OpenShift Container Platform 3.9

開発者ガイド

OpenShift Container Platform 3.9 開発者リファレンス

Last Updated: 2022-04-22

OpenShift Container Platform 3.9 開発者ガイド

OpenShift Container Platform 3.9 開発者リファレンス

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

これらのトピックは、開発者がワークステーションを設定および構成して、コマンドラインインターフェイス（CLI）を使用して OpenShift Container Platformクラウド環境でアプリケーションを開発およびデプロイするのに役立ちます。本書では、開発者向けの詳しい手順と例を紹介します。これは、開発者によるプロジェクトのモニター、Web コンソールの設定および参照、templatesManage ビルドおよび WebhookDefine を使用して CLIGenerate 設定を活用して deploymentIntegrate 外部サービス（データベース、SaaS エンドポイント）の統合を支援します。

目次

第1章 概要

本書はアプリケーション開発者を対象としており、OpenShift Container Platform クラウド環境でアプリケーションを開発およびデプロイするためにワークステーションを設定して構成する方法を説明します。また本書には、詳細な手順と例が含まれ、開発者が以下を実行するのに役立ちます。

1. 新規アプリケーションの作成
2. プロジェクトのモニターおよび設定
3. テンプレートを使用した設定の生成
4. ビルドストラテジーオプションおよび Webhook を含むビルドの管理
5. デプロイメントストラテジーを含むデプロイメントの定義
6. ルートの作成および管理
7. シークレットの作成および設定
8. データベースおよび SaaS エンドポイントなどの外部サービスの統合
9. プローブを使用したアプリケーションのヘルスチェック

第2章 アプリケーションライフサイクル管理

2.1. 開発プロセスの計画

2.1.1. 概要

OpenShift Container Platform はアプリケーションのビルドおよびデプロイするために設計されています。OpenShift Container Platform を開発プロセスにどの程度組み込むかに応じて、以下から選択できます。

- OpenShift Container Platformプロジェクト内で開発に集中し、そのプロジェクトを使用してアプリケーションをゼロから構築し、そのライフサイクルを継続的に開発および管理する。
- 別の環境ですでに開発したアプリケーション（例： バイナリー、コンテナーイメージ、ソースコード）を用意して OpenShift Container Platform にデプロイする。

2.1.2. 開発環境としての OpenShift Container Platform の使用

OPENSHIFT_396538_0316

OpenShift Container Platform を直接使用してアプリケーションの開発をゼロから行うことができます。この種の開発プロセスを計画する場合には、以下の手順を考慮してください。

初期計画

- アプリケーションにはどのような機能があるか？
- どのプログラミング言語を使用して開発するか？

OpenShift Container Platform へのアクセス

- この時点で、ご自身または組織内の管理者が OpenShift Container Platform をインストールする必要があります。

開発

- 任意のエディターまたは IDE を使用して、アプリケーションの基本的なスケルトンを作成します。OpenShift Container Platform で アプリケーションがどのようなタイプのものでるか を認識できるように適切に開発されている必要があります。
- コードを Git リポジトリーにプッシュします。

生成

- **oc new-app** コマンドを使用して 基本的なアプリケーションを作成します。OpenShift Container Platform はビルドおよびデプロイメント設定を生成します。

管理

- アプリケーションコードの開発を開始します。
- アプリケーションが正常にビルドされることを確認します。

- 引き続きコードをローカルで開発し、コードを改良します。
- コードを Git リポジトリーにプッシュします。
- 追加の設定が必要かどうかを確認します。追加のオプションについて『開発者ガイド』で確認してください。

検証

- アプリケーションはさまざまな方法で検証できます。変更をアプリケーションの Git リポジトリーにプッシュし、OpenShift Container Platform を使用してアプリケーションの再ビルドおよび再デプロイを行うことができます。または、**rsync** を使用してホットデプロイを実行し、コードを変更中の Pod に同期できます。

2.1.3. アプリケーションの OpenShift Container Platform へのデプロイ

アプリケーション開発ストラテジーの別の可能性として、ローカルで開発してから OpenShift Container Platform を使用して完全に開発されたアプリケーションをデプロイする方法があります。アプリケーションコードを先に準備してからビルドし、完了後に OpenShift Container Platform インストールにデプロイする場合は、以下の手順を使用します。

初期計画

- アプリケーションにはどのような機能があるか？
- どのプログラミング言語を使用して開発するか？

開発

- 任意のエディターまたは IDE を使用してアプリケーションコードを開発します。
- アプリケーションコードをローカルでビルドしてテストします。
- コードを Git リポジトリーにプッシュします。

OpenShift Container Platform へのアクセス

- この時点で、ご自身または組織内の管理者が OpenShift Container Platform をインストールする必要があります。

生成

- **oc new-app** コマンドを使用して 基本的なアプリケーションを作成します。OpenShift Container Platform はビルドおよびデプロイメント設定を生成します。

検証

- 前述の生成手順においてビルドおよびデプロイしたアプリケーションが OpenShift Container Platform で正常に実行されていることを確認します。

管理

- 結果に満足するまで、アプリケーションコードの開発を続けます。
- 新たにプッシュされたコードを受け入れるには、アプリケーションを OpenShift Container Platform で再ビルドします。
- 追加の設定が必要かどうかを確認します。追加のオプションについて『開発者ガイド』で確認してください。

2.2. 新規アプリケーションの作成

2.2.1. 概要

OpenShift CLI または Web コンソールのいずれかを使用して、ソースまたはバイナリーコード、イメージおよびテンプレート（あるいは両方）を含むコンポーネントから新規の OpenShift Container Platform アプリケーションを作成できます。

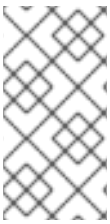
2.2.2. CLI を使用したアプリケーションの作成

2.2.2.1. ソースコードからのアプリケーションの作成

new-app コマンドを使用して、ローカルまたはリモート Git リポジトリーのソースコードからアプリケーションを作成できます。

ローカルディレクトリーの Git リポジトリーを使用してアプリケーションを作成するには、以下を実行します。

```
$ oc new-app /path/to/source/code
```



注記

ローカル Git リポジトリーを使用する場合には、リポジトリーで OpenShift Container Platform クラスターがアクセス可能な URL を参照する **origin** という名前のリモートリポジトリーが必要です。認識されているリモートがない場合は、**new-app** により バイナリービルド が作成されます。

リモート Git リポジトリーを使用してアプリケーションを作成するには、以下を実行します。

```
$ oc new-app https://github.com/sclorg/cakephp-ex
```

プライベートのリモート Git リポジトリーを使用してアプリケーションを作成するには、以下を実行します。

```
$ oc new-app https://github.com/youruser/yourprivaterepo --source-secret=yoursecret
```

注記

プライベートリモート Git リポジトリーを使用する場合には、**--source-secret** フラグを使用して、既存のソースクローンのシークレットを指定できます。このシークレットは、**BuildConfig** に挿入され、リポジトリーにアクセスできるようになります。

--context-dir フラグを指定することで、ソースコードリポジトリーのサブディレクトリーを使用できます。リモート Git リポジトリーおよびコンテキストサブディレクトリーを使用してアプリケーションを作成する場合は、以下を実行します。

```
$ oc new-app https://github.com/sclorg/s2i-ruby-container.git \
    --context-dir=2.0/test/puma-test-app
```

また、リモート URL を指定する場合は、以下のように URL の最後に **#<branch_name>** を追加することで、使用する Git ブランチを指定できます。

```
$ oc new-app https://github.com/openshift/ruby-hello-world.git#beta4
```

new-app コマンドは、ビルド設定を作成し、これはソースコードから新規のアプリケーションイメージを作成します。**new-app** コマンドは通常、デプロイメント設定を作成して新規のイメージをデプロイするほか、サービス を作成してイメージを実行するデプロイメントへの負荷分散したアクセスを提供します。

OpenShift Container Platform は **Docker**、**Pipeline** または **Source ビルドストラテジー** のいずれを使用すべきかを自動的に 検出 します。 また、**Source** ビルドの場合は、適切な言語のビルダーイメージを検出します。

ビルドストラテジーの検出

新規アプリケーションの作成時に **Jenkinsfile** がソースリポジトリーのルートまたは指定されたコンテキストディレクトリーに存在する場合に、OpenShift Container Platform は **Pipeline** ビルドストラテジーを生成します。または、**Dockerfile** がある場合、OpenShift Container Platform は **Docker** ビルドストラテジー を生成します。それ以外の場合は、**Source**ビルドストラテジーが生成されます。

ビルドストラテジーを上書きするには、**--strategy** フラグを **docker**、**pipeline** または **source**のいずれかに設定します。

```
$ oc new-app /home/user/code/myapp --strategy=docker
```

注記

oc コマンドを使用するには、ビルドソースを含むファイルがリモートの git リポジトリーで利用可能である必要があります。ソースのすべてのビルドには、**git remote -v** を使用する必要があります。

言語の検出

ソース ビルドストラテジーを使用する場合に、**new-app** はリポジトリーのルート または指定したコンテキストディレクトリーに特定のファイルが存在するかどうかで、使用する言語ビルダーを判別しようとします。

表2.1 new-app が検出する言語

言語	ファイル
dotnet	project.json、*.csproj
jee	pom.xml

言語	ファイル
nodejs	app.json、package.json
perl	cpanfile、index.pl
php	composer.json、index.php
python	requirements.txt、setup.py
ruby	Gemfile、Rakefile、config.ru
scala	build.sbt
golang	Godeps、main.go

言語の検出後、**new-app** は OpenShift Container Platform サーバーで、検出言語と一致して **supports** アノテーションが指定されたイメージストリームタグか、または検出された言語の名前に一致するイメージストリームの有無を検索します。一致するものが見つからない場合には、**new-app** は Docker Hub レジストリー で名前をベースにした検出言語と一致するイメージの検索を行います。

~ をセパレーターとして使用し、イメージ (イメージストリームまたはコンテナーの仕様) とリポジトリーを指定して、ビルダーが特定のソースリポジトリーを使用するようにイメージを上書きすることができます。この方法を使用すると、ビルドストラテジーの検出および言語の検出は実行されない点に留意してください。

たとえば、リモートリポジトリーのソースを使用して **myproject/my-ruby** イメージストリームを作成する場合は、以下を実行します。

```
$ oc new-app myproject/my-ruby~https://github.com/openshift/ruby-hello-world.git
```

ローカルリポジトリーのソースを使用して **openshift/ruby-20-centos7:latest** コンテナーのイメージストリームを作成するには、以下を実行します。

```
$ oc new-app openshift/ruby-20-centos7:latest~/home/user/code/my-ruby-app
```

2.2.2.2. イメージからアプリケーションを作成する方法

既存のイメージからアプリケーションのデプロイが可能です。イメージは、OpenShift Container Platform サーバー内のイメージストリーム、指定したレジストリー内のイメージ、Docker Hub レジストリー、またはローカルの Docker サーバー内のイメージから取得できます。

new-app コマンドは、渡された引数に指定されたイメージの種類を判断しようとします。ただし、イメージが (**--docker-image** 引数を使用した) Docker イメージなのか、または (**-i|--image** 引数) を使用したイメージストリームなのかを **new-app** に明示的に指示できます。

注記

ローカル Docker リポジトリーからイメージを指定した場合、同じイメージが OpenShift Container Platform のクラスターノードでも利用できることを確認する必要があります。

たとえば、DockerHub MySQL イメージからアプリケーションを作成するには、以下を実行します。

```
$ oc new-app mysql
```

プライベートのレジストリーのイメージを使用してアプリケーションを作成する場合には、Docker イメージの仕様全体を以下のように指定します。

```
$ oc new-app myregistry:5000/example/myimage
```

注記

イメージを含むレジストリーが SSL でセキュリティー保護されていない場合には、クラスター管理者は、OpenShift Container Platform ノードホストの Docker デーモンが、対象のレジストリーを参照する **--insecure-registry** フラグを指定して実行されていることを確認する必要があります。また **--insecure-registry** フラグを指定して、セキュアでないレジストリーからイメージが取得されていることを **new-app** に指定する必要があります。

