを参照してください。

## 3.6. 開発者パースペクティブを使用した RED HAT OPENSHIFT PIPELINES の使用

OpenShift Container Platform Web コンソールの **Developer** パースペクティブを使用して、ソフトウェア配信プロセスの CI/CD パイプラインを作成できます。

**Developer** パースペクティブ:

- **Add → Pipeline → Pipeline Builder** オプションを使用して、アプリケーションのカスタマイズされたパイプラインを作成します。
- **Add → From Git** オプションを使用して、OpenShift Container Platform でアプリケーションを作成する間に Operator によってインストールされたパイプラインテンプレートおよびリソースを使用してパイプラインを作成します。

アプリケーションのパイプラインの作成後に、**Pipelines** ビューでデプロイされたパイプラインを表示し、これらと視覚的に対話できます。**Topology** ビューを使用して、**From Git** オプションを使用して作成されたパイプラインと対話することもできます。**Pipeline Builder** を使用して作成されたパイプラインを **Topology** ビューに表示するには、カスタムラベルをこのパイプラインに適用する必要があります。

### 前提条件

- OpenShift Container Platform クラスターにアクセスでき、**開発者** パースペクティブ に切り替えている。
- クラスターに OpenShift Pipelines Operator がインストール されていること。
- クラスター管理者か、または create および edit パーミッションを持つユーザーであること。
- プロジェクトを作成していること。

### 3.6.1. Pipeline Builder を使用した Pipeline の構築

コンソールの **Developer** パースペクティブで、**+Add → Pipeline → Pipeline Builder** オプションを使用して以下を実行できます。

- **Pipeline ビルダー** または **YAML ビュー** のいずれかを使用してパイプラインを設定します。
- 既存のタスクおよびクラスタータスクを使用して、パイプラインフローを構築します。OpenShift Pipelines Operator をインストールする際に、再利用可能なパイプラインクラスタータスクをクラスターに追加します。

- パイプライン実行に必要なリソースタイプを指定し、必要な場合は追加のパラメーターをパイプラインに追加します。
- パイプラインの各タスクのこれらのパイプラインリソースを入力および出力リソースとして参照します。
- 必要な場合は、タスクのパイプラインに追加されるパラメーターを参照します。タスクのパラメーターは、Task の仕様に基づいて事前に設定されます。
- Operator によってインストールされた、再利用可能なスニペットおよびサンプルを使用して、詳細なパイプラインを作成します。

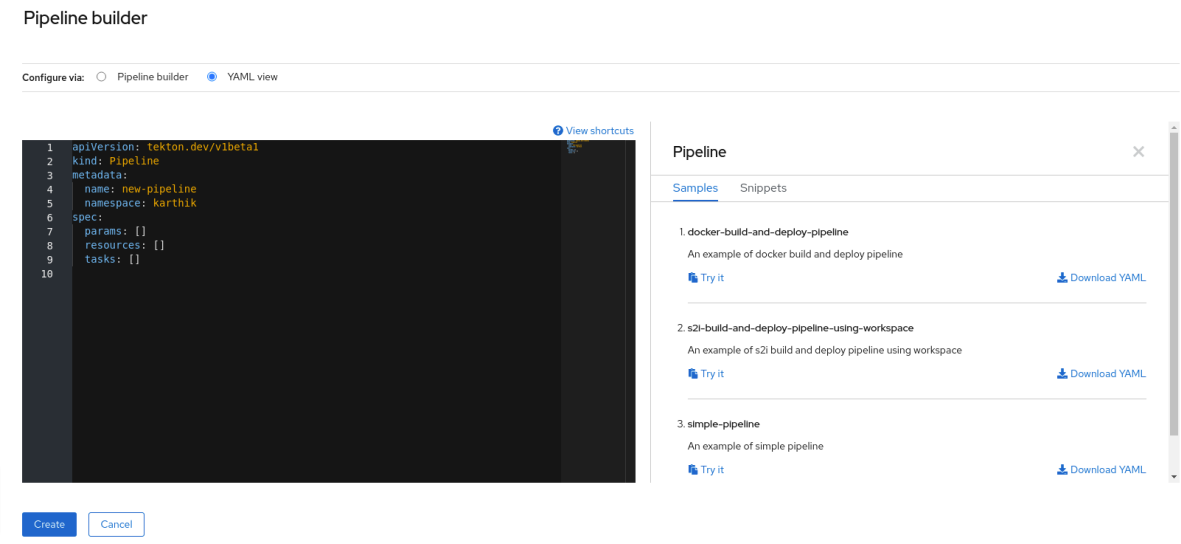
**手順**

1. **Developer** パースペクティブの **+Add** ビューで、**Pipeline** タイルをクリックし、**Pipeline Builder** ページを表示します。
2. **Pipeline ビルダー** ビューまたは **YAML ビュー** のいずれかを使用して、パイプラインを設定します。

**注記**

**Pipeline ビルダー** ビューは、限られた数のフィールドをサポートしますが、**YAML ビュー** は利用可能なすべてのフィールドをサポートします。オプションで、Operator によってインストールされた、再利用可能なスニペットおよびサンプルを使用して、詳細な Pipeline を作成することもできます。

**図3.1 YAML ビュー**




**Pipeline Builder** を使用してパイプラインを設定するには、以下を実行します。

a. パイプラインの一意の名前を入力します。

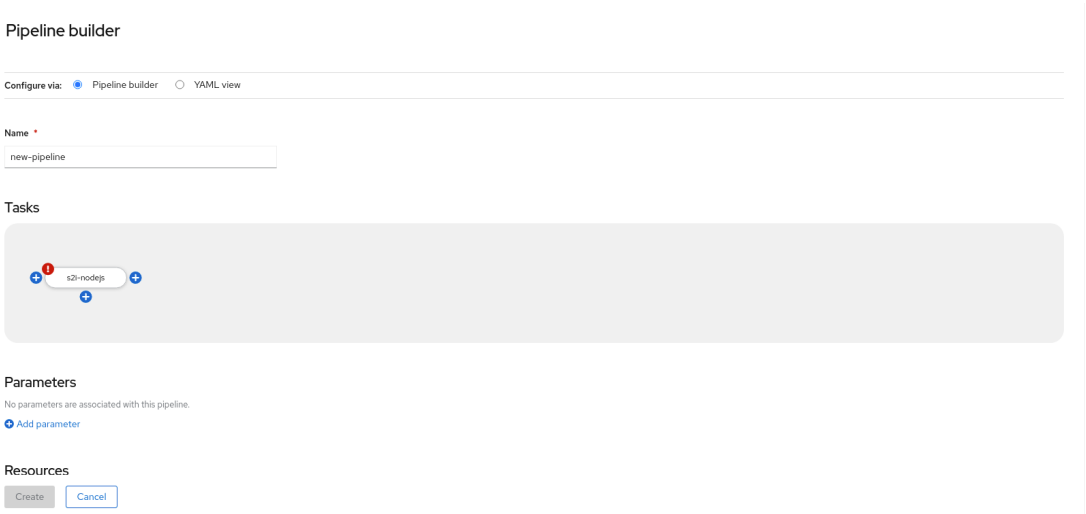
b. **Select Task** 一覧からタスクを選択し、タスクをパイプラインに追加します。この例では、**s2i-nodejs** タスクを使用します。

- 連続するタスクをパイプラインに追加するには、タスクの右側または左側にあるプラスアイコンをクリックし、**Select Task** 一覧から、パイプラインに追加する必要のあるタスクを選択します。この例では、**s2i-nodejs** タスクの右側にあるプラスアイコンを使

用して、**openshift-client** タスクを追加します。

- 並列のタスクを既存のタスクに追加するには、タスクの横に表示されるプラスアイコンをクリックし、**Select Task** 一覧からパイプラインに追加する必要のある並列タスクを選択します。

**図3.2 Pipeline Builder**




c. **Add Resources** をクリックし、パイプライン実行が使用するリソースの名前およびタイプを指定します。これらのリソースは、パイプラインのタスクによって入力および出力として使用されます。この例では、以下のようになります。

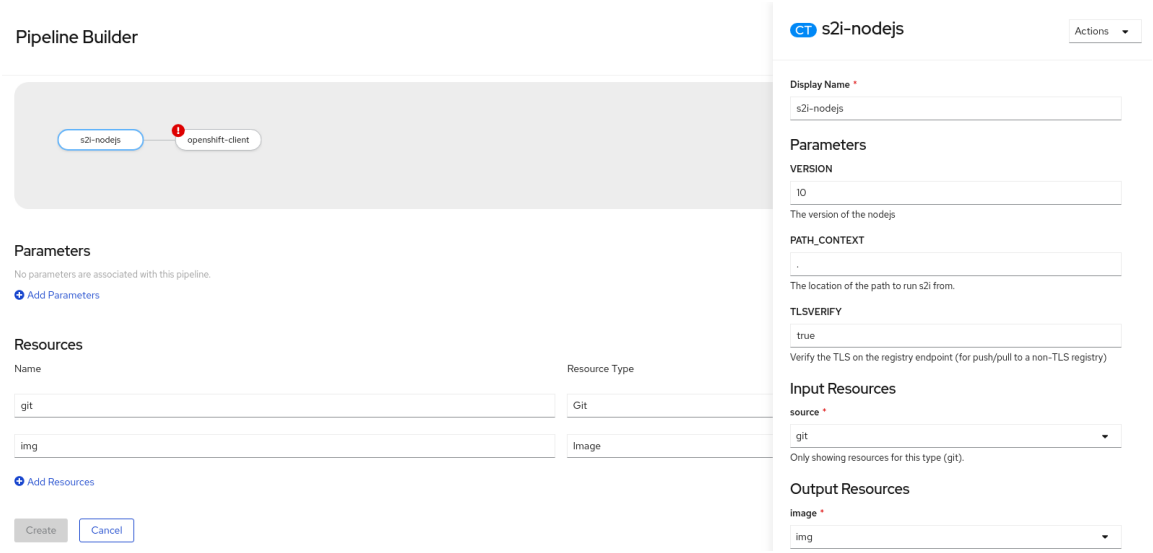
   i. 入力リソースを追加します。**Name** フィールドに **Source** を入力してから、**Resource Type** ドロップダウンリストから **Git** を選択します。

   ii. 出力リソースを追加します。**Name** フィールドに **Img** を入力してから、**Resource Type** ドロップダウンリストから **イメージ** を選択します。

d. オプション: タスクの **パラメーター** は、タスクの仕様に基づいて事前に設定されます。必要な場合は、**Add Parameters** リンクを使用してパラメーターを追加します。

e. タスクのリソースが指定されていない場合、**Missing Resources** (リソース不足) の警告がタスクに表示されます。**s2i-nodejs** タスクをクリックし、タスクの詳細情報が含まれるサイドパネルを表示します。

**図3.3 Pipelines Builder のタスクの詳細**



f. タスクのサイドパネルで、**s2i-nodejs** タスクのリソースおよびパラメーターを指定します。

i. **Input Resources → Source** セクションで、**Select Resources** ドロップダウンリストに、パイプラインに追加したリソースが表示されます。この例では、**Source** を選択します。

ii. **Output Resources → Image** セクションで **Select Resources** リストをクリックし、**Img** を選択します。

iii. 必要な場合は、**$(params.<param-name>)** 構文を使用して、**Parameters** セクションでデフォルトのパラメーターに他のパラメーターを追加します。

iv. 同様に、**openshift-client** タスクの入力リソースを追加します。

3. **Create** をクリックし、**Pipeline Details** ページでパイプラインを作成し、表示します。

4. **Actions** ドロップダウンメニューをクリックしてから **Start** をクリックし、Pipeline を起動します。

### 3.6.2. OpenShift Pipelines を使用したアプリケーションの作成

アプリケーションと共にパイプラインを作成するには、**Developer** パースペクティブの **Add** ビューで **From Git** オプションを使用します。詳細は、Developer パースペクティブを使用したアプリケーションの作成 を参照してください。