既存のイメージストリームおよび任意のイメージストリームタグでアプリケーションを作成します。

```
$ oc new-app my-stream:v1
```

2.2.2.3. テンプレートからのアプリケーションの作成

テンプレート名を引数として指定することで、事前に保存したテンプレートまたはテンプレートファイルからアプリケーションを作成することができます。たとえば、サンプルアプリケーションテンプレートを保存し、これを利用してアプリケーションを作成できます。

保存したテンプレートからアプリケーションを作成するには、以下を実行します。

```
$ oc create -f examples/sample-app/application-template-stibuild.json
$ oc new-app ruby-helloworld-sample
```

事前に OpenShift Container Platform に保存することなく、ローカルファイルシステムでテンプレートを直接使用するには、**-f|--file** 引数を使用します。

```
$ oc new-app -f examples/sample-app/application-template-stibuild.json
```

テンプレートパラメーター

テンプレート をベースとするアプリケーションを作成する場合、以下の **-p|--param** 引数を使用してテンプレートで定義したパラメーター値を設定します。

```
$ oc new-app ruby-helloworld-sample \
    -p ADMIN_USERNAME=admin -p ADMIN_PASSWORD=mypassword
```

パラメーターをファイルに保存しておいて、**--param-file** を指定して、テンプレートをインスタンス化する時にこのファイルを使用することができます。標準入力からパラメーターを読み込む場合は、以下のように**--param-file=-** を使用します。

```
$ cat helloworld.params
ADMIN_USERNAME=admin
```

```
ADMIN_PASSWORD=mypassword
$ oc new-app ruby-helloworld-sample --param-file=helloworld.params
$ cat helloworld.params | oc new-app ruby-helloworld-sample --param-file=-
```

2.2.2.4. アプリケーション作成における追加修正

new-app コマンドは、OpenShift Container Platform オブジェクトを生成します。このオブジェクトにより、作成されるアプリケーションがビルドされ、デプロイされ、実行されます。通常、これらのオブジェクトは、入力ソースリポジトリーまたはインプットイメージから派生する名前を使用して現在のプロジェクトに作成されます。ただし **new-app** を使用すると、この動作を修正できます。

new-app で作成したオブジェクトのセットは、ソースリポジトリー、イメージまたはテンプレートなどのインプットとして渡されるアーティファクトによって異なります。

表2.2 new-app 出力オブジェクト

オブジェクト	説明
BuildConfig	**BuildConfig** は、コマンドラインで指定された各ソースリポジトリーに作成されます。**BuildConfig** は使用するストラテジー、ソースのロケーション、およびビルドの出力ロケーションを指定します。
ImageStreams	**BuildConfig** では、通常 2 つの**ImageStreams** が作成されます。1 つ目は、インプットイメージを表します。**Source** ビルドの場合、これはビルダーイメージです。**Docker** ビルドでは、これは FROM イメージです。2 つ目は、アウトプットイメージを表します。コンテナーイメージが **new-app** にインプットとして指定された場合、このイメージに対してもイメージストリームが作成されます。
DeploymentConfig	**DeploymentConfig** は、ビルドの出力または指定されたイメージのいずれかをデプロイするために作成されます。**new-app** コマンドは、結果として生成される **DeploymentConfig** に含まれるコンテナーに指定されるすべての Docker ボリュームに emptyDir ボリュームを作成します。
Service	**new-app** コマンドは、インプットイメージで公開ポートを検出しようと試みます。公開されたポートで数値が最も低いものを使用して、そのポートを公開するサービスを生成します。**new-app** 完了後に別のポートを公開するには、単に **oc expose** コマンドを使用し、追加のサービスを生成するだけです。
その他	テンプレートのインスタンスを作成する際に、他のオブジェクトをテンプレートに基づいて生成できます。

2.2.2.4.1. 環境変数の指定

テンプレート、ソースまたはイメージからアプリケーションを生成する場合、**-e|--env** 引数を使用し、ランタイムに環境変数をアプリケーションコンテナーに渡すことができます。

```
$ oc new-app openshift/postgresql-92-centos7 \
    -e POSTGRESQL_USER=user \
    -e POSTGRESQL_DATABASE=db \
    -e POSTGRESQL_PASSWORD=password
```

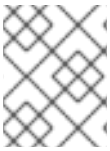
変数は、**--env-file** 引数を使用してファイルから読み取ることもできます。

```
$ cat postgresql.env
POSTGRESQL_USER=user
POSTGRESQL_DATABASE=db
POSTGRESQL_PASSWORD=password
$ oc new-app openshift/postgresql-92-centos7 --env-file=postgresql.env
```

さらに **--env-file=-** を使用することで、標準入力で環境変数を指定することもできます。

```
$ cat postgresql.env | oc new-app openshift/postgresql-92-centos7 --env-file=-
```

詳細は、「環境変数の管理」を参照してください。



注記

-e|--env または **--env-file** 引数で渡される環境変数では、**new-app** 処理の一環として作成される **BuildConfig** オブジェクトは更新されません。

2.2.2.4.2. ビルド環境変数の指定

テンプレート、ソースまたはイメージからアプリケーションを生成する場合、**--build-env** 引数を使用し、ランタイムに環境変数をビルドコンテナーに渡すことができます。

```
$ oc new-app openshift/ruby-23-centos7 \
    --build-env HTTP_PROXY=http://myproxy.net:1337/ \
    --build-env GEM_HOME=~/.gem
```

変数は、**--build-env-file** 引数を使用してファイルから読み取ることもできます。

```
$ cat ruby.env
HTTP_PROXY=http://myproxy.net:1337/
GEM_HOME=~/.gem
$ oc new-app openshift/ruby-23-centos7 --build-env-file=ruby.env
```

さらに **--build-env-file=-** を使用して、環境変数を標準入力で指定することもできます。

```
$ cat ruby.env | oc new-app openshift/ruby-23-centos7 --build-env-file=-
```

2.2.2.4.3. ラベルの指定

ソース、イメージ、またはテンプレートからアプリケーションを生成する場合、**-l|--label** 引数を使用し、作成されたオブジェクトにラベルを追加できます。ラベルを使用すると、アプリケーションに関連するオブジェクトを一括で選択、設定、削除することが簡単になります。

```
$ oc new-app https://github.com/openshift/ruby-hello-world -l name=hello-world
```

2.2.2.4.4. 作成前の出力の表示

new-app が作成する内容についてのドライランを確認するには、**yaml** または **json**の値と共に **-o|--output** 引数を使用できます。次にこの出力を使用して、作成されるオブジェクトのプレビューまたは編集可能なファイルへのリダイレクトを実行できます。問題がなければ、**oc create** を使用して OpenShift Container Platform オブジェクトを作成できます。

new-app アーティファクトをファイルに出力するには、これらを編集し、作成します。

```
$ oc new-app https://github.com/openshift/ruby-hello-world \
    -o yaml > myapp.yaml
$ vi myapp.yaml
$ oc create -f myapp.yaml
```

2.2.2.4.5. 別名でのオブジェクトの作成

通常 **new-app** で作成されるオブジェクトの名前はソースリポジトリーまたは生成に使用されたイメージに基づいて付けられます。コマンドに **--name** フラグを追加することで、生成されたオブジェクトの名前を設定できます。

```
$ oc new-app https://github.com/openshift/ruby-hello-world --name=myapp
```

2.2.2.4.6. 別のプロジェクトでのオブジェクトの作成

通常 **new-app** は現在のプロジェクトにオブジェクトを作成します。ただし、**-n|--namespace** 引数を使用して、アクセスできる別のプロジェクトにオブジェクトを作成することができます。

```
$ oc new-app https://github.com/openshift/ruby-hello-world -n myproject
```

2.2.2.4.7. 複数のオブジェクトの作成

new-app コマンドは、複数のパラメーターを **new-app** に指定して複数のアプリケーションを作成できます。コマンドラインで指定するラベルは、単一コマンドで作成されるすべてのオブジェクトに適用されます。環境変数は、ソースまたはイメージから作成されたすべてのコンポーネントに適用されます。

ソースリポジトリーおよび Docker Hub イメージからアプリケーションを作成するには、以下を実行します。

```
$ oc new-app https://github.com/openshift/ruby-hello-world mysql
```

注記

ソースコードリポジトリーおよびビルダーイメージが別個の引数として指定されている場合、**new-app** はソースコードリポジトリーのビルダーとしてそのビルダーイメージを使用します。これを意図していない場合は、~ セパレーターを使用してソースに必要なビルダーイメージを指定します。

2.2.2.4.8. 単一 Pod でのイメージとソースのグループ化

new-app コマンドにより、単一 Pod に複数のイメージをまとめてデプロイできます。イメージのグループ化を指定するには **+** セパレーターを使用します。**--group** コマンドライン引数をグループ化する必要のあるイメージを指定する際に使用することもできます。ソースリポジトリーからビルドされたイメージを別のイメージと共にグループ化するには、そのビルダーイメージをグループで指定します。

```
$ oc new-app ruby+mysql
```

ソースからビルドされたイメージと外部のイメージをまとめてデプロイするには、以下を実行します。

```
$ oc new-app \
```

```
ruby~https://github.com/openshift/ruby-hello-world \
mysql \
--group=ruby+mysql
```

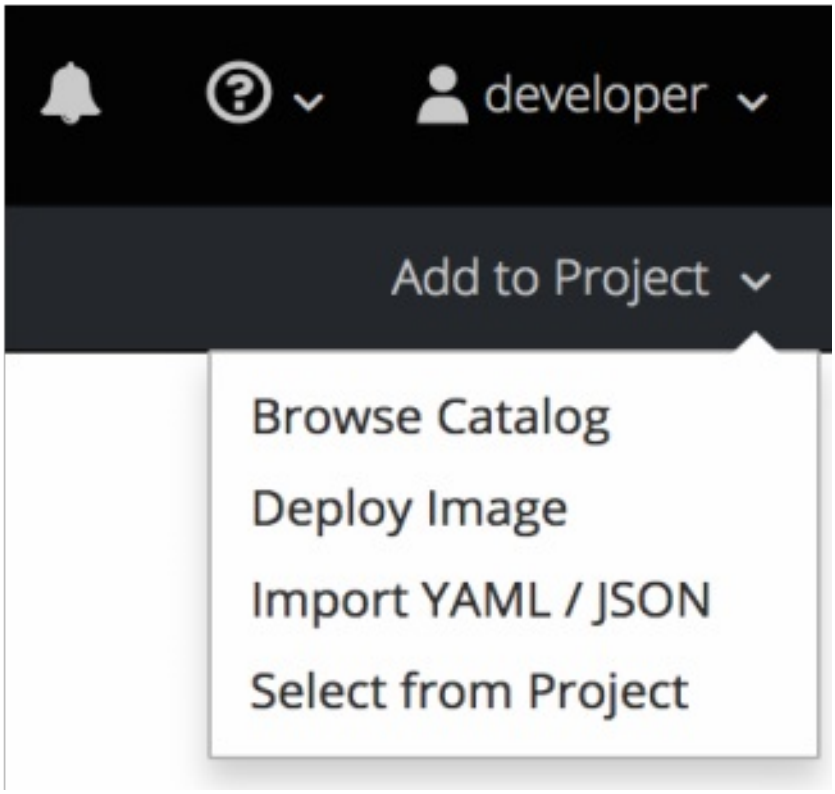
2.2.2.4.9. イメージ、テンプレート、および他の入力の検索

イメージ、テンプレート、および **oc new-app** コマンドの他の入力内容を検索するには、**--search** フラグおよび **--list** フラグを追加します。たとえば、PHP を含むすべてのイメージまたはテンプレートを検索するには、以下を実行します。