### 3.6.3. Developer パースペクティブを使用したパイプラインの使用

**Developer** パースペクティブの **Pipelines** ビューは、以下の詳細と共にプロジェクトのすべてのパイプラインを一覧表示します。

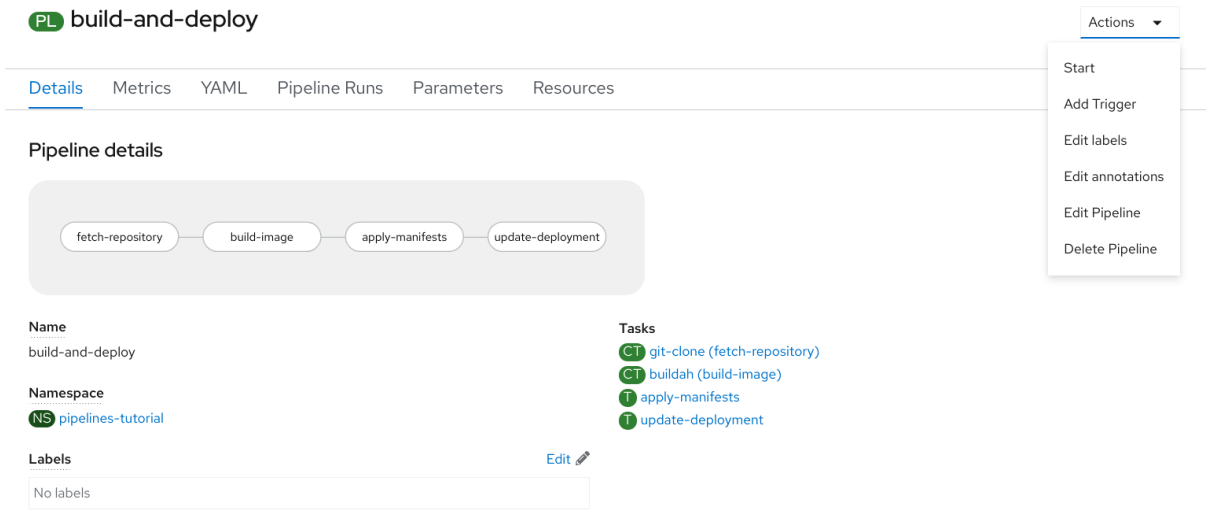
- パイプラインが作成された namespace
- 最後のパイプライン実行
- パイプライン実行のタスクのステータス
- パイプライン実行のステータス
- 最後のパイプライン実行の作成時間

**手順**

1. **Developer** パースペクティブの **Pipelines** ビューで、**Project** ドロップダウンリストからプロジェクトを選択し、そのプロジェクトのパイプラインを表示します。

2. 必要なパイプラインをクリックし、**Pipeline Details** ページを表示します。デフォルトで、**Details** タブが開き、パイプラインのすべての直列および並列タスクが視覚的に表示されます。タスクはページの右下にも一覧表示されます。一覧表示されている **Tasks** をクリックし、タスクの詳細を表示できます。

図3.4 Pipeline の詳細




3. オプションで、**Pipeline details** ページで以下を実行します。

- **Metrics** タブをクリックして、パイプラインについての以下の情報を表示します。
  - **Pipeline 成功比率**
  - **Pipeline Run の数**
  - **Pipeline Run の期間**
  - **Task Run Balancing**
    この情報を使用して、パイプラインのワークフローを改善し、パイプラインのライフサイクルの初期段階で問題をなくすことができます。
- **YAML** タブをクリックし、パイプラインの YAML ファイルを編集します。
- **Pipeline Runs** タブをクリックして、パイプラインの完了済み、実行中、または失敗した実行を確認します。

**注記**

**Pipeline Run Details** ページの **Details** セクションには、失敗したパイプライン実行の **Log Snippet** (ログスニペット) が表示されます。**Log Snippet** (ログスニペット) は、一般的なエラーメッセージとログのスニペットを提供します。**Logs** セクションへのリンクでは、失敗した実行に関する詳細へのクイックアクセスを提供します。**Log Snippet** は、**Task Run Details** ページの **Details** セクションにも表示されます。

Options メニュー  を使用して、実行中のパイプラインを停止するか、以前のパイプライン実行と同じパラメーターとリソースを使用してパイプラインを再実行するか、またはパイプライン実行を削除します。

- **Parameters** タブをクリックして、パイプラインに定義されるパラメーターを表示します。必要に応じて追加のパラメーターを追加するか、または編集することもできます。
- **Resources** タブをクリックして、パイプラインで定義されたリソースを表示します。必要に応じて追加のリソースを追加するか、または編集することもできます。

### 3.6.4. パイプラインの起動

パイプラインの作成後に、これを開始し、これに含まれるタスクを定義されたシーケンスで実行できるようにする必要があります。パイプラインを Pipelines ビュー、Pipeline Details ページ、または Topology ビューから開始できます。

#### 手順

Pipelines ビューを使用してパイプラインを開始するには、以下を実行します。

1. Developer パースペクティブの Pipelines ビューで、パイプラインに隣接する Options メニューで、Start を選択します。
2. Start Pipeline ダイアログボックスは、パイプライン定義に基づいて Git Resources および Image Resources を表示します。

**注記**

From Git オプションを使用して作成されるパイプラインの場合、Start Pipeline ダイアログボックスでは Parameters セクションに **APP_NAME** フィールドも表示され、ダイアログボックスのすべてのフィールドがパイプラインテンプレートによって事前に入力されます。

   a. namespace にリソースがある場合、Git Resources および Image Resources フィールドがそれらのリソースで事前に設定されます。必要な場合は、ドロップダウンを使用して必要なリソースを選択または作成し、Pipeline Run インスタンスをカスタマイズします。

3. オプション: Advanced Options を変更し、認証情報を追加して、指定されたプライベート Git サーバーまたはイメージレジストリーを認証します。

   a. Advanced Options で Show Credentials Options をクリックし、Add Secret を選択します。

   b. Create Source Secret セクションで、以下を指定します。

      i. シークレットの一意の **シークレット名**。

      ii. Designated provider to be authenticated セクションで、Access to フィールドで認証されるプロバイダー、およびベース Server URL を指定します。

      iii. Authentication Type を選択し、認証情報を指定します。

      - Authentication Type **Image Registry Credentials** については、認証する Registry Server Address を指定し、Username、Password、および Email フィールドに認証情報を指定します。
        追加の Registry Server Address を指定する必要がある場合は、Add Credentials を選択します。
      - Authentication Type **Basic Authentication** については、UserName および Password or Token フィールドの値を指定します。
      - Authentication Type **SSH Keys** については、SSH Private Key フィールドの値を指定します。

      iv. シークレットを追加するためにチェックマークを選択します。

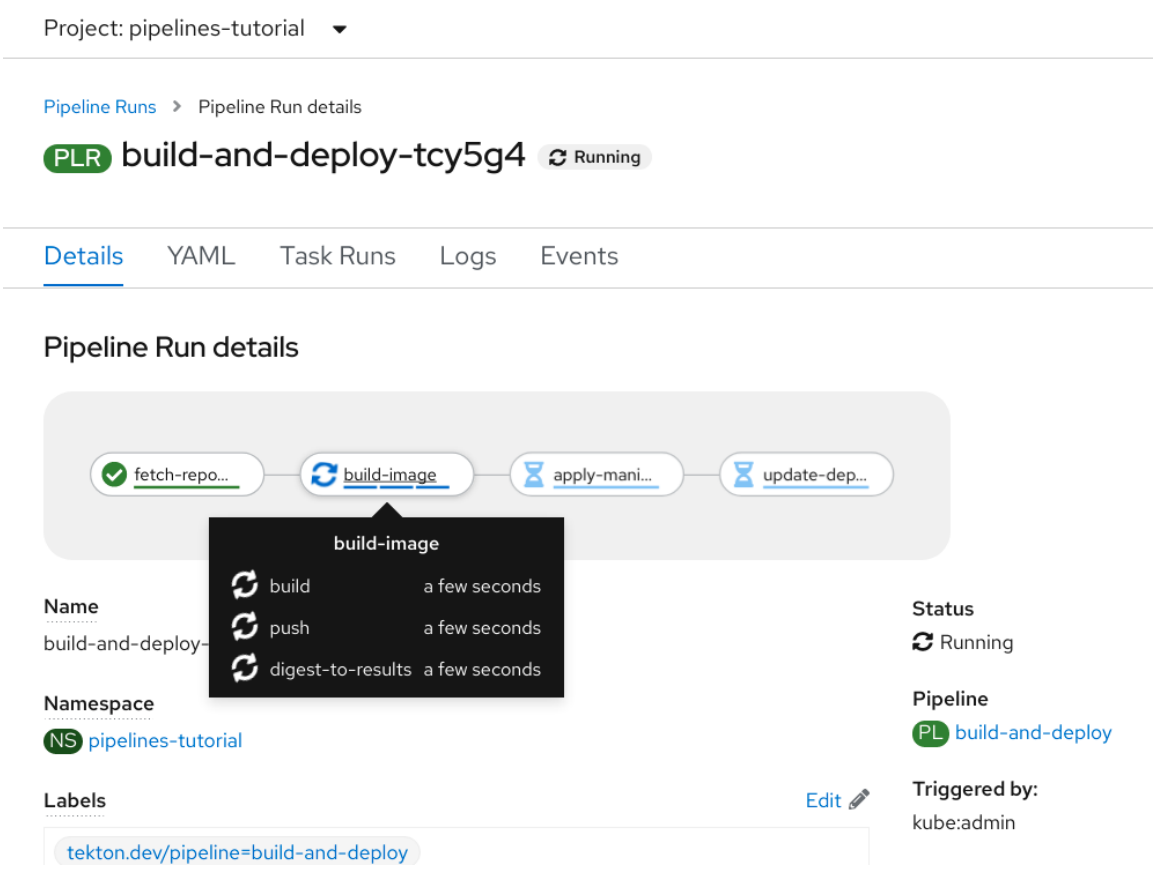
パイプラインのリソースの数に基づいて、複数のシークレットを追加できます。

4. **Start** をクリックしてパイプラインを開始します。

5. **Pipeline Run Details** ページには、実行されるパイプラインが表示されます。パイプラインが開始すると、タスクおよび各タスク内のステップが実行されます。以下を実行することができます。

   - 各ステップの実行にかかった時間を表示するには、タスクにカーソルを合わせます。
   - タスクをクリックし、タスクの各ステップのログを表示します。
   - **Logs** タブをクリックして、タスクの実行シーケンスに関連するログを表示します。該当するボタンを使用して、ペインを展開し、ログを個別に、または一括してダウンロードすることもできます。
   - **Events** タブをクリックして、パイプライン実行で生成されるイベントのストリームを表示します。
     **Task Runs**、**Logs**、および **Events** タブを使用すると、失敗したパイプラインの実行またはタスクの実行のデバッグに役立ちます。

   **図3.5 パイプライン実行の詳細**




6. **From Git** オプションを使用して作成されるパイプラインの場合、**Topology** ビューを使用して、開始後のパイプラインと対話することができます。