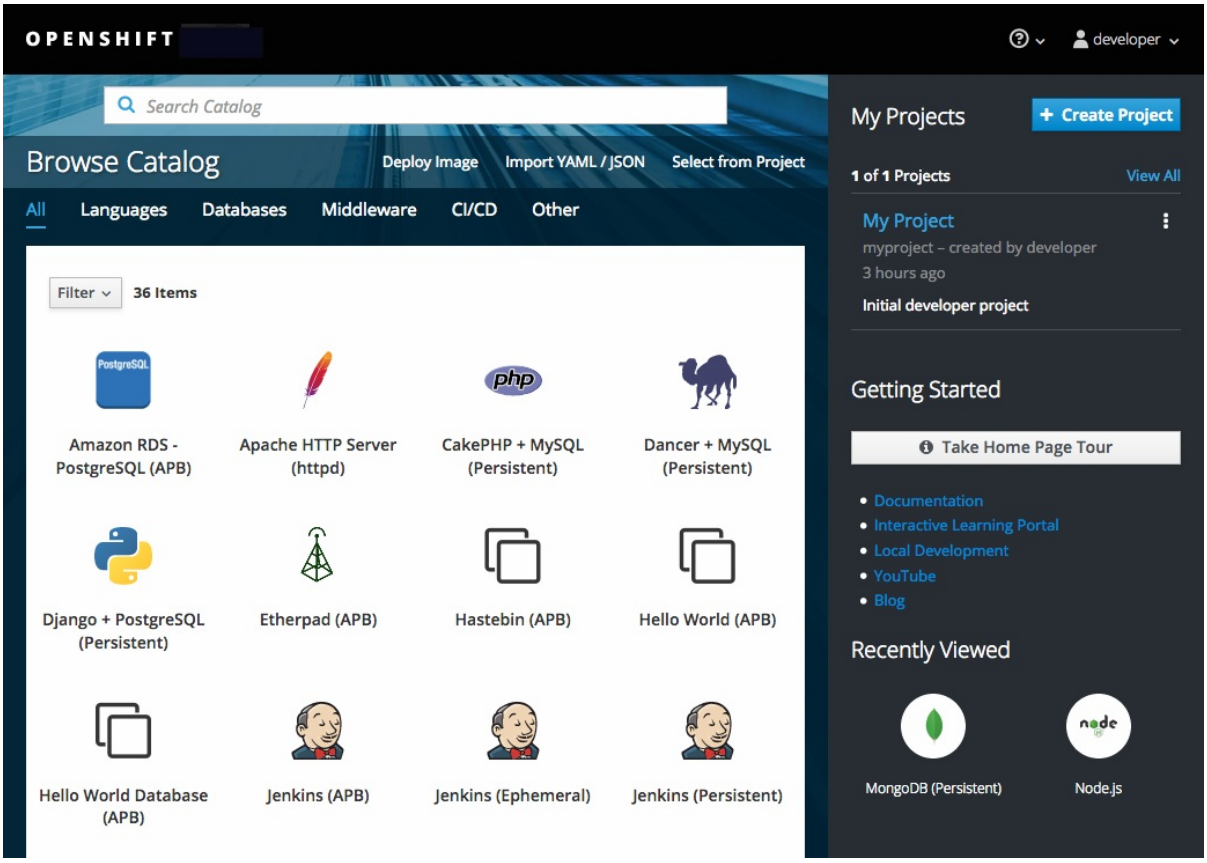
```
$ oc new-app --search php
```

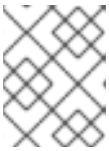
2.2.3. Web コンソールを使用したアプリケーションの作成

1. 必要なプロジェクトで **Add to Project** をクリックします。



2. プロジェクト内にあるイメージの一覧またはサービスカタログからビルダーイメージを選択します。





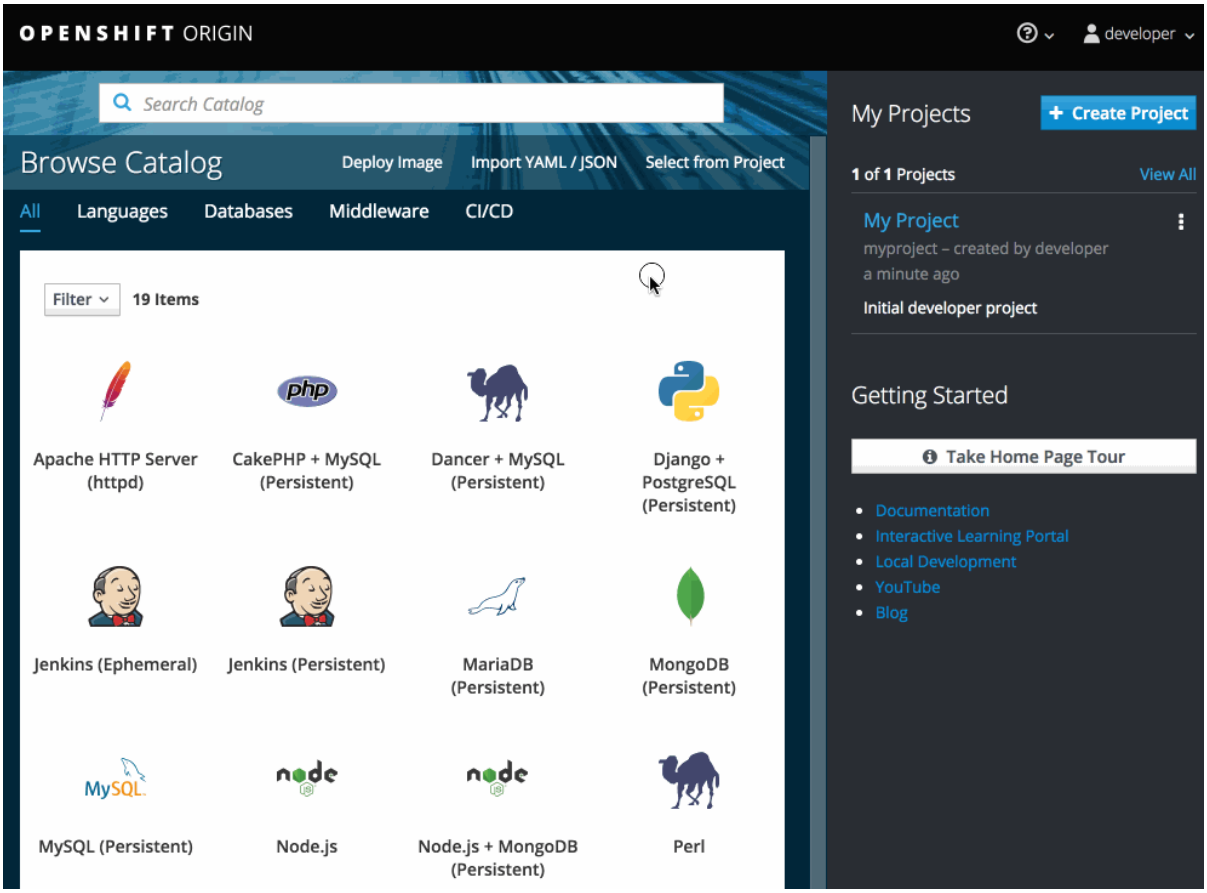
注記

以下に示すように、**builder** タグがアノテーションに一覧表示されている イメージストリームタグ のみが一覧に表示されます。

```
kind: "ImageStream"
apiVersion: "v1"
metadata:
  name: "ruby"
  creationTimestamp: null
spec:
  dockerImageRepository: "registry.access.redhat.com/openshift3/ruby-20-rhel7"
  tags:
    -
      name: "2.0"
      annotations:
        description: "Build and run Ruby 2.0 applications"
        iconClass: "icon-ruby"
        tags: "builder,ruby" (1)
        supports: "ruby:2.0,ruby"
        version: "2.0"
```

(1) ここに **builder** を含めると、この **ImageStreamTag** がビルダーとして Web コンソールに表示されます。

3. 新規アプリケーション画面で設定を変更し、オブジェクトをアプリケーションをサポートするように設定します。



2.3. 環境全体におけるアプリケーションのプロモート

2.3.1. 概要

アプリケーションのプロモーションとは、さまざまなランタイム環境でのアプリケーションの移動を意味し、通常、移動すると成熟度が増します。たとえば、あるアプリケーションが開発環境から開始され、ステージング環境へとプロモートされたあとに、さらなるテストが行われ、最後に実稼働環境へとプロモートされます。アプリケーションに変更が加えられると、変更が開発環境に加えられ、ステージング環境および実稼働環境へとプロモートされます。

「アプリケーション」は Java、Perl、Python などで記述された単なるソースコードではありません。これは、アプリケーションの言語固有のランタイムに関する静的Webコンテンツ、統合スクリプト、または関連の設定を超えたものになっています。これは、言語固有のランタイムによって使用されるアプリケーション固有のアーカイブ以上のものです。

OpenShift Container Platform および Kubernetes と Docker を統合した基盤のコンテキストでは、追加のアプリケーションのアーティファクトには以下が含まれます。

- メタデータと関連ツールの豊富なセットを含む **Docker コンテナーイメージ**。
- アプリケーションでの使用向けにコンテナーに挿入される **環境変数**。
- OpenShift Container Platform の **API オブジェクト**（リソース定義としても知られています。「Core Concepts」を参照してください）。
    - アプリケーションで使用できるようにコンテナーに挿入されます。
    - OpenShift Container Platform のコンテナーおよび Pod の管理方法を指定します。

OpenShift Container Platform でのアプリケーションのプロモート方法を検証するにあたり、以下の内容を説明します。

- アプリケーション定義に導入される新規アーティファクトについて説明する。
- アプリケーションのプロモーションパイプラインの各種環境を区別する方法を説明する。
- 新規アーティファクトを管理する方法およびツールについて説明する。
- 各種の概念、構成、方法およびツール、アプリケーションのプロモートなど内容に該当する実例を紹介する。

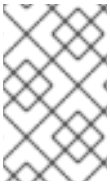
2.3.2. アプリケーションコンポーネント

2.3.2.1. API オブジェクト

OpenShift Container Platform および Kubernetes リソース定義（アプリケーションインベントリーに新規に導入された項目）に関連して、アプリケーションのプロモートについて検討する場合に API オブジェクトの設計ポイントで留意しておくべき 2 つの主要な点があります。

1 つ目の点として、すべての API オブジェクトは、OpenShift Container Platform ドキュメント全体で強調されているように JSON または YAML のいずれかで表現できるので、これらのリソース定義は従来のソースコントロールおよびスクリプトを使用して容易に管理できます。

また、API オブジェクトは、システムの必要な状態を指定するオブジェクトの部分とシステムのステータスまたは現在の状態を反映する部分で構成されるように設計されています。これはインプットおよびアウトプットとして考えることができます。インプット部分は JSON または YAML で表現され、ソースコントロール管理(SCM)のアーティファクトとして適合します。



注記

API オブジェクトのインプット部分または仕様部分は、インスタンス化のタイミングでテンプレート処理による変数置換 が可能であるため、完全に静的または動的に機能する点に留意してください。

API オブジェクトに関する上記の点により、JSON または YAML ファイルの表現を使ってアプリケーションの設定をコードとして処理できます。

ほぼすべての API オブジェクトについて、組織はこれらをアプリケーションのアーティファクトとみなすことができます。以下は、アプリケーションのデプロイおよび管理に最も関連するオブジェクトです。

BuildConfigs

これはアプリケーションのプロモーションのコンテキストにおける特殊なリソースです。**BuildConfig** はとくに開発者の観点ではアプリケーションの一部ではありますが、**BuildConfig** は通常パイプラインでプロモートされません。これはパイプラインで（他のアイテムと共に）プロモートされる **イメージ** を作成します。

テンプレート

アプリケーションのプロモーションの観点では、**Templates** はとくにパラメーター化機能を使ってリソースを所定のステージング環境でセットアップするための開始点として機能します。ただしアプリケーションがプロモーションのパイプラインを通過する場合でも、インスタンス化の後に追加の変更が生じる可能性が高くなります。詳細は、「シナリオおよび実例」を参照してください。

ルート

ルートは、最も一般的なリソースで、アプリケーションのさまざまなステージに対するテストとは、**Route** を使用してアプリケーションにアクセスするので、アプリケーションのプロモーションパイプラインのステージごとに異なります。また、ホスト名だけでなく **Route** の HTTP レベルのセキュリティーに関しても、手動指定や自動生成のオプションがある点に留意してください。

サービス

初期ステージでの個々の開発者の便宜を考慮する場合など、所定のアプリケーションプロモーションステージで **ルーター** および **ルート** を避ける理由がある場合には、アプリケーションは **クラスター** の IP アドレスおよびポート経由でアクセスできます。これらを使用した場合、ステージ間のアドレスおよびポートの管理の一部が必要となる可能性があります。

Endpoints (エンドポイント)

アプリケーションレベルのサービス（多くの企業によっては、データベースのインスタンスなど）は OpenShift Container Platform で管理されない場合があります。そのような場合に、独自に **エンドポイント** を作成して、関連する **サービス** （**サービス** のセレクターフィールドを省略）に必要な修正を加えると、（環境をどのようにプランニングするかにより異なりますが）アクティビティーがステージ間で重複または共有されます。

シークレット

シークレット でカプセル化された機密情報は、その情報関連の対応するエンティティー（OpenShift Container Platform が管理する **サービス** または OpenShift Container Platform 外で管理する外部サービス）が共有されると、ステージ環境間で共有されます。このエンティティーの異なるバージョンがアプリケーションのプロモーションパイプラインの各ステージにある場合には、パイプラインの各ステージで固有の **Secret** を維持するか、パイプラインを通過する際に変更を加える必要がある場合があります。また **Secret** を SCM に JSON または YAML として保存する場合には、機密情報を保護するための暗号化フォームが必要となる場合があります。

DeploymentConfigs

このオブジェクトは、アプリケーションの起動方法を制御するため、所定のアプリケーションのプロモーションパイプラインステージの環境を定義し、そのスコープを設定する時の最も重要なリソースになります。各種ステージ間で共通する部分がありますが、アプリケーションプロモーションパイプラインの移動に伴い、このオブジェクトには当然変更が加えられます。この変更には、各ステージの環境の違いを反映させるための修正や、アプリケーションがサポートする必要のある各種シナリオのテストを容易にするためのシステム動作の変更が含まれます。

ImageStreams, ImageStreamTags、および ImageStreamImage

イメージ および Image Streams の各セクションで説明されているように、これらのオブジェクトは、コンテナーイメージの管理に関連して OpenShift Container Platform の追加要素の中核となります。

ServiceAccounts および RoleBindings

アプリケーション管理において、OpenShift Container Platform や外部サービスでの他の API オブジェクトに対するパーミッション管理は必要不可欠です。**Secrets** と同様に、**ServiceAccounts** および **RoleBindings** オブジェクトのアプリケーションプロモーションパイプラインのステージ間での共有方法は、各種環境を共有または分離する必要性によって異なる可能性があります。

PersistentVolumeClaims

データベースのようなステートフルなサービスに関連して、どの程度異なるアプリケーションプロモーションステージ間で共有されるかは、組織がアプリケーションデータのコピーを共有または分離する方法に直接関係します。

ConfigMap

Pod 設定の **Pod** 自体から分離（環境変数スタイルの設定など）するのに便利です。これらは **Pod** の動作に一貫性をもたせる必要がある場合などに各種のステージング環境で共有することができます。また、これらはステージ間で変更して **Pod** 動作を修正することもできます（通常はアプリケーションのさまざまな側面はステージごとに検証されます）。

2.3.2.2. イメージ

前述のように、コンテナーイメージはアプリケーションのアーティファクトです。実際、新しいアプリケーションのアーティファクト、イメージ、およびイメージの管理は、アプリケーションのプロモーションに関する主要な要素です。場合によっては、イメージがアプリケーションの全体をカプセル化し、アプリケーションプロモーションフローがイメージの管理のみで構成されることがあります。

通常イメージは SCM システムでは管理されません (アプリケーションのバイナリーが以前のシステムで管理されていなかったのと同様です)。ただしバイナリーと同様に、インストール可能なアーティファクトおよび対応するリポジトリー (RPM、RPM リポジトリー、Nexus など) は SCM と同様のセマンティクスで生成されるので、SCM に似たイメージ管理の構成および専門用語が導入されました。

- Image registry == SCM server
- Image repository == SCM repository

イメージはレジストリーに存在するので、アプリケーションプロモーションでは、適切なイメージがレジストリーに存在し、そのイメージで表されるアプリケーションを実行する必要のある環境からアクセスできるようにします。

イメージを直接参照するよりも、アプリケーションの定義は通常イメージストリームに参照を抽象化します。これは、イメージストリームがアプリケーションコンポーネントを構成する別の API オブジェクトになることを意味します。イメージストリームの詳細は、「Core Concepts」を参照してください。

2.3.2.3. 概要

これまでノート、イメージ、および API オブジェクトのアプリケーションのアーティファクトについて OpenShift Container Platform 内のアプリケーションプロモーションのコンテキストで説明しました。次は、アプリケーションをプロモーションパイプラインの各種ステージの **どこで** 実行するのかを見ていきます。

2.3.3. デプロイメント環境

このコンテキストでのデプロイメント環境は、CI/CD パイプラインの特定ステージでアプリケーションが実行される固有のスペースを表します。通常の環境には、**開発**、**テスト**、**ステージング** および **実稼働環境** などが含まれます。環境の境界については、以下のように様々な方法で定義できます。

- 単一プロジェクト内のラベルおよび独自の名前を使用する
- クラスター内の固有のプロジェクトを使用する
- 固有のクラスターを使用する

上記の 3 つ方法すべてを利用できることが想定されます。

2.3.3.1. 留意事項

通常デプロイメント環境の構成を検討する際は、以下のヒューリスティックな側面について検討します。

- プロモーションフローの各種ステージで許可するリソース共有の度合い
- プロモーションフローの各種ステージで必要な分離の度合い
- プロモーションフローの各種ステージの中心からの位置 (またはどの程度地理的に分散しているか)

さらに OpenShift Container Platform のクラスターおよびプロジェクトがイメージレジストリーにどのように関係するかについて以下の重要な点に留意してください。

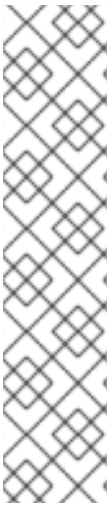
- 同一クラスター内の複数のプロジェクトは同一のイメージストリームにアクセスできる。
- 複数のクラスターが同一の外部レジストリーにアクセスできる。
- OpenShift Container Platform の内部イメージレジストリーがルート経由で公開される場合、クラスターはレジストリーのみを共有できる。

2.3.3.2. 概要

デプロイメント環境が定義された後、パイプライン内のステージの記述を含むプロモーションフローを実装できます。以下では、これらのプロモーションフローの実装を構成する方法およびツールについて説明します。

2.3.4. 方法およびツール

基本的にアプリケーションのプロモートとは、前述のアプリケーションのコンポーネントをある環境から別の環境に移動するプロセスのことです。アプリケーションのプロモートの自動化に関する全体的なソリューションを検討する前に、各種コンポーネントを手動で移動する場合に使用できるツールの概要について以下のサブセクションで見ていきましょう。



注記

ビルドおよびデプロイメントの両方のプロセスにおいて多数の挿入ポイントを利用できます。これらは **BuildConfig** および **DeploymentConfig** API オブジェクトで定義されます。これらのフックにより、データベースなどのデプロイされたコンポーネントおよび OpenShift Container Platform クラスター自体と対話できるカスタムスクリプトの呼び出しが可能となります。

したがって、フック内からイメージタグ操作を実行するなど、このようなフックを使用して、アプリケーションを環境間で効果的に移動するコンポーネント管理操作を実行できます。ただし、これらのフックポイントの使用は、環境間でアプリケーションコンポーネントを移動する場合よりも、所定の環境でアプリケーションのライフサイクル管理を行う場合に適しています (アプリケーションの新バージョンがデプロイされる際のデータベーススキーマの移行に使用するなど)。

2.3.4.1. API オブジェクトの管理

1 つの環境で定義されるリソースは、新しい環境へのインポートに備えて JSON または YAML ファイルの内容としてエクスポートされます。したがって JSON または YAML としての API オブジェクトの表現は、アプリケーションパイプラインで API オブジェクトをプロモートする際の作業単位として機能します。このコンテンツのエクスポートやインポートには **oc** CLI を使用します。

ヒント

OpenShift Container Platform のプロモーションフローには必要ないですが、JSON または YAML はファイルに保存されるので、SCM システムを使用したコンテンツの保存や取得について検討することができます。これにより、ブランチの作成、バージョンに関連する各種ラベルやタグの割り当てやクエリーなど、SCM のバージョン関連の機能を活用できるようになります。

2.3.4.1.1. API オブジェクトステートのエクスポート

API オブジェクトの仕様は、**oc export** で取り込む必要があります。この操作は、オブジェクト定義か

ら環境に固有のデータを取り除き (現在の namespace または割り当てられた IP アドレスなど)、異なる環境で再作成できるようにします (オブジェクトのフィルターされていないステートを出力する **oc get** 操作とは異なります)。

oc label を使用すると、API オブジェクトに対するラベルの追加、変更、または削除が可能になり、ラベルがあれば、操作 1 回で Pod のグループの選択や管理ができるので、プロモーションフロー用に収集されたオブジェクトを整理するのに有用であることが分かります。oc label を使用すると、適切なオブジェクトをエクスポートするのが簡単になります。 また、オブジェクトが新しい環境で作成された場合にラベルが継承されるので、各環境のアプリケーションコンポーネントの管理も簡素化されます。

注記

API オブジェクトには、**Secret** を参照する **DeploymentConfig** などの参照が含まれることがよくあります。API オブジェクトをある環境から別の環境へと移動する際、これらの参照も新しい環境へと移動することを確認する必要があります。

同様に **DeploymentConfig** などの API オブジェクトには、外部レジストリーを参照する **ImageStreams** の参照が含まれることがよくあります。API オブジェクトをある環境から別の環境へと移動する際、このような参照が新しい環境内で解決可能であることを確認する必要があります。つまり、参照が解決可能であり、**ImageStream** は新しい環境でアクセス可能なレジストリーを参照できる必要があります。詳細については、「イメージの移動」および「プロモートの注意事項」を参照してください。

2.3.4.1.2. API オブジェクトステートのインポート

2.3.4.1.2.1. 初期作成

アプリケーションを新しい環境に初めて導入する場合は、API オブジェクトの仕様を表現する JSON または YAML を使用し、**oc create** を実行して適切な環境で作成するだけで十分です。**oc create** を使用する場合、**--save-config** オプションに留意してください。アノテーション一覧にオブジェクトの設定要素を保存しておくことで、後の **oc apply** を使用したオブジェクトの変更が容易になります。

2.3.4.1.2.2. 反復修正

各種のステージング環境が最初に確立されると、プロモートサイクルが開始し、アプリケーションがステージからステージへと移動します。アプリケーションの更新には、アプリケーションの一部である API オブジェクトの修正を含めることができます。API オブジェクトはOpenShift Container Platform システムの設定を表すことから、それらの変更が想定されます。 それらの変更の目的として以下のケースが想定されます。

- ステージング環境間における環境の違いについて説明する。
- アプリケーションがサポートする各種シナリオを検証する。

oc CLI を使用することで、API オブジェクトの次のステージ環境への移行が実行されます。API オブジェクトを変更する **oc** コマンドセットは充実していますが、本トピックではオブジェクト間の差分を計算し、適用する **oc apply** に焦点を当てます。

とりわけ **oc apply** は既存のオブジェクト定義と共にファイルまたは標準入力 (stdin) を入力として取る 3 方向マージと見ることができます。以下の間で 3 方向マージを実行します。