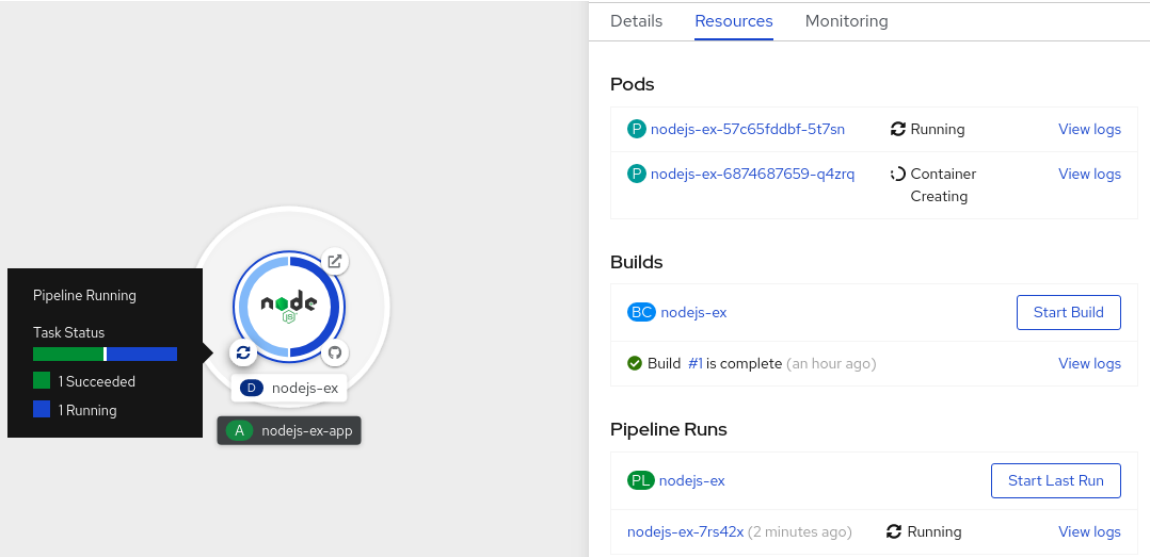
**注記**

**Topology** ビューで **Pipeline Builder** を使用して作成されるパイプラインを表示するには、パイプラインのラベルをカスタマイズし、パイプラインをアプリケーションのワークロードにリンクします。

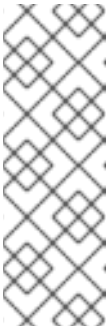
a. 左側のナビゲーションパネルで **Topology** をクリックし、アプリケーションをクリックしてサイドパネルでパイプラインの実行を表示します。

b. **Pipeline Runs** セクションで、**Start Last Run** をクリックし、直前のパラメーターおよびリソースと同じパラメーターおよびリソースを使用して新規パイプライン実行を開始します。このオプションは、パイプライン実行が開始されていない場合は無効になります。

**図3.6 Topology ビューのパイプライン**




c. **Topology** ページで、アプリケーションの左側にカーソルを合わせ、アプリケーションのパイプライン実行のステータスを確認します。



**注記**

**Topology** ページのアプリケーションノードのサイドパネルには、特定のタスクの実行時にパイプライン実行が失敗する場合に **Log Snippet** (ログスニペット) が表示されます。**Resources** タブの **Pipeline Runs** セクションに **Log Snippet** を表示できます。**Log Snippet** (ログスニペット) は、一般的なエラーメッセージとログのスニペットを提供します。**Logs** セクションへのリンクでは、失敗した実行に関する詳細へのクイックアクセスを提供します。

## 3.6.5. Pipeline の編集

Web コンソールの **Developer** パースペクティブを使用して、クラスターの Pipeline を編集できます。

**手順**

1. **Developer** パースペクティブの **Pipelines** ビューで、編集する必要のある Pipeline を選択し、Pipeline の詳細を表示します。**Pipeline Details** ページで **Actions** をクリックし、**Edit Pipeline** を選択します。

2. **Pipeline Builder** ページで以下を行います。

   - 追加の Task、パラメーター、またはリソースを Pipeline に追加できます。
   - 変更する必要のある Task をクリックし、サイドパネルで Task の詳細を表示し、表示名、パラメーターおよびリソースなどの必要な Task の詳細を変更できます。

- または、Task を削除するには、Task をクリックし、サイドパネルで **Actions** をクリックし、**Remove Task** を選択します。

3. **Save** をクリックして変更された Pipeline を保存します。

### 3.6.6. Pipeline の削除

Web コンソールの **Developer** パースペクティブを使用して、クラスターの Pipeline を削除できます。

**手順**

1. **Developer** パースペクティブの **Pipelines** ビューで、Pipeline に隣接する **Options**  をクリックし、**Delete Pipeline** を選択します。

2. **Delete Pipeline** 確認プロンプトで、**Delete** をクリックし、削除を確認します。

## 3.7. パイプラインのリソース消費の削減

マルチテナント環境でクラスターを使用する場合、各プロジェクトおよび Kubernetes オブジェクトの CPU、メモリー、およびストレージリソースの使用を制御する必要があります。これにより、1 つのアプリケーションがリソースを過剰に消費し、他のアプリケーションに影響を与えるのを防ぐことができます。

結果として作成される Pod に設定される最終的なリソース制限を定義するために、Red Hat OpenShift Pipelines は、それらが実行されるプロジェクトのリソースクォータの制限および制限範囲を使用します。

プロジェクトのリソース消費を制限するには、以下を実行できます。

- リソースクォータを設定し、管理 して、リソースの総消費量を制限します。
- 制限範囲を使用し、リソース消費を制限 します。この対象は、Pod、イメージ、イメージストリームおよび永続ボリューム要求 (PVC) などの特定のオブジェクトのリソース消費です。

### 3.7.1. パイプラインでのリソース消費について

各タスクは、**Task** リソースの **steps** フィールドで定義される特定の順序で実行される必要な多数の必要なステップで設定されます。各タスクは Pod として実行され、各ステップは同じ Pod 内のコンテナーとして実行されます。

ステップは一度に 1 つずつ実行されます。タスクを実行する Pod は、一度にタスクの単一コンテナーイメージ (ステップ) を実行するのに十分なリソースのみを要求するため、タスクのすべてのステップについてのリソースを保存しません。

**steps** 仕様の **Resources** フィールドは、リソース消費の制限を指定します。デフォルトで、CPU、メモリー、および一時ストレージのリソース要求は、**BestEffort** (ゼロ) 値またはそのプロジェクトの制限範囲で設定される最小値に設定されます。

**ステップのリソース要求および制限の設定例**

```
spec:
  steps:
  - name: <step_name>
```

```
    resources:
      requests:
        memory: 2Gi
        cpu: 600m
      limits:
        memory: 4Gi
        cpu: 900m
```

**LimitRange** パラメーターおよびコンテナーリソース要求の最小値がパイプラインおよびタスクが実行されるプロジェクトに指定される場合、Red Hat OpenShift Pipelines はプロジェクトのすべての **LimitRange** 値を確認し、ゼロではなく最小値を使用します。

**プロジェクトレベルでの制限範囲パラメーターの設定例**

```
apiVersion: v1
kind: LimitRange
metadata:
  name: <limit_container_resource>
spec:
  limits:
  - max:
      cpu: "600m"
      memory: "2Gi"
    min:
      cpu: "200m"
      memory: "100Mi"
    default:
      cpu: "500m"
      memory: "800Mi"
    defaultRequest:
      cpu: "100m"
      memory: "100Mi"
    type: Container
...
```

### 3.7.2. パイプラインでの追加のリソース消費を軽減する

Pod 内のコンテナーにリソース制限を設定する場合、OpenShift Container Platform はすべてのコンテナーが同時に実行される際に要求されるリソース制限を合計します。

呼び出されるタスクで一度に 1 つのステップを実行するために必要なリソースの最小量を消費するために、Red Hat OpenShift Pipelines は、最も多くのリソースを必要とするステップで指定される CPU、メモリー、および一時ストレージの最大値を要求します。これにより、すべてのステップのリソース要件が満たされます。最大値以外の要求はゼロに設定されます。

ただしこの動作により、リソースの使用率が必要以上に高くなる可能性があります。リソースクォータを使用する場合、これにより Pod がスケジュールできなくなる可能性があります。

たとえば、スクリプトを使用する 2 つのステップを含むタスクと、リソース制限および要求を定義しないタスクについて考えてみましょう。作成される Pod には 2 つの init コンテナー (エントリーポイントコピー用に 1 つとスクリプトの作成用に 1 つ) と 2 つのコンテナー (各ステップに 1 つ) があります。

OpenShift Container Platform はプロジェクトに設定された制限範囲を使用して、必要なリソース要求および制限を計算します。この例では、プロジェクトに以下の制限範囲を設定します。

```
apiVersion: v1
kind: LimitRange
metadata:
  name: mem-min-max-demo-lr
spec:
  limits:
  - max:
      memory: 1Gi
    min:
      memory: 500Mi
    type: Container
```

このシナリオでは、各 init コンテナーは要求メモリー 1Gi (制限範囲の上限) を使用し、各コンテナーは 500Mi の要求メモリーを使用します。そのため、Pod のメモリー要求の合計は 2Gi になります。

同じ制限範囲が 10 のステップのタスクで使用される場合、最終的なメモリー要求は 5Gi になります。これは、各ステップで実際に必要とされるサイズ (500Mi) よりも大きくなります (それぞれのステップは他のステップの後に実行されるためです)。

そのため、リソースのリソース消費を減らすには、以下を行います。

- スクリプト機能および同じイメージを使用して、複数の異なるステップを 1 つの大きなステップにグループ化し、特定のタスクのステップ数を減らします。これにより、要求される最小リソースを減らすことができます。
- 相互に独立しており、独立して実行できるステップを、単一のタスクではなく、複数のタスクに分散します。これにより、各タスクのステップ数が減り、各タスクの要求が小さくなるため、スケジューラーはリソースが利用可能になるとそれらを実行できます。

### 3.7.3. 関連情報

- リソースクォータ
- 制限範囲によるリソース消費の制限
- リソース要求および制限

## 3.8. 特権付きセキュリティーコンテキストでの POD の使用

OpenShift Pipelines 1.3.x 以降のバージョンのデフォルト設定では、パイプライン実行またはタスク実行から Pod が作成される場合、特権付きセキュリティーコンテキストで Pod を実行できません。このような Pod の場合、デフォルトのサービスアカウントは **pipeline** であり、**pipelines** サービスアカウントに関連付けられた SCC (Security Context Constraint) は **pipelines-scc** になります。**pipelines-scc** SCC は **anyuid** SCC と似ていますが、パイプラインの SCC について YAML ファイルに定義されるように若干の違いがあります。

**SecurityContextConstraints オブジェクトの例**

```
apiVersion: security.openshift.io/v1
kind: SecurityContextConstraints
...
fsGroup:
  type: MustRunAs
...
```

さらに、OpenShift Pipeline の一部として提供される **Buildah** クラスタータスクは、デフォルトのストレージドライバーとして **vfs** を使用します。

### 3.8.1. 特権付きセキュリティーコンテキストを使用したパイプライン実行 Pod およびタスク実行 Pod の実行

#### 手順

**privileged** セキュリティーコンテキストで (パイプライン実行またはタスク実行で作成された) Pod を実行するには、以下の変更を行います。

- 関連するユーザーアカウントまたはサービスアカウントを、明示的な SCC を持つように設定します。以下の方法のいずれかを使用して設定を実行できます。
  - 以下の OpenShift コマンドを実行します。

```
$ oc adm policy add-scc-to-user <scc-name> -z <service-account-name>
```

  - または、**RoleBinding** および **Role** または **ClusterRole** の YAML ファイルを変更します。

**RoleBinding オブジェクトの例**

```
apiVersion: rbac.authorization.k8s.io/v1
kind: RoleBinding
metadata:
  name: service-account-name 1
  namespace: default
roleRef:
  apiGroup: rbac.authorization.k8s.io
  kind: ClusterRole
  name: pipelines-scc-clusterrole 2
subjects:
- kind: ServiceAccount
  name: pipeline
  namespace: default
```

1. 適切なサービスアカウント名に置き換えます。
2. 使用するロールバインディングに基づいて適切なクラスターロールに置き換えます。

**ClusterRole オブジェクトの例**

```
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRole
metadata:
  name: pipelines-scc-clusterrole 1
rules:
- apiGroups:
  - security.openshift.io
  resourceNames:
  - nonroot
  resources:
```

```
- securitycontextconstraints
verbs:
- use
```