1. コマンドへの入力
2. オブジェクトの現行バージョン
3. 現行オブジェクトにアノテーションとして保存された最新のユーザー指定オブジェクト定義

その後に既存のオブジェクトは結果と共に更新されます。

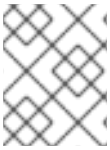
オブジェクトがソース環境とターゲット環境間で同一であることが予期されていない場合など、API オブジェクトの追加のカスタマイズが必要な場合に、**oc set** などの **oc** コマンドは、アップストリーム環境から最新のオブジェクト定義を適用した後に、オブジェクトを変更するために使用できます。

使用方法についての詳細は、「シナリオおよび実例」を参照してください。

2.3.4.2. イメージおよびイメージストリームの管理

OpenShift Container Platform のイメージも一連の API オブジェクトで管理されます。ただし、イメージの管理はアプリケーションのプロモートにおける非常に中心的な部分であるため、イメージに最も直接的に関係するツールおよび API オブジェクトについては別途扱います。イメージのプロモートの管理には、手動および自動の方法を使用できます (パイプラインによるイメージの伝搬) 。

2.3.4.2.1. イメージの移動



注記

イメージの管理に関する注意事項すべての詳細については、「イメージの管理」のトピックを参照してください。

2.3.4.2.1.1. ステージング環境がレジストリーを共有する場合

ステージング環境が同じ OpenShift Container Platform レジストリーを共有する場合 (すべてが同じ OpenShift Container Platform クラスター上にある場合など)、アプリケーションのプロモートパイプラインのステージ間でイメージを **移動する** 基本的な方法として、以下の 2 つ操作を実行できます。

1. 1 つ目は、 **docker tag** および **git tag** と類似する **oc tag** コマンドにより、OpenShift Container Platform のイメージストリームを特定のイメージへの参照で更新できます。また、あるイメージストリームから別のイメージストリームへとイメージの特定のバージョンへの参照をコピーすることも可能で、クラスター内の複数の異なるプロジェクト全体でもコピーが可能です。

2. 2 つ目として、**oc import-image** は外部レジストリーとイメージストリーム間の橋渡しの機能を持ちます。レジストリーから所定のイメージのメタデータをインポートし、これを イメージストリームタグ としてイメージストリームに保存します。プロジェクトの各種の **BuildConfigs** および **DeploymentConfigs** がこれらの特定のイメージを参照できます。

2.3.4.2.1.2. ステージング環境が異なるレジストリーを使用する場合

ステージング環境が異なる OpenShift Container Platform レジストリーを活用している場合、より高度な使用方法が見られます。内部レジストリーへのアクセス で、手順を詳細に説明していますが、まとめると以下のようになります。

1. OpenShift Container Platform のアクセストークンの取得と関連して **docker** コマンドを使用し、**docker login** コマンドに指定します。

2. OpenShift Container Platform レジストリーにログインした後、**docker pull**、**docker tag** および **docker push** を使用してイメージを移行します。

3. イメージがパイプラインの次の環境のレジストリーで利用可能になってから、必要に応じて **oc tag** を使用してイメージストリームを設定します。

2.3.4.2.2. デプロイ

変更対象が基礎となるアプリケーションイメージであるか、アプリケーションを設定する API オブジェクトであるかを問わず、プロモートされた変更を認識するにはデプロイメントが通常必要になります。アプリケーションのイメージが変更される場合 (アップストリームからのイメージのプロモートの一環としての **oc tag** 操作または **docker push** の実行による場合など)、**DeploymentConfig** の **ImageChangeTriggers** が新規デプロイメントをトリガーできます。同様に **DeploymentConfig** API オブジェクト自体が変更されている場合、API オブジェクトがプロモーション手順によって更新されると (例: **oc apply**)、**ConfigChangeTrigger** がデプロイメントを開始できます。

それ以外の場合に、手動のデプロイメントを容易にする **oc** コマンドには以下が含まれます。

- **oc rollout**: デプロイメント管理の新しいアプローチです (停止と再開のセマンティクスおよび履歴管理に関する充実した機能を含む)。
- **oc rollback**: 以前のデプロイメントに戻すことができます。 プロモーションのシナリオでは、新しいバージョンのテストで問題が発生した場合には、以前のバージョンで問題がないかどうかを確認する必要がある場合があります。

2.3.4.2.3. Jenkins でのプロモーションフローの自動化

アプリケーションをプロモートする際に環境間での移動が必要なアプリケーションのコンポーネントを理解し、コンポーネントを移動する際に必要な手順を理解した後に、ワークフローのオーケストレーションおよび自動化を開始できます。OpenShift Container Platform は、このプロセスで役立つ Jenkins イメージおよびプラグインを提供しています。

OpenShift Container Platform Jenkins のイメージについては、Using Images で詳細に説明されています。これには Jenkins と Jenkins パイプラインの統合を容易にする OpenShift Container Platform プラグインのセットも含まれます。また、パイプラインビルドストラテジー により、Jenkins Pipeline と OpenShift Container Platform との統合が容易になります。また、パイプラインビルドストラテジー により、Jenkins Pipeline と OpenShift Container Platform との統合が容易になります。 これらすべてはアプリケーションのプロモートを含む、CI/CD の様々な側面の有効化に焦点を当てています。

アプリケーションのプロモート手順の手動による実行から自動へと切り替える際には、以下の OpenShift Container Platform が提供する Jenkins 関連の機能に留意してください。

- OpenShift Container Platform は、OpenShift Container Platform クラスターでのデプロイメントを非常に容易なものとするために高度にカスタマイズされた Jenkins のイメージを提供します。
- Jenkins イメージには OpenShift Pipeline プラグインが含まれます。これはプロモーションワークフローを実装する構成要素を提供します。 これらの構成要素には、イメージストリームの変更に伴う Jenkins ジョブのトリガーやそれらのジョブ内でのビルドおよびデプロイメントのトリガーも含まれます。
- OpenShift Container Platform の Jenkins Pipeline のビルドストラテジーを使用する **BuildConfigs** により、Jenkinsfile ベースの Jenkins Pipeline ジョブの実行が可能になります。パイプラインジョブは Jenkins における複雑なプロモーションフロー用の戦略を構成するものであり、OpenShift Pipeline プラグインにより提供される手順を利用できます。

2.3.4.2.4. プロモーションについての注意事項

2.3.4.2.4.1. API オブジェクト参照

API オブジェクトは他のオブジェクトを参照することができます。この一般的な使用方法として、イメージストリームを参照する**DeploymentConfig** を設定します (他の参照関係も存在する場合があります)。

ある環境から別の環境へと API オブジェクトをコピーする場合、すべての参照がターゲット環境内で解決できることが重要となります。以下のような参照のシナリオを見てみましょう。

- プロジェクトに「ローカル」から参照している場合。この場合、参照オブジェクトは、プロジェクトを参照しているオブジェクトと同じプロジェクトに存在します。通常の方法として、参照しているオブジェクトと同じプロジェクト内にあるターゲット環境に参照オブジェクトをコピーできることを確認します。

- 他のプロジェクトのオブジェクトを参照する場合。これは、共有プロジェクトのイメージストリームが複数のアプリケーションプロジェクトによって使用されている場合によくあるケースです (「イメージの管理」を参照してください)。この場合、参照するオブジェクトを新しい環境にコピーする際、ターゲット環境内で解決できるように参照を随時更新しなければなりません。以下が必要になる場合があります。

    - 共有されるプロジェクトの名前がターゲット環境では異なる場合、参照先のプロジェクトを変更する。

    - 参照されるオブジェクトを共有プロジェクトからターゲット環境のローカルプロジェクトへと移動し、主要オブジェクトをターゲット環境へと移動する際に参照をローカルルプロジェクトをポイントするよう更新する。

    - 参照されるオブジェクトのターゲット環境へのコピーおよびその参照の更新の他の組み合わせ。

通常は、新しい環境にコピーされるオブジェクトによって参照されるオブジェクトを確認し、参照がターゲット環境で解決可能であることを確認することをお勧めします。それ以外には、参照の修正を行うための適切なアクションを取り、ターゲット環境で参照されるオブジェクトを利用可能にすることができます。

2.3.4.2.4.2. イメージレジストリー参照

イメージストリームはイメージレポジトリーを参照してそれらが表すイメージのソースを示唆します。イメージストリームがある環境から別の環境へと移動する場合、レジストリーおよびレポジトリーの参照も変更すべきかどうかを検討することが重要です。

- テスト環境と実稼働環境間の分離をアサートするために異なるイメージレジストリーが使用されている場合。

- テスト環境および実稼働環境に対応したイメージを分離するために異なるイメージレポジトリーが使用されている場合。

上記のいずれかが該当する場合、イメージストリームはソース環境からターゲット環境にコピーされる際に、適切なイメージに対して解決されるよう変更される必要があります。これは、あるレジストリーおよびレポジトリーから別のレジストリーおよびレポジトリーへとイメージをコピーするという シナリオおよび実例に説明されている手順の追加として行われます。

2.3.4.3. 概要

現時点で、以下が定義されています。

- デプロイされたアプリケーションを構成する新規アプリケーションアーティファクト。

- アプリケーションのプロモーションアクティビティーと OpenShift Container Platform によって提供されるツールおよびコンセプトとの相関関係。

- OpenShift Container Platform と CI/CD パイプラインエンジン Jenkins との統合。

このトピックにおける残りの部分では、OpenShift Container Platform 内のアプリケーションのプロモーションフローのいくつかの例について扱います。

2.3.5. シナリオおよび実例

Docker、Kubernetes および OpenShift Container Platform のエコシステムにより導入された新規アプリケーションアーティファクトのコンポーネントを定義した上に、このセクションでは OpenShift Container Platform によって提供される方法およびツールを使用してこれらのコンポーネントを環境間でプロモートする方法を説明します。

アプリケーションを構成するコンポーネントにおいて、イメージは主要なアーティファクトです。これを前提とし、かつアプリケーションのプロモーションに当てはめると、中心的なアプリケーションのプロモーションパターンとなるのがイメージのプロモーションであり、この場合にイメージが作業単位となります。ほとんどのアプリケーションプロモーションシナリオでは、プロモーションパイプラインを使用したイメージの管理および伝搬が行われます。

単純なシナリオでは、パイプラインを使用したイメージの管理および伝搬のみを扱います。プロモーションシナリオの対象範囲が広がるにつれ、API オブジェクトを筆頭とする他のアプリケーションアーティファクトがパイプラインで管理および伝搬されるアイテムのインベントリーに含まれます。

このトピックでは、手動および自動の両方のアプローチを使用して、イメージおよび API オブジェクトのプロモートに関する特定の実例をいくつか紹介します。最初にアプリケーションのプロモーションパイプラインの環境のセットアップに関して、以下の点に留意してください。

2.3.5.1. プロモーションのセットアップ

アプリケーションの初期リビジョンの開発が完了すると、次の手順として、プロモーションパイプラインのステージング環境に移行できるようにアプリケーションのコンテンツをパッケージ化します。

1. 最初に、表示されるすべての API オブジェクトを移行可能なものとしてグループ化し、共通の **label** を適用します。

```
labels:
  promotion-group: <application_name>
```

前述のように **oc label** コマンドは、さまざまな API オブジェクトのラベルの管理を容易にします。

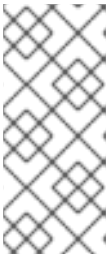
ヒント

OpenShift Container Platform テンプレートに API オブジェクトを最初に定義する場合、プロモート用にエクスポートする際にクエリーに使用する共通のラベルがすべての関連するオブジェクトにあることを簡単に確認できます。

2. このラベルは後続のクエリーで使用できます。たとえば、アプリケーションの API オブジェクトの移行を行う以下の **oc** コマンドセットの呼び出しについて検討しましょう。

```
$ oc login <source_environment>
$ oc project <source_project>
$ oc export dc,is,svc,route,secret,sa -l promotion-group=<application_name> -o yaml >
export.yaml
$ oc login <target_environment>
$ oc new-project <target_project> 1
$ oc create -f export.yaml
```

1 または、すでに存在している場合は **oc project <target_project>** を実行します。



注記

oc export コマンドでは、イメージストリーム用に **is** タイプを含めるかどうかは、パイプライン内の異なる環境全体でイメージ、イメージストリーム、およびレジストリーの管理方法をどのように選択するかによって変わってきます。この点に関する注意事項を以下で説明しています。イメージの管理 のトピックも参照してください。

3. プロモーションパイプラインの各種のステージング環境で使用されるそれぞれのレジストリーに対して機能するトークンを取得する必要があります。各環境について以下を実行します。

   a. 環境にログインします。

   ```
   $ oc login <each_environment_with_a_unique_registry>
   ```

   b. 以下を実行してアクセストークンを取得します。

   ```
   $ oc whoami -t
   ```

   c. 次回に使用できるようにトークン値をコピーアンドペーストします。

2.3.5.2. 繰り返し可能なプロモーションプロセス

パイプラインの異なるステージング環境での初回のセットアップ後に、プロモーションパイプラインを使用したアプリケーションの反復を検証する繰り返し可能な手順のセットを開始できます。これらの基本的な手順は、ソース環境のイメージまたは API オブジェクトが変更されるたびに実行されます。

更新後のイメージの移動→更新後の API オブジェクトの移動→環境固有のカスタマイズの適用

1. 通常、最初の手順ではアプリケーションに関連するイメージの更新をパイプラインの次のステージにプロモートします。前述のように、ステージング環境間で OpenShift Container Platform レジストリーが共有されるかどうかが、イメージをプロモートする上での主要な差別化要因となります。

   a. レジストリーが共有されている場合、単に **oc tag** を使用します。

   ```
   $ oc tag <project_for_stage_N>/<imagestream_name_for_stage_N>:<tag_for_stage_N>
   <project_for_stage_N+1>/<imagestream_name_for_stage_N+1>:<tag_for_stage_N+1>
   ```

   b. レジストリーが共有されていない場合、ソースおよび宛先の両方のレジストリーにログインする際、各プロモーションパイプラインレジストリーに対してアクセストークンを使用でき、アプリケーションイメージのプル、タグ付け、およびプッシュを随時実行できます。

      i. ソース環境レジストリーにログインします。

      ```
      $ docker login -u <username> -e <any_email_address> -p <token_value>
      <src_env_registry_ip>:<port>
      ```

      ii. アプリケーションのイメージをプルします。

```
$ docker pull <src_env_registry_ip>:<port>/<namespace>/<image name>:<tag>
```

iii. アプリケーションのイメージを宛先レジストリーの場所にタグ付けし、宛先ステージング環境と一致するように namespace、名前、タグを随時更新します。

```
$ docker tag <src_env_registry_ip>:<port>/<namespace>/<image name>:<tag>
<dest_env_registry_ip>:<port>/<namespace>/<image name>:<tag>
```

iv. 宛先ステージング環境レジストリーにログインします。

```
$ docker login -u <username> -e <any_email_address> -p <token_value>
<dest_env_registry_ip>:<port>
```

v. イメージを宛先にプッシュします。

```
$ docker push <dest_env_registry_ip>:<port>/<namespace>/<image name>:<tag>
```

ヒント

外部レジストリーからイメージの新バージョンを自動的にインポートするために、**oc tag** コマンドで **--scheduled** オプションを使用できます。これを使用する場合、**ImageStreamTag** が参照するイメージは、イメージをホストするレジストリーから定期的にプルされます。

2. 次に、アプリケーションの変化によってアプリケーションを構成する API オブジェクトの根本的な変更や API オブジェクトセットへの追加と削除が必要となるケースがあります。アプリケーションの API オブジェクトにこのような変化が生じると、OpenShift Container Platform CLI はあるステージング環境から次の環境へと変更を移行するための広範囲のオプションを提供します。

a. プロモーションパイプラインの初回セットアップ時と同じ方法で開始します。

```
$ oc login <source_environment>
$ oc project <source_project>
$ oc export dc,is,svc,route,secret,sa -l promotion-group=<application_name> -o yaml >
export.yaml
$ oc login <target_environment>
$ oc <target_project>
```

b. 単に新しい環境でリソースを作成するのではなく、それらを更新します。これを実行するための方法がいくつかあります。

i. より保守的なアプローチとして、**oc apply** を使用し、ターゲット環境内の各 API オブジェクトに新しい変更をマージできます。これを実行することにより、**--dry-run=true** オプションを実行し、オブジェクトを実際に変更する前に結果として得られるオブジェクトを確認することができます。

```
$ oc apply -f export.yaml --dry-run=true
```

問題がなければ、**apply** コマンドを実際に実行します。

```
$ oc apply -f export.yaml
```

apply コマンドはより複雑なシナリオで役立つ追加の引数をオプションで取ります。詳細については **oc apply --help** を参照してください。

ii. または、よりシンプルで積極的なアプローチとして、**oc replace** を使用できます。この更新および置換についてはドライランは利用できません。最も基本的な形式として、以下を実行できます。

```
$ oc replace -f export.yaml
```

apply と同様に、**replace** はより高度な動作については他の引数をオプションで取ります。詳細は、**oc replace --help** を参照してください。

3. 直前の手順では、導入された新しい API オブジェクトは自動的に処理されますが、API オブジェクトがソースの環境から削除された場合には、 **oc delete** を使用してこれらをターゲットの環境から手動で削除する必要があります。

4. ステージング環境ごとに必要な値が異なる可能性があるため、API オブジェクトで引用された環境変数を調整する必要がある場合があります。この場合は **oc set env** を使用します。

```
$ oc set env <api_object_type>/<api_object_ID> <env_var_name>=<env_var_value>
```

5. 最後に、**oc rollout** コマンドまたは、上記の「デプロイメント」のセクションで説明した他のメカニズムを使用して、更新したアプリケーションの新規デプロイメントをトリガーします。

2.3.5.3. Jenkins を使用した反復可能なプロモーションプロセス

OpenShift Container Platform の Jenkins Docker イメージ で定義された OpenShift サンプル ジョブは、Jenkins 構成ベースの OpenShift Container Platform でのイメージのプロモーションの例です。このサンプルのセットアップは OpenShift Origin ソースリポジトリー にあります。

このサンプルには以下が含まれます。

- **CI/CD エンジンとして Jenkins** の使用。
- **OpenShift Pipeline plug-in for Jenkins**の使用。このプラグインでは、Jenkins Freestyle および DSL Job ステップとしてパッケージされた OpenShift Container Platform の **oc** CLI が提供する機能サブセットを提供します。**oc** バイナリーは、OpenShift Container Platform 用の Jenkins Docker イメージにも含まれており、Jenkins ジョブで OpenShift Container Platform と対話するために使用することも可能です。
- OpenShift Container Platform が提供する **Jenkins のテンプレート**。一時ストレージおよび永続ストレージの両方のテンプレートがあります。
- **サンプルアプリケーション**: OpenShift Origin ソースリポジトリー で定義されます。 このアプリケーションは **ImageStreams**、**ImageChangeTriggers**、**ImageStreamTags**、**BuildConfigs** およびプロモーションパイプラインの各種ステージに対応した別個の **DeploymentConfigs** と **Services** を利用します。

以下では、OpenShift のサンプルジョブを詳細に検証していきます。

1. 最初のステップ は、**oc scale dc frontend --replicas=0** の呼び出しと同じです。この手順は、実行されている可能性のあるアプリケーションイメージの以前のバージョンを終了させるために実行されます。

2. 2 番目のステップ は **oc start-build frontend** の呼び出しと同じです。

3. 3 番目のステップ は **oc rollout latest dc/frontend** の呼び出しと同じです。

4. 4 番目のステップ は、このサンプルの「テスト」を行います。このステップでは、アプリケーションに関連するサービスがネットワークからアクセス可能であることを確認します。背後で、OpenShift Container Platform サービスに関連する IP アドレスやポートにソケット接続を試みます。当然のこととして別のテストを追加することも可能です (OpenShift Pipepline plug-in ステップを使用しない場合は、Jenkins Shell ステップを使用して、OS レベルのコマンドとスクリプトを使用してアプリケーションをテストします)。

5. 5 番目のステップ は、アプリケーションがテストに合格したことを前提としているため、イメージは「Ready (使用準備完了)」としてマークされます。このステップでは、新規の **prod** タグが **最新の** イメージをベースにしたアプリケーションイメージ用に作成されます。**フロントエンド** の **DeploymentConfig** でそのタグに対して **ImageChangeTrigger** が 定義されている場合には、対応する「実稼働」デプロイメントが起動されます。

6. 6 番目と最後のステップ は検証のステップで、プラグインは OpenShift Container Platform が「実稼働」デプロイメントの必要な数のレプリカを起動したことを確認します。

第3章 認証

3.1. WEB コンソール認証

ブラウザーで <master_public_addr>:8443 の**Web コンソール** にアクセスすると、自動的にログインページにリダイレクトされます。

ブラウザーのバージョンとオペレーティングシステム を使用して、Web コンソールにアクセスできることを確認します。

このページでログイン認証情報を入力して、API 呼び出しを行うためのトークンを取得します。ログイン後、Webコンソールを使用してプロジェクトをナビゲートできます。

3.2. CLI 認証

CLI コマンドの **oc login** を使用して、コマンドラインで認証することができます。オプションなしにこのコマンドを実行して、CLI の使用を開始できます。

```
$ oc login
```

このコマンドの対話式フローでは、指定の認証情報を使用して OpenShift Container Platform サーバーへのセッションを確立することができます。OpenShift Container Platform サーバーに正常にログインするための情報がない場合には、コマンドにより、必要に応じてユーザー入力を求めるプロンプトが出されます。設定は自動的に保存され、その後のコマンドすべてに使用されます。

oc login コマンドのすべての設定オプションは **oc login --help** コマンドの出力で表示されますが、オプションの指定は任意です。以下の例では、一般的なオプションの使用方法を紹介します。

```
$ oc login [-u=<username>] \
  [-p=<password>] \
  [-s=<server>] \
  [-n=<project>] \
  [--certificate-authority=</path/to/file.crt>|--insecure-skip-tls-verify]
```

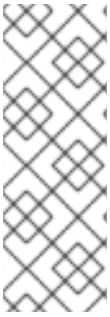
以下の表では、一般的なオプションを紹介しています。

表3.1 一般的な CLI 設定オプション

オプション	構文	説明
-s, --server	`$ oc login -s=<server>`	OpenShift Container Platform サーバーのホスト名を指定します。サーバーがこのフラグで指定されている場合には、このコマンドではホスト名は対話的に確認されません。また、このフラグは、CLI 設定ファイルがある場合や、ログインして別のサーバーに切り替える場合に使用できます。

オプション	構文	説明
-u, --username および **-p, --password**	$ oc login -u=<username> -p=<password>	OpenShift Container Platform サーバーにログインするための認証情報を指定できます。これらのフラグを指定してユーザー名またはパスワードを入力した場合は、このコマンドでは、ユーザー名やパスワードが対話的に確認されません。設定ファイルでセッショントークンを確立し、ログインしてから別のユーザー名に切り替える場合に、これらのフラグを使用することができます。
-n, --namespace	$ oc login -u=<username> -p=<password> -n=<project>	**oc login** と合わせて使用する場合は、グローバル CLI オプションは、指定のユーザーとしてログインしている場合に、切り替え後のプロジェクトを指定することができます。
--certificate-authority	$ oc login --certificate-authority=<path/to/file.crt>	HTTPS を使用する OpenShift Container Platform サーバーで正常かつセキュアに認証します。認証局ファイルへのパスは指定する必要があります。
--insecure-skip-tls-verify	$ oc login --insecure-skip-tls-verify	HTTPS サーバーとの対話を可能にして、サーバーの証明書チェックを省略します。ただし、これはセキュリティーが確保されない点に注意してください。 有効な証明書を提示しない HTTPS サーバーに **oc login** を試行する際に、これかまたは **--certificate-authority** フラグを指定しない場合に、**oc login** は接続がセキュアでないこと確認するユーザー入力 (**y/N** の入力形式) を求めるプロンプトを出します。