1 使用するロールバインディングに基づいて適切なクラスターロールに置き換えます。

**注記**

ベストプラクティスとして、デフォルトの YAML ファイルのコピーを作成し、複製ファイルに変更を加えます。

- **vfs** ストレージドライバーを使用しない場合、タスク実行またはパイプライン実行に関連付けられたサービスアカウントを特権付き SCC を持つように設定し、セキュリティーコンテキストを **privileged: true** に設定します。

### 3.8.2. カスタム SCC およびカスタムサービスアカウントを使用したパイプライン実行およびタスク実行

デフォルトの **pipelines** サービスアカウントに関連付けられた **pipelines-scc** SCC(Security Context Constraints) を使用する場合には、パイプライン実行およびタスク実行 Pod はタイムアウトが生じる可能性があります。これは、デフォルトの **pipelines-scc** SCC で **fsGroup.type** パラメーターが **MustRunAs** に設定されているために発生します。

**注記**

Pod タイムアウトの詳細は、BZ#1995779 を参照してください。

Pod タイムアウトを回避するには、**fsGroup.type** パラメーターを **RunAsAny** に設定してカスタム SCC を作成し、これをカスタムサービスアカウントに関連付けることができます。

**注記**

ベストプラクティスとして、パイプライン実行およびタスク実行にカスタム SCC およびカスタムサービスアカウントを使用します。このアプローチを使用することで、柔軟性が増し、アップグレード時にデフォルト値が変更されても実行が失敗することはありません。

**手順**

1. **fsGroup.type** パラメーターを **RunAsAny** に設定してカスタム SCC を定義します。

**例: カスタム SCC**

```
apiVersion: security.openshift.io/v1
kind: SecurityContextConstraints
metadata:
  annotations:
    kubernetes.io/description: my-scc is a close replica of anyuid scc. pipelines-scc has
fsGroup - RunAsAny.
  name: my-scc
allowHostDirVolumePlugin: false
allowHostIPC: false
```

```
allowHostNetwork: false
allowHostPID: false
allowHostPorts: false
allowPrivilegeEscalation: true
allowPrivilegedContainer: false
allowedCapabilities: null
defaultAddCapabilities: null
fsGroup:
  type: RunAsAny
groups:
- system:cluster-admins
priority: 10
readOnlyRootFilesystem: false
requiredDropCapabilities:
- MKNOD
runAsUser:
  type: RunAsAny
seLinuxContext:
  type: MustRunAs
supplementalGroups:
  type: RunAsAny
volumes:
- configMap
- downwardAPI
- emptyDir
- persistentVolumeClaim
- projected
- secret
```

2. カスタム SCC を作成します。

   **例: my-scc SCC の作成**

   ```
   $ oc create -f my-scc.yaml
   ```

3. カスタムサービスアカウントを作成します。

   **例: fsgroup-runasany サービスアカウントの作成**

   ```
   $ oc create serviceaccount fsgroup-runasany
   ```

4. カスタム SCC をカスタムサービスアカウントに関連付けます。

   **例: my-scc SCC を fsgroup-runasany サービスアカウントに関連付けます。**

   ```
   $ oc adm policy add-scc-to-user my-scc -z fsgroup-runasany
   ```

   特権付きタスクにカスタムサービスアカウントを使用する必要がある場合は、以下のコマンドを実行して **特権付き** SCC をカスタムサービスアカウントに関連付けることができます。

   **例: fsgroup-runasany サービスアカウントを使用した 特権付き SCC の関連付け**

   ```
   $ oc adm policy add-scc-to-user privileged -z fsgroup-runasany
   ```

5. パイプライン実行およびタスク実行でカスタムサービスアカウントを使用します。

**例: fsgroup-runasany カスタムサービスアカウントを使用した Pipeline 実行 YAML**

```
apiVersion: tekton.dev/v1beta1
kind: PipelineRun
metadata:
  name: <pipeline-run-name>
spec:
  pipelineRef:
    name: <pipeline-cluster-task-name>
  serviceAccountName: 'fsgroup-runasany'
```

**例: fsgroup-runasany カスタムサービスアカウントを使用したタスク実行 YAML**

```
apiVersion: tekton.dev/v1beta1
kind: TaskRun
metadata:
  name: <task-run-name>
spec:
  taskRef:
    name: <cluster-task-name>
  serviceAccountName: 'fsgroup-runasany'
```

### 3.8.3. 関連情報

- SCC の管理についての詳細は、Managing security context constraints を参照してください。

## 3.9. OPENSHIFT LOGGING OPERATOR を使用したパイプラインログの表示

パイプライン実行、タスク実行、およびイベントリスナーによって生成されるログは、それぞれの Pod に保存されます。トラブルシューティングおよび監査に関するログを確認し、分析すると便利です。

ただし、Pod を無期限に保持すると、リソースを無駄に消費したり、namespace が不必要に分散されたりする可能性があります。

Pod の依存関係を削除して、パイプラインログを表示するには、OpenShift Elasticsearch Operator および OpenShift Logging Operator を使用できます。これらの Operator は、ログを含む Pod を削除した場合でも、Elasticsearch Kibana スタックを使用してパイプラインログを表示するのに役立ちます。

### 3.9.1. 前提条件

Kibana ダッシュボードでパイプラインログを表示しようとする前に、以下を確認してください。

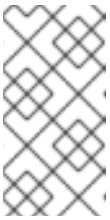
- この手順がクラスター管理者により実行される。
- パイプライン実行およびタスク実行のログが利用可能である。
- OpenShift Elasticsearch Operator および OpenShift Logging Operator がインストールされている。

### 3.9.2. Kibana でのパイプラインログの表示

Kibana Web コンソールでパイプラインログを表示するには、以下を実行します。

**手順**

1. クラスター管理者として OpenShift Container Platform Web コンソールにログインします。
2. メニューバーの右上にある **グリッド** アイコン → **可観測性** → **Logging** をクリックします。Kibana Web コンソールが表示されます。
3. インデックスパターンを作成します。
   a. **Kibana** Web コンソールの左側のナビゲーションパネルで **Management** をクリックします。
   b. **Create index pattern** をクリックします。
   c. **ステップ 1/2: Define index pattern**→ **Index pattern** で、* のパターンを入力して **Next Step** をクリックします。
   d. **ステップ 2/2: Configure settings** → **Time filter field name** で、ドロップダウンメニューから **@timestamp** を選択し、**Create index pattern** をクリックします。
4. フィルターを追加します。
   a. **Kibana** Web コンソールの左側のナビゲーションパネルで **Discover** をクリックします。
   b. **Add a filter +**→ **Edit Query DSL** をクリックします。



**注記**

- 以下のフィルター例の例ごとに、クエリーを編集し、**Save** をクリックします。
- フィルターは順次、適用されます。

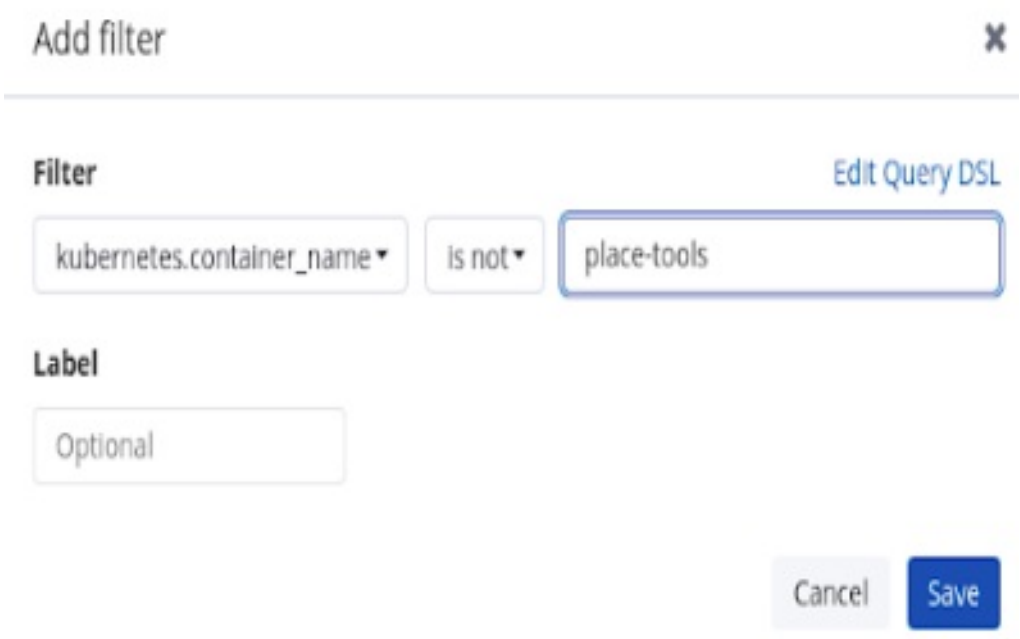
   i. パイプラインに関連するコンテナーをフィルターします。

**パイプラインコンテナーをフィルターするクエリーの例**

```
{
  "query": {
 "match": {
   "kubernetes.flat_labels": {
     "query": "app_kubernetes_io/managed-by=tekton-pipelines",
     "type": "phrase"
   }
 }
  }
}
```

   ii. **place-tools** コンテナーではないすべてのコンテナーをフィルターします。クエリー DSL を編集する代わりに、グラフィカルドロップダウンメニューを使用する例として、以下の方法を考慮してください。

**図3.7 ドロップダウンフィールドを使用したフィルターリングの例**




iii. 強調表示できるように **pipelinerun** をラベルでフィルターします。

**強調表示できるように pipelinerun をラベルでフィルターするクエリーの例**

```
{
  "query": {
 "match": {
   "kubernetes.flat_labels": {
     "query": "tekton_dev/pipelineRun=",
     "type": "phrase"
   }
 }
  }
}
```

iv. 強調表示できるように **pipeline** をラベルでフィルターします。

**強調表示できるように pipeline をラベルでフィルターするクエリーの例**

```
{
  "query": {
 "match": {
   "kubernetes.flat_labels": {
     "query": "tekton_dev/pipeline=",
     "type": "phrase"
   }
 }
  }
}
```

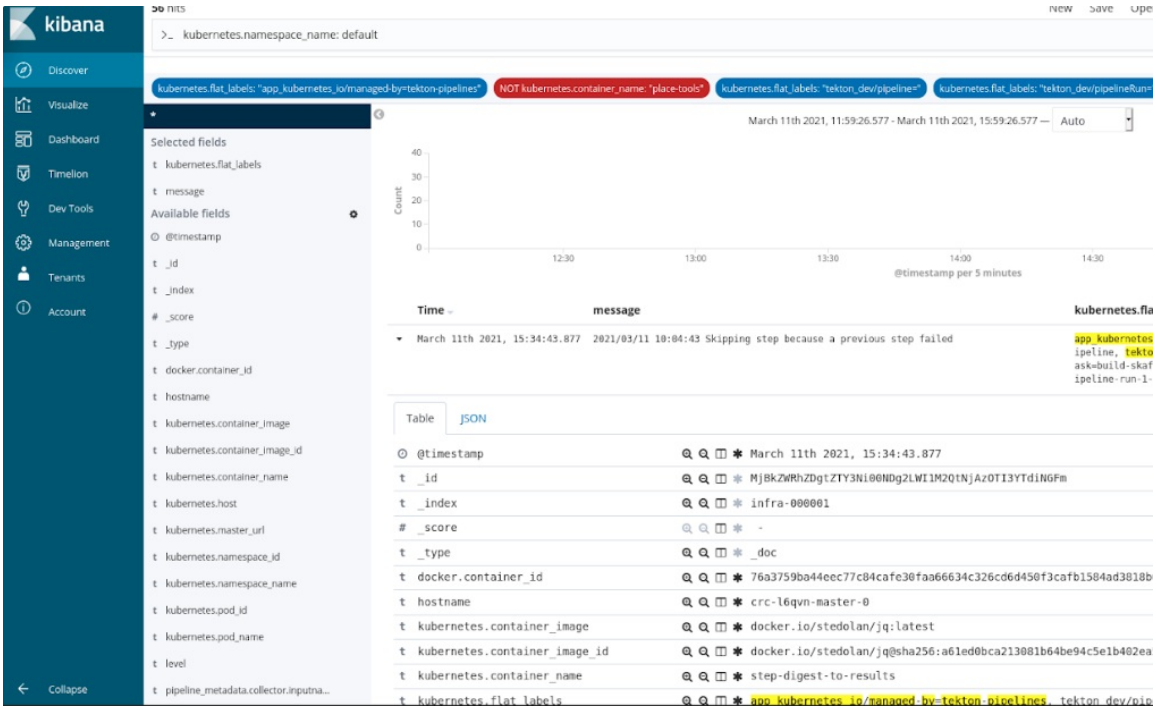
c. **Available fields** リストから以下のフィールドを選択します。