CLI構成ファイルを使用すると、複数のCLIプロファイルを簡単に管理できます。



注記

管理者の認証情報がある場合でも デフォルトシステムユーザー の **system:admin** としてログインしていない場合は、認証情報が CLI 設定ファイル にある限り、いつでもこのユーザーとしてログインし直すことができます。以下のコマンドはログインを実行し、**デフォルト** のプロジェクトに切り替えます。

```
$ oc login -u system:admin -n default
```

第4章 承認

4.1. 概要

以下のトピックでは、アプリケーション開発者向けの 認証タスク と、クラスター管理者が指定する認証機能について紹介します。

4.2. ユーザーの POD 作成権限の有無の確認

scc-review と **scc-subject-review** オプションを使用することで、個別ユーザーまたは特定のサービスアカウントのユーザーが Pod を作成または更新可能かどうかを確認できます。

scc-review オプションを使用すると、サービスアカウントが Pod を作成または更新可能かどうかを確認できます。このコマンドは、リソースを許可する SCC (Security Context Constraints) について出力します。

たとえば、**system:serviceaccount:projectname:default** サービスアカウントのユーザーが Pod を作成可能かどうかを確認するには、以下を実行します。

```
$ oc policy scc-review -z system:serviceaccount:projectname:default -f my_resource.yaml
```

scc-subject-review オプションを使用して、特定のユーザーが Pod を作成または更新できるかどうかを確認することも可能です。

```
$ oc policy scc-subject-review -u <username> -f my_resource.yaml
```

特定のグループに所属するユーザーが特定のファイルで Pod を作成できるかどうかを確認するには、以下を実行します。

```
$ oc policy scc-subject-review -u <username> -g <groupname> -f my_resource.yaml
```

4.3. 認証済みのユーザーとして何が実行できるのかを判断する方法

OpenShift Container Platform プロジェクト内で、namespace でスコープ設定されたすべてのリソース (サードパーティーのリソースを含む) に対して実行できる動詞を判別します。

can-i コマンドオプションは、ユーザーとロール関連のスコープをテストします。

```
$ oc policy can-i --list --loglevel=8
```

この出力で、情報収集のために呼び出す API 要求を判断しやすくなります。

ユーザーが判読可能な形式で情報を取得し直すには、以下を実行します。

```
$ oc policy can-i --list
```

この出力では、完全な一覧が表示されます。

特定の verb (動詞) を実行可能かどうかを判断するには、以下を実行します。

```
$ oc policy can-i <verb> <resource>
```

User scopesは、指定のスコープに関する詳細情報を提供します。以下に例を示します。

```
$ oc policy can-i <verb> <resource> --scopes=user:info
```

第5章 プロジェクト

5.1. 概要

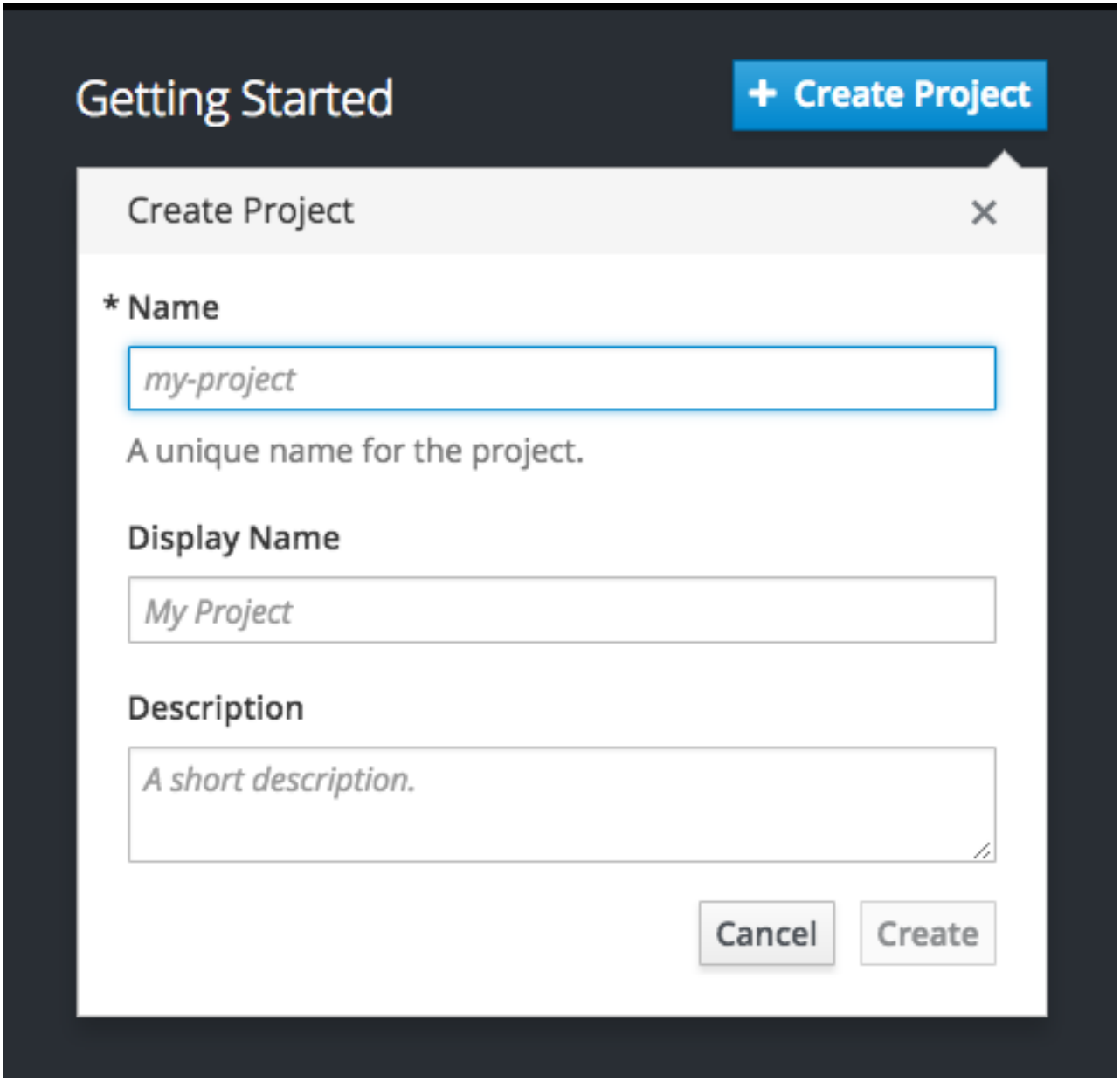
プロジェクト を使用することにより、あるユーザーコミュニティーは、他のコミュニティーと切り離された状態で独自のコンテンツを整理し、管理することができます。

5.2. プロジェクトの作成

クラスター管理者が 許可した場合は、CLI または Web コンソール を使用して新規プロジェクトを作成することができます。

5.2.1. Web コンソールの使用

Web コンソールを使用して新規プロジェクトを作成するには、Project パネルまたは Project ページの **Create Project** ボタンをクリックします。



Create Project ボタンはデフォルトで表示されていますが、オプションで非表示にしたり、カスタマイズしたりすることができます。

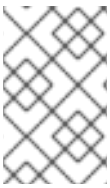
5.2.2. CLI の使用

CLI を使用して新規プロジェクトを作成するには、以下を実行します。

```
$ oc new-project <project_name> \
    --description="<description>" --display-name="<display_name>"
```

以下に例を示します。

```
$ oc new-project hello-openshift \
    --description="This is an example project to demonstrate OpenShift v3" \
    --display-name="Hello OpenShift"
```



注記

作成できるプロジェクトの数は、システム管理者によって制限される場合があります。上限に達すると、既存のプロジェクトを削除してからでないと、新しいプロジェクトは作成できません。

5.3. プロジェクトの表示

プロジェクトを表示する際は、認証ポリシーに基づいて、表示アクセスのあるプロジェクトだけを表示できるように制限されます。

プロジェクトの一覧を表示します。

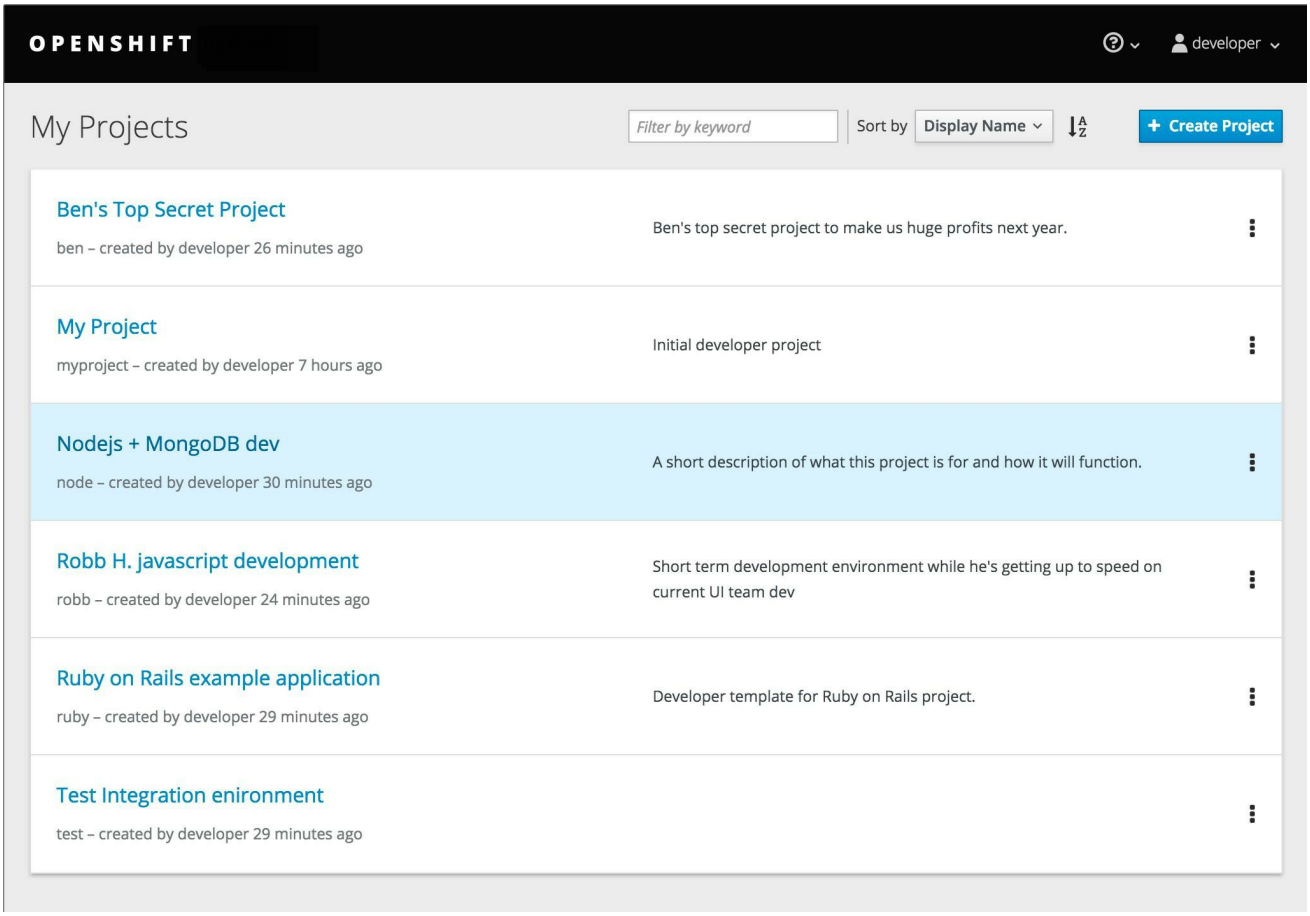
```
$ oc get projects
```

CLI 操作について現在のプロジェクトから別のプロジェクトに切り換えることができます。その後の操作についてはすべて指定のプロジェクトが使用され、プロジェクトスコープのコンテンツの操作が実行されます。