- **kubernetes.flat_labels**

- **message**
  選択したフィールドが **Selected fields** 一覧に表示されていることを確認します。

d. ログは **message** フィールドの下に表示されます。

**図3.8 フィルターされたメッセージ**




## 3.9.3. 関連情報

- OpenShift Logging のインストール
- リソースのログの表示
- Kibana を使用したクラスターログの表示

# 第4章 GITOPS

## 4.1. RED HAT OPENSHIFT GITOPS リリースノート

Red Hat OpenShift GitOps は、クラウドネイティブアプリケーションの継続的デプロイメントを実装するための宣言的な方法です。Red Hat OpenShift GitOps は、異なる環境 (開発、ステージ、実稼働環境など) の異なるクラスターにアプリケーションをデプロイする場合に、アプリケーションの一貫性を確保します。Red Hat OpenShift GitOps は、以下のタスクを自動化する上で役立ちます。

- クラスターに設定、モニターリングおよびストレージについての同様の状態があることの確認。
- クラスターを既知の状態からのリカバリーまたは再作成。
- 複数の OpenShift Container Platform クラスターに対する設定変更を適用するか、またはこれを元に戻す。
- テンプレート化された設定の複数の異なる環境への関連付け。
- ステージから実稼働環境へと、クラスター全体でのアプリケーションのプロモート。

Red Hat OpenShift GitOps の概要については、OpenShift GitOps について を参照してください。

### 4.1.1. 多様性を受け入れるオープンソースの強化

Red Hat では、コード、ドキュメント、Web プロパティーにおける配慮に欠ける用語の置き換えに取り組んでいます。まずは、マスター (master)、スレーブ (slave)、ブラックリスト (blacklist)、ホワイトリスト (whitelist) の 4 つの用語の置き換えから始めます。この取り組みは膨大な作業を要するため、今後の複数のリリースで段階的に用語の置き換えを実施して参ります。詳細は、弊社の CTO、Chris Wright のメッセージ を参照してください。

### 4.1.2. Red Hat OpenShift GitOps 1.2.1 のリリースノート

Red Hat OpenShift GitOps 1.2.1 を OpenShift Container Platform 4.7 および 4.8 でご利用いただけるようになりました。

#### 4.1.2.1. サポート表

現在、今回のリリースに含まれる機能にはテクノロジープレビューのものがあります。これらの実験的機能は、実稼働環境での使用を目的としていません。

テクノロジープレビュー機能のサポート範囲

以下の表では、機能は以下のステータスでマークされています。

- **TP: テクノロジープレビュー**
- **GA: 一般公開機能**

これらの機能に関しては、Red Hat カスタマーポータルの以下のサポート範囲を参照してください。

**表4.1 サポート表**

| 機能 | Red Hat OpenShift GitOps 1.2.1 |
| --- | --- |
| Argo CD | GA |
| Argo CD ApplicationSet | TP |
| Red Hat OpenShift GitOps Application Manager (kam) | TP |

#### 4.1.2.2. 修正された問題

以下の問題は、現在のリリースで解決されています。

- 以前のバージョンでは、起動時にアプリケーションコントローラーでメモリーが大幅に急増していました。アプリケーションコントローラーのフラグ **--kubectl-parallelism-limit** は、デフォルトで 10 に設定されますが、この値は Argo CD CR 仕様に **.spec.controller.kubeParallelismLimit** の数字を指定して上書きできます。GITOPS-1255

- 最新の Triggers APIs により、**kam bootstrap** コマンドの使用時に kustomization.yaml のエントリーが重複していることが原因で、Kubernetes のビルドが失敗しました。この問題に対処するために、Pipelines および Tekton トリガーコンポーネントが v0.24.2 および v0.14.2 にそれぞれ更新されました。GITOPS-1273

- ソース namespace から Argo CD インスタンスが削除されると、永続的な RBAC ロールおよびバインディングがターゲット namespace から自動的に削除されるようになりました。GITOPS-1228

- 以前のバージョンでは、Argo CD インスタンスを namespace にデプロイする際に、Argo CD インスタンスは "managed-by" ラベルを独自の namespace に変更していました。今回の修正により、namespace のラベルが解除されると同時に、namespace に必要な RBAC ロールおよびバインディングが作成され、削除されるようになりました。GITOPS-1247

- 以前のバージョンでは、Argo CD ワークロードのデフォルトのリソース要求制限 (特に repo-server およびアプリケーションコントローラーの制限) が、非常に厳しかったことがわかりました。現在は、既存のリソースクォータが削除され、リポジトリーサーバーのデフォルトのメモリー制限が 1024M に増えました。この変更は新規インストールにのみ影響することに注意してください。既存の Argo CD インスタンスのワークロードには影響はありません。GITOPS-1274

### 4.1.3. Red Hat OpenShift GitOps 1.2 のリリースノート

Red Hat OpenShift GitOps 1.2 を OpenShift Container Platform 4.7 および 4.8 でご利用いただけるようになりました。

#### 4.1.3.1. サポート表

現在、今回のリリースに含まれる機能にはテクノロジープレビューのものがあります。これらの実験的機能は、実稼働環境での使用を目的としていません。

テクノロジープレビュー機能のサポート範囲

以下の表では、機能は以下のステータスでマークされています。

- **TP**: テクノロジープレビュー

- **GA**: **一般公開機能**

これらの機能に関しては、Red Hat カスタマーポータルの以下のサポート範囲を参照してください。

**表4.2 サポート表**

| 機能 | Red Hat OpenShift GitOps 1.2 |
|---|---|
| Argo CD | GA |
| Argo CD ApplicationSet | TP |
| Red Hat OpenShift GitOps Application Manager (kam) | TP |

#### 4.1.3.2. 新機能

以下のセクションでは、修正および安定性の面での改善点に加え、Red Hat OpenShift GitOps 1.2 の主な新機能について説明します。

- openshift-gitops namespace への読み取りまたは書き込みアクセスがない場合、GitOps Operator で **DISABLE_DEFAULT_ARGOCD_INSTANCE** 環境変数を使用でき、値を **TRUE** に設定し、デフォルトの Argo CD インスタンスが **openshift-gitops** namespace で開始されないようにすることができます。

- リソース要求および制限は Argo CD ワークロードで設定されるようになりました。リソースクォータは **openshift-gitops** namespace で有効になっています。そのため、openshift-gitops namespace に手動でデプロイされる帯域外ワークロードは、リソース要求および制限で設定し、リソースクォータを増やす必要がある場合があります。

- Argo CD 認証は Red Hat SSO と統合され、クラスターの OpenShift 4 アイデンティティープロバイダーに自動的に設定されるようになりました。この機能はデフォルトで無効にされています。Red Hat SSO を有効にするには、以下に示すように **ArgoCD** CR に SSO 設定を追加します。現在、**keycloak** が唯一サポートされているプロバイダーです。

```
apiVersion: argoproj.io/v1alpha1
kind: ArgoCD
metadata:
  name: example-argocd
  labels:
    example: basic
spec:
  sso:
    provider: keycloak
  server:
    route:
      enabled: true
```

- ルートラベルを使用してホスト名を定義して、ルーターのシャード化をサポートするようになりました。**server** (argocd サーバー)、**grafana** ルートおよび **prometheus** ルートに対するラベルの設定のサポートが利用可能になりました。ルートにラベルを設定するには、**ArgoCD** CR のサーバーのルート設定に **labels** を追加します。

  **argocd サーバーにラベルを設定する ArgoCD CR YAML の例**

```
apiVersion: argoproj.io/v1alpha1
kind: ArgoCD
metadata:
  name: example-argocd
  labels:
    example: basic
spec:
  server:
    route:
      enabled: true
      labels:
        key1: value1
        key2: value2
```

- GitOps Operator は、ラベルを適用してターゲット namespace のリソースを管理するために Argo CD インスタンスへのパーミッションを自動的に付与するようになりました。ユーザーは、ターゲット namespace に **argocd.argoproj.io/managed-by: <source-namespace>** のラベルを付けます。**source-namespace** は、argocd インスタンスがデプロイされる namespace に置き換えます。

#### 4.1.3.3. 修正された問題

以下の問題は、現在のリリースで解決されています。

- 以前のバージョンでは、ユーザーが openshift-gitops namespace のデフォルトのクラスターインスタンスで管理される Argo CD の追加のインスタンスを作成した場合は、新規の Argo CD インスタンスに対応するアプリケーションが **OutOfSync** ステータスのままになる可能性がありました。この問題は、所有者の参照をクラスターシークレットに追加することで解決されています。GITOPS-1025

#### 4.1.3.4. 既知の問題

これらは Red Hat OpenShift GitOps 1.2 の既知の問題です。

- Argo CD インスタンスがソース namespace から削除されると、ターゲット namespace の **argocd.argoproj.io/managed-by** ラベルは削除されません。GITOPS-1228

- リソースクォータが Red Hat OpenShift GitOps 1.2 の openshift-gitops namespace で有効になっています。これは、手動でデプロイされる帯域外ワークロードおよび **openshift-gitops** namespace のデフォルトの Argo CD インスタンスによってデプロイされるワークロードに影響を及ぼします。Red Hat OpenShift GitOps **v1.1.2** から **v1.2** にアップグレードする場合は、このようなワークロードをリソース要求および制限で設定する必要があります。追加のワークロードがある場合は、openshift-gitops namespace のリソースクォータを増やす必要があります。
  **openshift-gitops** namespace の現在のリソースクォータ。

| リソース | 要求 | 制限 |
|---|---|---|
| CPU | 6688m | 13750m |
| メモリー | 4544Mi | 9070Mi |

以下のコマンドを使用して CPU 制限を更新できます。

```
$ oc patch resourcequota openshift-gitops-compute-resources -n openshift-gitops --
type='json' -p='[{"op": "replace", "path": "/spec/hard/limits.cpu", "value":"9000m"}]'
```

以下のコマンドを使用して CPU 要求を更新できます。

```
$ oc patch resourcequota openshift-gitops-compute-resources -n openshift-gitops --
type='json' -p='[{"op": "replace", "path": "/spec/hard/cpu", "value":"7000m"}]
```

上記のコマンドのパスは、**cpu** から **memory** を置き換えてメモリーを更新できます。

### 4.1.4. Red Hat OpenShift GitOps 1.1 のリリースノート

Red Hat OpenShift GitOps 1.1 を OpenShift Container Platform 4.7 でご利用いただけるようになりました。

#### 4.1.4.1. サポート表

現在、今回のリリースに含まれる機能にはテクノロジープレビューのものがあります。これらの実験的機能は、実稼働環境での使用を目的としていません。

テクノロジープレビュー機能のサポート範囲

以下の表では、機能は以下のステータスでマークされています。

- **TP: テクノロジープレビュー**
- **GA: 一般公開機能**

これらの機能に関しては、Red Hat カスタマーポータルの以下のサポート範囲を参照してください。

**表4.3 サポート表**

| 機能 | Red Hat OpenShift GitOps 1.1 |
|---|---|
| Argo CD | GA |
| Argo CD ApplicationSet | TP |
| Red Hat OpenShift GitOps Application Manager (kam) | TP |

#### 4.1.4.2. 新機能

以下のセクションでは、修正および安定性の面での改善点に加え、Red Hat OpenShift GitOps 1.1 の主な新機能について説明します。

- **ApplicationSet** 機能が追加されました (テクノロジープレビュー)。**ApplicationSet** 機能は、多数のクラスターまたはモノリポジトリー内で Argo CD アプリケーションを管理する際に、自動化およびより大きな柔軟性を可能にします。また、マルチテナント Kubernetes クラスターでセルフサービスを使用できるようにします。
- Argo CD はクラスターロギングスタックおよび OpenShift Container Platform Monitoring およびアラート機能に統合されるようになりました。

- Argo CD 認証が OpenShift Container Platform に統合されるようになりました。
- Argo CD アプリケーションコントローラーが水平的なスケーリングをサポートするようになりました。
- Argo CD Redis サーバーが高可用性 (HA) をサポートするようになりました。

#### 4.1.4.3. 修正された問題

以下の問題は、現在のリリースで解決されています。

- 以前のバージョンでは、Red Hat OpenShift GitOps は、アクティブなグローバルプロキシー設定のあるプロキシーサーバー設定で予想通りに機能しませんでした。この問題は修正され、Argo CD は Pod の完全修飾ドメイン名 (FQDN) を使用して Red Hat OpenShift GitOps Operator によって設定され、コンポーネント間の通信を有効にできるようになりました。GITOPS-703
- Red Hat OpenShift GitOps バックエンドは、Red Hat OpenShift GitOps URL の **?ref=** クエリーパラメーターを使用して API 呼び出しを行います。以前のバージョンでは、このパラメーターは URL から読み取られず、バックエンドでは常にデフォルトの参照が考慮されました。この問題は修正され、Red Hat OpenShift GitOps バックエンドは Red Hat OpenShift GitOps URL から参照クエリーパラメーターを抽出し、入力参照が指定されていない場合にのみデフォルトの参照を使用します。GITOPS-817
- 以前のバージョンでは、Red Hat OpenShift GitOps バックエンドは有効な GitLab リポジトリーを見つけることができませんでした。これは、Red Hat OpenShift GitOps バックエンドが GitLab リポジトリーの **master** ではなく、ブランチ参照として **main** の有無を確認していたためです。この問題は修正されています。GITOPS-768
- OpenShift Container Platform Web コンソールの **Developer** パースペクティブの **Environments** ページには、アプリケーションの一覧および環境の数が表示されるようになりました。このページには、すべてのアプリケーションを一覧表示する Argo CD **Applications** ページに転送する Argo CD リンクも表示されます。Argo CD **Applications** ページには、選択したアプリケーションのみをフィルターできる **LABELS** (例: **app.kubernetes.io/name=appName**) があります。GITOPS-544

#### 4.1.4.4. 既知の問題

これらは Red Hat OpenShift GitOps 1.1 の既知の問題です。

- Red Hat OpenShift GitOps は Helm v2 および ksonnet をサポートしません。
- Red Hat SSO (RH SSO) Operator は、非接続クラスターではサポートされません。そのため、Red Hat OpenShift GitOps Operator および RH SSO 統合は非接続クラスターではサポートされません。
- OpenShift Container Platform Web コンソールから Argo CD アプリケーションを削除すると、Argo CD アプリケーションはユーザーインターフェイスで削除されますが、デプロイメントは依然としてクラスターに残ります。回避策として、Argo CD コンソールから Argo CD アプリケーションを削除します。GITOPS-830

#### 4.1.4.5. 互換性を破る変更

##### 4.1.4.5.1. Red Hat OpenShift GitOps v1.0.1 からのアップグレード

Red Hat OpenShift GitOps **v1.0.1** から **v1.1** にアップグレードすると、Red Hat OpenShift GitOps Operator は **openshift-gitops** namespace で作成されたデフォルトの Argo CD インスタンスの名前を **argocd-cluster** から **openshift-gitops** に変更します。

これは互換性を破る変更であり、アップグレード前に以下の手順を手動で実行する必要があります。

1. OpenShift Container Platform Web コンソールに移動し、**openshift-gitops** namespace の **argocd-cm.yml** 設定マップファイルの内容をローカルファイルにコピーします。コンテンツの例を以下に示します。

**argocd 設定マップ YAML の例**

```
kind: ConfigMap
apiVersion: v1
metadata:
selfLink: /api/v1/namespaces/openshift-gitops/configmaps/argocd-cm
resourceVersion: '112532'
name: argocd-cm
uid: f5226fbc-883d-47db-8b53-b5e363f007af
creationTimestamp: '2021-04-16T19:24:08Z'
managedFields:
...
namespace: openshift-gitops
labels:
  app.kubernetes.io/managed-by: argocd-cluster
  app.kubernetes.io/name: argocd-cm
  app.kubernetes.io/part-of: argocd
data: "" (1)
admin.enabled: 'true'
statusbadge.enabled: 'false'
resource.exclusions: |
  - apiGroups:
    - tekton.dev
    clusters:
    - '*'
    kinds:
    - TaskRun
    - PipelineRun
ga.trackingid: ''
repositories: |
  - type: git
    url: https://github.com/user-name/argocd-example-apps
ga.anonymizeusers: 'false'
help.chatUrl: ''
url: >-
  https://argocd-cluster-server-openshift-gitops.apps.dev-svc-4.7-
041614.devcluster.openshift.com    "" (2)
help.chatText: ''
kustomize.buildOptions: ''
resource.inclusions: ''
repository.credentials: ''
users.anonymous.enabled: 'false'
configManagementPlugins: ''
application.instanceLabelKey: ''
```

1. **argocd-cm.yml** 設定マップファイルの内容の **data** セクションのみを手動で復元します。
2. 設定マップエントリーの URL の値を、新規インスタンス名 **openshift-gitops** に置き換えます。

2. デフォルトの **argocd-cluster** インスタンスを削除します。
3. 新規の **argocd-cm.yml** 設定マップファイルを編集して、**data** セクション全体を手動で復元します。
4. 設定マップエントリーの URL の値を、新規インスタンス名 **openshift-gitops** に置き換えます。たとえば、前述の例では、URL の値を以下の URL の値に置き換えます。

```
url: >-
  https://openshift-gitops-server-openshift-gitops.apps.dev-svc-4.7-
041614.devcluster.openshift.com
```

5. Argo CD クラスターにログインし、直前の設定が存在することを確認します。

## 4.2. OPENSHIFT GITOPS について

### 4.2.1. GitOps について

GitOps は、クラウドネイティブアプリケーションの継続的デプロイメントを実装するための宣言的な方法です。GitOps を使用して、複数クラスターの Kubernetes 環境全体で、OpenShift Container Platform クラスターおよびアプリケーションを管理するための反復可能なプロセスを作成できます。GitOps は、速いペースで複雑なデプロイメントを処理して自動化し、デプロイメントおよびリリースサイクルでの時間を節約します。

GitOps ワークフローは、開発、テスト、ステージング、および実稼働環境にアプリケーションをプッシュします。GitOps は新しいアプリケーションをデプロイするか、または既存のアプリケーションを更新するため、必要なのはリポジトリーの更新のみとなります。他のものはすべて GitOps が自動化します。

GitOps は、Git プル要求を使用してインフラストラクチャーおよびアプリケーションの設定を管理する一連の手法で設定されます。GitOps では、Git リポジトリーが、システムおよびアプリケーション設定の信頼できる唯一の情報源 (source of truth) になります。この Git リポジトリーには、指定した環境に必要なインフラストラクチャーの宣言的な説明が含まれ、環境を説明した状態に一致させるための自動プロセスが含まれます。また、Git リポジトリーにはシステムの全体の状態が含まれるため、システムの状態への変更の追跡情報が表示され、監査可能になります。GitOps を使用することで、インフラストラクチャーおよびアプリケーション設定のスプロールの問題を解決します。

GitOps は、インフラストラクチャーおよびアプリケーションの定義をコードとして定義します。次に、このコードを使用して複数のワークスペースおよびクラスターを管理し、インフラストラクチャーおよびアプリケーション設定の作成を単純化します。コードの原則に従って、クラスターおよびアプリケーションの設定を Git リポジトリーに保存し、Git ワークフローに従って、これらのリポジトリーを選択したクラスターに適用することができます。Git リポジトリーでのソフトウェアの開発およびメンテナーンスのコアとなる原則を、クラスターおよびアプリケーションの設定ファイルの作成および管理に適用できます。

### 4.2.2. Red Hat OpenShift GitOps について

Red Hat OpenShift GitOps は、異なる環境 (開発、ステージ、実稼働環境など) の異なるクラスターにアプリケーションをデプロイする場合に、アプリケーションの一貫性を確保します。Red Hat

OpenShift GitOps は、設定リポジトリーに関連するデプロイメントプロセスを整理し、それらを中心的な要素にします。これには、少なくとも 2 つのリポジトリーが常に含まれます。

1. ソースコードを含むアプリケーションリポジトリー
2. アプリケーションの必要な状態を定義する環境設定リポジトリー

これらのリポジトリーには、指定した環境で必要なインフラストラクチャーの宣言的な説明が含まれます。また、環境を記述された状態に一致させる自動プロセスも含まれています。

Red Hat OpenShift GitOps は Argo CD を使用してクラスターリソースを維持します。Argo CD は、アプリケーションの継続的インテグレーションおよび継続的デプロイメント (CI/CD) のオープンソースの宣言型ツールです。Red Hat OpenShift GitOps は Argo CD をコントローラーとして実装し、Git リポジトリーで定義されるアプリケーション定義および設定を継続的に監視します。次に、Argo CD は、これらの設定の指定された状態をクラスターのライブ状態と比較します。

Argo CD は、指定した状態から逸脱する設定を報告します。これらの報告により、管理者は、設定を定義された状態に自動または手動で再同期することができます。したがって、ArgoCD を使用して、OpenShift Container Platform クラスターを設定するために使用されるリソースなどのグローバルカスタムリソースを配信できます。

#### 4.2.2.1. 主な特長

Red Hat OpenShift GitOps は、以下のタスクを自動化する上で役立ちます。

- クラスターに設定、モニターリングおよびストレージについての同様の状態があることの確認。
- クラスターを既知の状態からのリカバリーまたは再作成。
- 複数の OpenShift Container Platform クラスターに対する設定変更を適用するか、またはこれを元に戻す。
- テンプレート化された設定の複数の異なる環境への関連付け。
- ステージから実稼働環境へと、クラスター全体でのアプリケーションのプロモート。

## 4.3. OPENSHIFT GITOPS の使用を開始する

Red Hat OpenShift GitOps は Argo CD を使用して、プラットフォーム Operator、オプションの Operator Lifecycle Manager (OLM) Operator、およびユーザー管理などの特定のクラススコープのリソースを管理します。

以下では、Red Hat OpenShift GitOps Operator を OpenShift Container Platform クラスターにインストールし、Argo CD インスタンスにログインする方法について説明します。

### 4.3.1. Web コンソールでの GitOps Operator のインストール

**前提条件**

- OpenShift Container Platform Web コンソールにアクセスします。
- **cluster-admin** ロールを持つアカウントがある。
- OpenShift クラスターにログインしている。

**警告**

Red Hat OpenShift GitOps Operator をインストールする前にコミュニティーバージョンの Argo CD Operator がすでにインストールされている場合は、Argo CD Community Operator を削除します。

**手順**

1. Web コンソールの **Administrator** パースペクティブで、左側のメニューにある **Operators → OperatorHub** に移動します。
2. **OpenShift GitOps** を検索し、**Red Hat OpenShift GitOps** タイルをクリックし、**Install** をクリックします。
   Red Hat OpenShift GitOps は、クラスターのすべての namespace にインストールされます。

Red Hat OpenShift GitOps Operator がインストールされると、**openshift-gitops** namespace で利用可能なすぐに使える Argo CD インスタンスが自動的に設定され、Argo CD アイコンがコンソールツールバーに表示されます。プロジェクトでアプリケーション用に後続の Argo CD インスタンスを作成できます。

## 4.4. GIT リポジトリーとアプリケーションを再帰的に同期するための ARGO CD の設定

### 4.4.1. クラスター設定を使用したアプリケーションのデプロイによる OpenShift クラスターの設定

Red Hat OpenShift GitOps では、Argo CD を、クラスターのカスタム設定が含まれるアプリケーションと Git ディレクトリーの内容を再帰的に同期するように設定することができます。

**前提条件**

- Red Hat OpenShift GitOps がクラスターにインストールされている。

#### 4.4.1.1. OpenShift 認証情報を使用した Argo CD インスタンスへのログイン

Red Hat OpenShift GitOps Operator は **openshift-gitops** namespace で利用可能なすぐに使用できる Argo CD インスタンスを自動的に作成します。

**前提条件**

- Red Hat OpenShift GitOps Operator がクラスターにインストールされている。

**手順**

1. OpenShift Container Platform Web コンソールの **Administrator** パースペクティブで、**Operators → Installed Operators** に移動し、Red Hat OpenShift GitOps Operator がインストールされていることを確認します。

2.  menu → **OpenShift GitOps → Cluster Argo CD** の順に移動します。Argo CD UI のログインページは、新規ウィンドウに表示されます。

3. Argo CD インスタンスのパスワードを取得します。

   a. Web コンソールの **Developer** パースペクティブに移動します。利用可能なプロジェクトの一覧が表示されます。

   b. **openshift-gitops** プロジェクトに移動します。

   c. 左側のナビゲーションパネルを使用して、**Secrets** ページに移動します。

   d. **openshift-gitops-cluster** インスタンスを選択して、パスワードを表示します。

   e. パスワードをコピーします。

4. このパスワードおよび **admin** をユーザー名として使用し、新しいウィンドウで Argo CD UI にログインします。

#### 4.4.1.2. Argo CD ダッシュボードを使用したアプリケーションの作成

Argo CD は、アプリケーションを作成できるダッシュボードを提供します。

このサンプルワークフローでは **cluster** ディレクトリーの内容を **cluster-configs** アプリケーションに対して再帰的に同期するために Argo CD を設定するプロセスについて説明します。ディレクトリーは Web コンソールの  メニューで **Red Hat Developer Blog - Kubernetes**へのリンクを追加する OpenShift Container Platform Web コンソールクラスター設定を定義してクラスターの namespace **spring-petclinic** を定義します。

**手順**

1. Argo CD ダッシュボードで、**New App** をクリックして新規の Argo CD アプリケーションを追加します。

2. このワークフローでは、以下の設定で **cluster-configs** アプリケーションを作成します。

   **アプリケーション名**
   **cluster-configs**
   **プロジェクト**
   **default**
   **同期ポリシー**
   **Manual**
   **リポジトリー URL**
   **https://github.com/redhat-developer/openshift-gitops-getting-started**
   **リビジョン**
   **HEAD**
   **パス**
   **cluster**
   **宛先**

**https://kubernetes.default.svc**

namespace

**spring-petclinic**

**ディレクトリーの再帰処理**

**checked**

3. **Create** をクリックしてアプリケーションを作成します。
4. Web コンソールの **Administrator** パースペクティブで、左側のメニューにある **Administration → Namespaces** に移動します。
5. namespace を検索、選択してから **Label** フィールドに **argocd.argoproj.io/managed-by=openshift-gitops** を入力し、**openshift-gitops** namespace にある Argo CD インスタンスが namespace を管理できるようにします。

#### 4.4.1.3. oc ツールを使用したアプリケーションの作成

**oc** ツールを使用して、ターミナルで Argo CD アプリケーションを作成できます。

**手順**

1. サンプルアプリケーション をダウンロードします。

```
$ git clone git@github.com:redhat-developer/openshift-gitops-getting-started.git
```

2. アプリケーションを作成します。

```
$ oc create -f openshift-gitops-getting-started/argo/cluster.yaml
```

3. **oc get** コマンドを実行して、作成されたアプリケーションを確認します。

```
$ oc get application -n openshift-gitops
```

4. アプリケーションがデプロイされている namespace にラベルを追加し、**openshift-gitops** namespace の Argo CD インスタンスが管理できるようにします。

```
$ oc label namespace spring-petclinic argocd.argoproj.io/managed-by=openshift-gitops
```

#### 4.4.1.4. アプリケーションの Git リポジトリーとの同期

**手順**

1. Argo CD ダッシュボードでは、**cluster-configs** Argo CD アプリケーションに **Missing** および **OutOfSync** のステータスがあることに注意してください。アプリケーションは手動の同期ポリシーで設定されているため、Argo CD はこれを自動的に同期しません。
2. **cluster-configs** タイルの **同期** をクリックし、変更を確認してから、**Synchronize** をクリックします。Argo CD は Git リポジトリーの変更を自動的に検出します。設定が変更されると、Argo CD は **cluster-configs** のステータスを **OutOfSync** に変更します。Argo CD の同期ポリシーを変更し、Git リポジトリーからクラスターに変更を自動的に適用できるようにします。

3. **cluster-configs** Argo CD アプリケーションに **Healthy** および **Synced** のステータスがあることに注意してください。**cluster-configs** タイルをクリックし、クラスター上で同期されたリソースおよびそれらのステータスの詳細を確認します。

4. OpenShift Container Platform Web コンソールに移動し、  をクリックして **Red Hat Developer Blog - Kubernetes** へのリンクが表示されることを確認します。

5. **Project** ページに移動し、**spring-petclinic** namespace を検索し、これがクラスターに追加されていることを確認します。
クラスター設定がクラスターに正常に同期されます。

## 4.4.2. Argo CD を使用した Spring Boot アプリケーションのデプロイ

Argo CD を使用すると、Argo CD ダッシュボードまたは **oc** ツールを使用して、アプリケーションを OpenShift クラスターにデプロイできます。

**前提条件**

- Red Hat OpenShift GitOps がクラスターにインストールされている。

### 4.4.2.1. OpenShift 認証情報を使用した Argo CD インスタンスへのログイン

Red Hat OpenShift GitOps Operator は **openshift-gitops** namespace で利用可能なすぐに使用できる Argo CD インスタンスを自動的に作成します。

**前提条件**

- Red Hat OpenShift GitOps Operator がクラスターにインストールされている。

**手順**

1. OpenShift Container Platform Web コンソールの **Administrator** パースペクティブで、**Operators → Installed Operators** に移動し、Red Hat OpenShift GitOps Operator がインストールされていることを確認します。

2.  menu → **OpenShift GitOps → Cluster Argo CD** の順に移動します。Argo CD UI のログインページは、新規ウィンドウに表示されます。

3. Argo CD インスタンスのパスワードを取得します。

   a. Web コンソールの **Developer** パースペクティブに移動します。利用可能なプロジェクトの一覧が表示されます。

   b. **openshift-gitops** プロジェクトに移動します。

   c. 左側のナビゲーションパネルを使用して、**Secrets** ページに移動します。

   d. **openshift-gitops-cluster** インスタンスを選択して、パスワードを表示します。

   e. パスワードをコピーします。

4. このパスワードおよび **admin** をユーザー名として使用し、新しいウィンドウで Argo CD UI にログインします。

#### 4.4.2.2. Argo CD ダッシュボードを使用したアプリケーションの作成

Argo CD は、アプリケーションを作成できるダッシュボードを提供します。

このサンプルワークフローでは **cluster** ディレクトリーの内容を **cluster-configs** アプリケーションに対して再帰的に同期するために Argo CD を設定するプロセスについて説明します。ディレクトリーは Web コンソールの  メニューで **Red Hat Developer Blog - Kubernetes**へのリンクを追加する OpenShift Container Platform Web コンソールクラスター設定を定義してクラスターの namespace **spring-petclinic** を定義します。

**手順**

1. Argo CD ダッシュボードで、**New App** をクリックして新規の Argo CD アプリケーションを追加します。
2. このワークフローでは、以下の設定で **cluster-configs** アプリケーションを作成します。

   **アプリケーション名**
   **cluster-configs**
   **プロジェクト**
   **default**
   **同期ポリシー**
   **Manual**
   **リポジトリー URL**
   **https://github.com/redhat-developer/openshift-gitops-getting-started**
   **リビジョン**
   **HEAD**
   **パス**
   **cluster**
   **宛先**
   **https://kubernetes.default.svc**
   namespace
   **spring-petclinic**
   **ディレクトリーの再帰処理**
   **checked**

3. このワークフローでは、以下の設定で **spring-petclinic** アプリケーションを作成します。

   **アプリケーション名**
   **spring-petclinic**
   **プロジェクト**
   **default**
   **同期ポリシー**
   **Automatic**
   **リポジトリー URL**
   **https://github.com/redhat-developer/openshift-gitops-getting-started**

**リビジョン**

**HEAD**

**パス**

**app**

**宛先**

**https://kubernetes.default.svc**

**namespace**

**spring-petclinic**

4. **Create** をクリックしてアプリケーションを作成します。
5. Web コンソールの **Administrator** パースペクティブで、左側のメニューにある **Administration → Namespaces** に移動します。
6. namespace を検索、選択してから **Label** フィールドに **argocd.argoproj.io/managed-by=openshift-gitops** を入力し、**openshift-gitops** namespace にある Argo CD インスタンスが namespace を管理できるようにします。

#### 4.4.2.3. oc ツールを使用したアプリケーションの作成

**oc** ツールを使用して、ターミナルで Argo CD アプリケーションを作成できます。

**手順**

1. サンプルアプリケーション をダウンロードします。

```
$ git clone git@github.com:redhat-developer/openshift-gitops-getting-started.git
```

2. アプリケーションを作成します。

```
$ oc create -f openshift-gitops-getting-started/argo/app.yaml
```

```
$ oc create -f openshift-gitops-getting-started/argo/cluster.yaml
```

3. **oc get** コマンドを実行して、作成されたアプリケーションを確認します。

```
$ oc get application -n openshift-gitops
```

4. アプリケーションがデプロイされている namespace にラベルを追加し、**openshift-gitops** namespace の Argo CD インスタンスが管理できるようにします。

```
$ oc label namespace spring-petclinic argocd.argoproj.io/managed-by=openshift-gitops
```

```
$ oc label namespace spring-petclinic argocd.argoproj.io/managed-by=openshift-gitops
```

#### 4.4.2.4. Argo CD の自己修復動作の確認

Argo CD は、デプロイされたアプリケーションの状態を常に監視し、Git の指定されたマニフェストとクラスターのライブの変更の違いを検出し、それらを自動的に修正します。この動作は自己修復として言及されます。

Argo CD で自己修復動作をテストし、確認することができます。

**前提条件**

- サンプル **app-spring-petclinic** アプリケーションがデプロイされ、設定されている。

**手順**

1. Argo CD ダッシュボードで、アプリケーションに **Synced** ステータスがあることを確認します。
2. Argo CD ダッシュボードの **app-spring-petclinic** タイルをクリックし、クラスターにデプロイされたアプリケーションのリソースを表示します。
3. OpenShift Web コンソールで、**Developer** パースペクティブに移動します
4. Spring PetClinic デプロイメントを変更し、Git リポジトリーの **app**/ ディレクトリーに変更をコミットします。Argo CD は変更をクラスターに自動的にデプロイします。
5. OpenShift Web コンソールでアプリケーションを監視している間に、クラスターでデプロイメントを変更し、これを 2 つの Pod にスケールアップして自己修復動作をテストします。
   a. 以下のコマンドを実行してデプロイメントを変更します。

   ```
   $ oc scale deployment spring-petclinic --replicas 2  -n spring-petclinic
   ```

   b. OpenShift Web コンソールでは、デプロイメントは 2 つの Pod にスケールアップし、すぐに再び 1 つの Pod にスケールダウンすることに注意してください。Argo CD は Git リポジトリーとの差異を検知し、OpenShift クラスターでアプリケーションを自動的に修復しました。
6. Argo CD ダッシュボードで、**app-spring-petclinic** タイル → **APP DETAILS** → **EVENTS** をクリックします。**EVENTS** タブには、以下のイベントが表示されます。Argo CD がクラスターのデプロイメントリソースが同期されていないことを検知し、Git リポジトリーを再同期してこれを修正します。

## 4.5. OPENSHIFT での ARGO CD の SSO 設定

Red Hat OpenShift GitOps Operator がインストールされると、Argo CD は **admin** パーミッションを持つユーザーを自動的に作成します。複数のユーザーを管理するために、Argo CD ではクラスター管理者が SSO を設定できます。

**注記**

バンドルされた Dex OIDC プロバイダーはサポートされません。

**前提条件**

- Red Hat SSO がクラスターにインストールされている。

### 4.5.1. Keycloak での新規クライアントの作成

**手順**

1. Keycloak サーバーにログインし、使用するレルムを選択して **Clients** ページに移動し、画面の右上にある **Create** をクリックします。
2. 以下の値を指定します。

   **クライアント ID**

   **argocd**

   **クライアントプロトコル**

   **openid-connect**

   **ルート URL**

   <your-argo-cd-route-url>

   **アクセスタイプ**

   **confidential**

   **有効なリダイレクト URI**

   <your-argo-cd-route-url>/auth/callback

   **ベース URL**

   **/applications**
3. **Save** をクリックし、**Client** ページに追加された **Credentials** タブを表示します。
4. その他の設定については、**Credentials** タブからシークレットをコピーします。

### 4.5.2. groups 要求の設定

Argo CD でユーザーを管理するには、認証トークンに追加できるグループ要求を設定する必要があります。

**手順**

1. Keycloak ダッシュボードで **Client Scope** に移動し、以下の値を使用して新規クライアントを追加します。

   Name

   **groups**

   Protocol

   **openid-connect**

   Display On Content Scope

   **ON**

   Include to Token Scope

   **オン**
2. **Save** をクリックし、**groups** → **Mappers** に移動します。
3. 以下の値を使用して新規トークンマッパーを追加します。

   Name

   **groups**

   Mapper Type

   **Group Membership**

Token Claim Name

**groups**
トークンマッパーはクライアントが **groups** を要求する際に **groups** 要求をトークンに追加します。

4. **Clients** → **Client Scopes** に移動し、クライアントをグループスコープを提供するように設定します。**Assigned Default Client Scopes** テーブルで **groups** を選択し、**Add selected** をクリックします。**groups** スコープは、**Available Client Scopes** テーブルにある必要があります。

5. **Users** → **Admin** → **Groups** に移動し、グループ **ArgoCDAdmins** を作成します。

### 4.5.3. Argo CD OIDC の設定

Argo CD OpenID Connect (OIDC) を設定するには、クライアントシークレットを生成し、エンコードし、これをカスタムリソースに追加する必要があります。

**前提条件**

- クライアントシークレットを取得している。

**手順**

1. 生成したクライアントシークレットを保存します。

   a. base64 でクライアントシークレットをエンコードします。

   ```
   $ echo -n '83083958-8ec6-47b0-a411-a8c55381fbd2' | base64
   ```

   b. シークレットを編集し、base64 の値を **oidc.keycloak.clientSecret** キーに追加します。

   ```
   $ oc edit secret argocd-secret -n <namespace>
   ```

   **シークレットの YAML サンプル**

   ```
   apiVersion: v1
   kind: Secret
   metadata:
     name: argocd-secret
   data:
     oidc.keycloak.clientSecret:
   ODMwODM5NTgtOGVjNi00N2IwLWE0MTEtYThjNTUzODFmYmQy
   ```

2. **argocd** カスタムリソースを編集し、OIDC 設定を追加して Keycloak 認証を有効にします。

   ```
   $ oc edit argocd -n <your_namespace>
   ```

   **argocd カスタムリソースの例**

   ```
   apiVersion: argoproj.io/v1alpha1
   kind: ArgoCD
   metadata:
     creationTimestamp: null
   ```

```
  name: argocd
  namespace: argocd
spec:
  resourceExclusions: |
    - apiGroups:
      - tekton.dev
      clusters:
      - '*'
      kinds:
      - TaskRun
      - PipelineRun
  oidcConfig: |
    name: OpenShift Single Sign-On
    issuer: https://keycloak.example.com/auth/realms/myrealm 1
    clientID: argocd 2
    clientSecret: $oidc.keycloak.clientSecret 3
    requestedScopes: ["openid", "profile", "email", "groups"] 4
  server:
    route:
      enabled: true
```

1. **issuer** は正しいレルム名 (この例では **myrealm**) で終了する必要があります。
2. **clientId** は、Keycloak アカウントに設定したクライアント ID です。
3. **clientSecret** は、argocd-secret シークレットで作成した正しいキーを参照します。
4. **requestedScopes** には、グループ要求が Default スコープに追加されていない場合はこれが含まれます。

### 4.5.4. OpenShift での Keycloak Identity Brokering

Keycloak インスタンスを、Identity Brokering 経由の認証に OpenShift を使用するように設定できます。これにより、OpenShift クラスターと Keycloak インスタンス間のシングルサインオン (SSO) が可能になります。

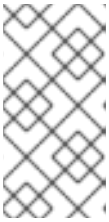
#### 前提条件

- **oc** CLI ツールをインストールしている。

#### 手順

1. OpenShift Container Platform API URL を取得します。

```
$ curl -s -k -H "Authorization: Bearer $(oc whoami -t)" https://<openshift-user-facing-api-url>/apis/config.openshift.io/v1/infrastructures/cluster | jq ".status.apiServerURL".
```



**注記**

多くの場合、OpenShift Container Platform API のアドレスは HTTPS によって保護されます。そのため、コンテナーで X509_CA_BUNDLE を設定し、これを **/var/run/secrets/kubernetes.io/serviceaccount/ca.crt** に設定する必要があります。そうでない場合に、Keycloak は API サーバーと通信できません。

2. Keycloak サーバーのダッシュボードで、 **Identity Providers** に移動し、 **Openshift v4** を選択します。以下の値を指定します。

**Base Url**

OpenShift 4 API URL

**Client ID**

**keycloak-broker**

**Client Secret**

定義するシークレット
これで、Keycloak を Identity Broker として使用し、OpenShift 認証情報を使用して Argo CD にログインできます。

### 4.5.5. 追加の OAuth クライアントの登録

OpenShift Container Platform クラスターの認証を管理するために追加の OAuth クライアントが必要になる場合は、これを登録することができます。

**手順**

- クライアントを登録するには、以下を実行します。

```
$ oc create -f <(echo '
kind: OAuthClient
apiVersion: oauth.openshift.io/v1
metadata:
 name: keycloak-broker (1)
secret: "..." (2)
redirectURIs:
- "https://keycloak-keycloak.apps.dev-svc-4.7-
020201.devcluster.openshift.com/auth/realms/myrealm/broker/openshift-v4/endpoint" (3)
grantMethod: prompt (4)
')
```

(1) OAuth クライアント名は、**<namespace_route>/oauth/authorize** および **<namespace_route>/oauth/token** への要求を実行する際に **client_id** パラメーターとして使用されます。

(2) **secret** は、**<namespace_route>/oauth/token** への要求の実行時に client_secret パラメーターとして使用されます。

(3) **<namespace_route>/oauth/authorize** および **<namespace_route>/oauth/token** への要求で指定される **redirect_uri** パラメーターは、**redirectURIs** パラメーター値に一覧表示されるいずれかの URI と等しいか、またはこれによって接頭辞が付けられている必要があります。

(4) ユーザーがこのクライアントにアクセスを付与していない場合、**grantMethod** は、このクライアントがトークンを要求する場合に実行するアクションを判別します。付与を自動的に承認し、要求を再試行するには **auto** を指定し、ユーザーに対して付与の承認または付与を求めるプロンプトを出す場合には **prompt** を指定します。

### 4.5.6. グループおよび Argo CD RBAC の設定

ロールベースアクセス制御 (RBAC) を使用すると、ユーザーに関連するパーミッションを指定できます。

**前提条件**

- Keycloak で **ArgoCDAdmins** グループを作成している。
- パーミッションを付与するユーザーが Argo CD にログインしている。

**手順**

1. Keycloak ダッシュボードで、**Users → Groups** に移動します。ユーザーを Keycloak グループ **ArgoCDAdmins** に追加します。
2. **ArgoCDAdmins** グループに **argocd-rbac** 設定マップで必要なパーミッションがあることを確認します。
   - 設定マップを編集します。

     ```
     $ oc edit configmap argocd-rbac-cm -n <namespace>
     ```

     **admin パーミッションを定義する設定マップの例。**

     ```
     apiVersion: v1
     kind: ConfigMap
     metadata:
       name: argocd-rbac-cm
     data:
       policy.csv: |
         g, /ArgoCDAdmins, role:admin
     ```

### 4.5.7. Argo CD の組み込みパーミッション

このセクションでは、クラスター Operator、オプションの OLM Operator、およびユーザー管理を含む特定のクラスタースコープのリソースを管理するために ArgoCD に付与されるパーミッションを一覧表示します。ArgoCD には **cluster-admin** パーミッションが付与されていないことに注意してください。

**表4.4 Argo CD に付与されるパーミッション**

| リソースグループ | ユーザーまたは管理者に設定するもの |
|---|---|
| operators.coreos.com | OLM によって管理されるオプション Operator |
| user.openshift.io, rbac.authorization.k8s.io | グループ、ユーザー、およびそれらのパーミッション |
| config.openshift.io | クラスター全体のビルド設定、レジストリー設定、およびスケジューラーポリシーを設定するために使用される CVO によって管理されるコントロールプレーン Operator |
| storage.k8s.io | ストレージ |

| | |
|---|---|
| console.openshift.io | コンソールのカスタマイズ |

## 4.6. GITOPS OPERATOR のサイズ要件

サイジング要件ページには、Red Hat OpenShift GitOps に OpenShift Container Platform をインストールするためのサイジング要件が表示されます。また、GitOps オペレーターによってインスタンス化されるデフォルトの ArgoCD インスタンスのサイジングの詳細も提供します。

### 4.6.1. GitOps のサイジング要件

Red Hat OpenShift GitOps は、クラウドネイティブアプリケーションの継続的デプロイメントを実装するための宣言的な方法です。GitOps を使用すると、アプリケーションの CPU とメモリーの要件を定義および設定できます。

Red Hat OpenShift GitOps Operator をインストールするたびに、namespace 上のリソースが定義された制限内にインストールされます。デフォルトのインストールで制限やリクエストが設定されていない場合、Operator は namespace でクォータを使用して失敗します。十分なリソースがないと、クラスターは Argo CD 関連の Pod をスケジュールできません。次の表に、デフォルトのワークロードのリソースリクエストと制限の詳細を示します。

| ワークロード | CPU 要求 | CPU 上限 | メモリー要求 | メモリー上限 |
|---|---|---|---|---|
| argocd-application-controller | 1 | 2 | 1024M | 2048M |
| applicationset-controller | 1 | 2 | 512M | 1024M |
| argocd-server | 0.125 | 0.5 | 128M | 256M |
| argocd-repo-server | 0.5 | 1 | 256M | 1024M |
| argocd-redis | 0.25 | 0.5 | 128M | 256M |
| argocd-dex | 0.25 | 0.5 | 128M | 256M |
| HAProxy | 0.25 | 0.5 | 128M | 256M |

オプションで、**oc** コマンドで ArgoCD カスタムリソースを使用して、詳細を確認し、変更することもできます。

```
oc edit argocd <name of argo cd> -n namespace
```