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

また、Web コンソールを使用してプロジェクト間の表示や切り替えが可能です。認証してログインすると、アクセスのあるプロジェクト一覧が表示されます。

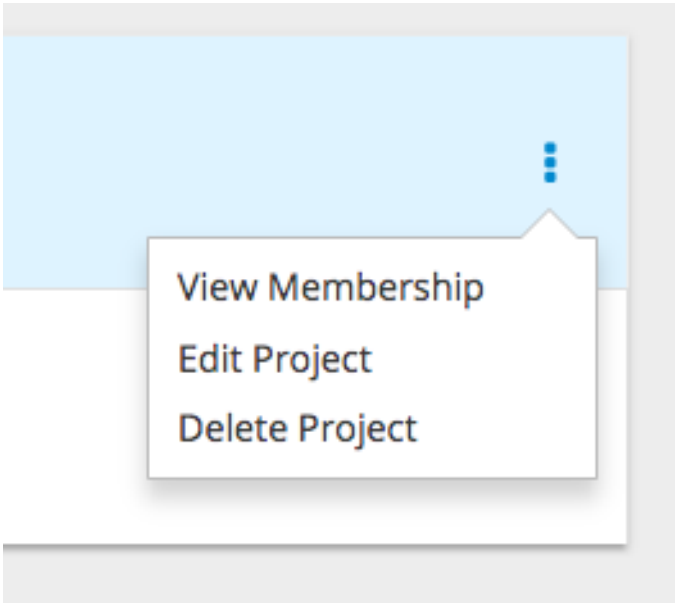
サービスカタログの右側のパネルでは、最近アクセスしたプロジェクト (最大 5 個) へのクィックアクセスが可能です。プロジェクトの詳細一覧については、右パネルの上部にある **View All** リンクを使用します。



CLI を使用して 新規プロジェクト を作成する場合は、ブラウザーでページを更新して、新規プロジェクトを表示することができます。

プロジェクトを選択すると、そのプロジェクトの プロジェクトの概要が表示されます。

特定プロジェクトの kebab (ケバブ) メニューをクリックすると、以下のオプションが表示されます。



5.4. プロジェクトステータスの確認

oc status コマンドは、コンポーネントと関係を含む現在のコンポーネントの概要を示します。このコマンドには引数は指定できません。

-

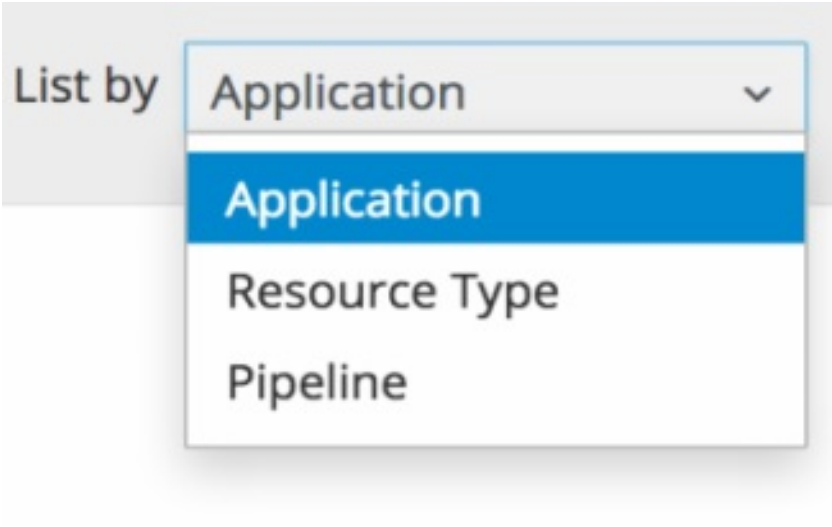
```
$ oc status
```

5.5. ラベル別の絞り込み

リソースの ラベルを使用して Web コンソールのプロジェクトページのコンテンツを絞り込むことができます。提案されたラベル名や値から選択することも、独自の内容を入力することも可能です。また、複数のフィルターを指定することもできます。複数のフィルターが適用される場合には、リソースはすべてのフィルターと一致しないと表示されなくなります。

ラベル別で絞り込むには以下を実行します。

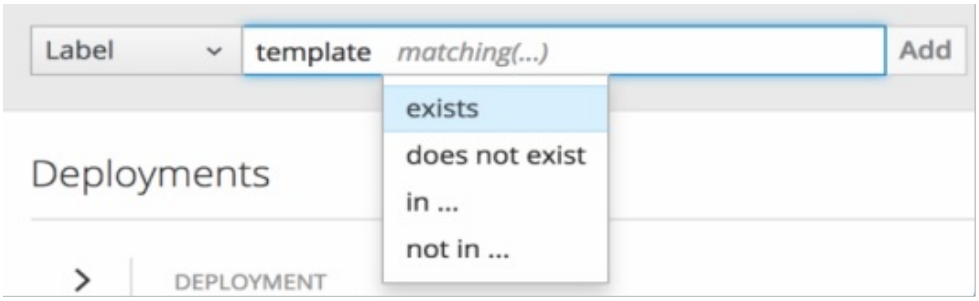
1. ラベルタイプを選択します。



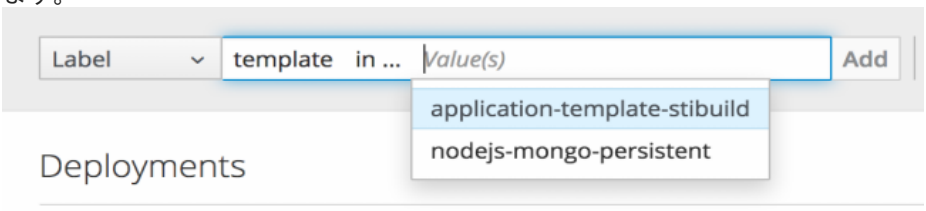
2. 以下のいずれかを選択します。

exists	ラベル名が存在することを確認するだけで、値は無視します。
does not exist	ラベル名が存在しないことを確認してこの値を無視します。
in	ラベル名が存在し、選択した値の 1 つと同じであることを確認します。

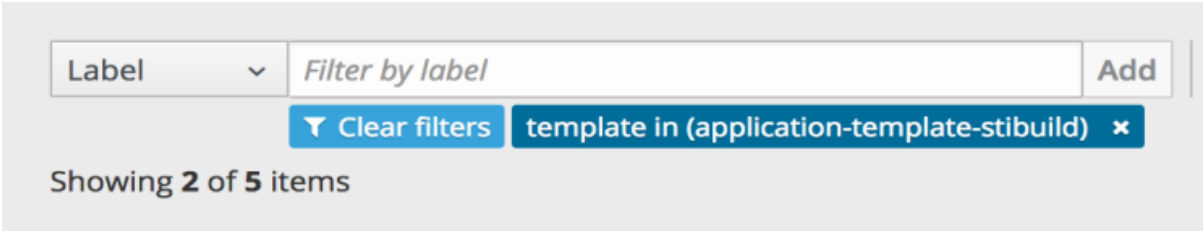
not in	ラベル名が存在しない、または選択した値に該当しないことを確認します。



a. **in** または **not in** を選択した場合には、値セットを選択してから、**Filter** を選択します。



3. フィルターの追加後に、**Clear all filters** を選択するか、削除するフィルターをそれぞれクリックして、絞り込みを停止します。



5.6. ページの状態のブックマーク

OpenShift Container Platform Web コンソール では、ページの状態をブックマークできるようになり、ラベルのフィルターや他の設定を保存する際に役立ちます。

タブ間の切り替えなど、ページの状態を変更する作業を行った場合には、ブラウザーのナビゲーションバーの URL が自動的に更新されます。

5.7. プロジェクトの削除

プロジェクトを削除する際に、サーバーはプロジェクトのステータスを Active から Terminating に更新します。次にサーバーは、Terminating の状態のプロジェクトからコンテンツをすべて削除してから、プロジェクトを最終的に削除します。プロジェクトが Terminating のステータスの間は、ユーザーはそのプロジェクトに新規コンテンツを追加できません。プロジェクトは CLI または Web コンソールから削除できます。

CLI を使用してプロジェクトを削除するには以下を実行します。

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

第6章 アプリケーションの移行

6.1. 概要

以下のトピックでは、OpenShift version 2 (v2) アプリケーションから OpenShift version 3 (v3) に移行する手順を説明します。

注記

以下のトピックでは、OpenShift v2 に固有の用語を使用します。「Comparing OpenShift Enterprise 2 and OpenShift Enterprise 3」では、これら 2 つのバージョンや、使用する用語の違いについて詳しく説明しています。

OpenShift v2 アプリケーションから OpenShift Container Platform v3 に移行するには、各 v2 カートリッジは OpenShift Container Platform v3 の対応のイメージまたはテンプレートと同等であり、個別に移行する必要があるので、v2 アプリケーションのすべてのカートリッジを記録する必要があります。またそれぞれのカートリッジについて、すべての依存関係または必要なパッケージは v3 イメージに含める必要があるため、それらを記録する必要もあります。

一般的な移行手順は以下のとおりです。

1. v2 アプリケーションをバックアップします。
   - Web カートリッジ: ソースコードは、GitHub のリポジトリーにプッシュするなど、Git リポジトリーにバックアップすることができます。
   - データベースカートリッジ: データベースは、dump コマンドを使用してバックアップすることができます (**mongodump**、**mysqldump**、**pg_dump**)。
   - Web およびデータベースカートリッジ: **rhc** クライアントツールには、複数のカートリッジをバックアップするスナップショットの機能があります。

     ```
     $ rhc snapshot save <app_name>
     ```

     スナップショットは展開可能な tar ファイルであり、このファイルには、アプリケーションのソースコードとデータベースのダンプが含まれます。
2. アプリケーションにデータベースカートリッジが含まれる場合には、v3 データベースアプリケーションを作成し、データベースダンプを新しい v3 データベースアプリケーションの Pod に同期してから、データベースの復元コマンドを使用して v3 データベースアプリケーションに v2 データベースを復元します。
3. Web フレームワークアプリケーションの場合には、v3 と互換性を持たせるようにアプリケーションのソースコードを編集します。次に、Git リポジトリーの適切なファイルに必要な依存関係またはパッケージを追加します。v2 環境変数を対応する v3 環境変数に変換します。
4. ソース (Git リポジトリー) または Git URL のクイックスタートから v3 アプリケーションを作成します。また、データベースのサービスパラメーターを新規アプリケーションに追加して、データベースアプリケーションと Web アプリケーションをリンクします。
5. v2 には統合 git 環境があり、アプリケーションは v2 git リポジトリーに変更がプッシュされるたびに自動的に再ビルドされ、再起動されます。v3 では、ビルドがパブリックの git リポジトリーにプッシュされるソースコードの変更で自動的にトリガーされるようにするために、v3 の初期ビルドの完了後に webhook を設定する必要があります。

6.2. データベースアプリケーションの移行

6.2.1. 概要

以下のトピックでは、MySQL、PostgreSQL および MongoDB データベースアプリケーションを OpenShift バージョン 2 (v2) から OpenShift version 3 (v3) に移行する方法を確認します。

6.2.2. サポートされているデータベース

v2	v3
MongoDB: 2.4	MongoDB: 2.4, 2.6
MySQL: 5.5	MySQL: 5.5, 5.6
PostgreSQL: 9.2	PostgreSQL: 9.2, 9.4

6.2.3. MySQL

1. すべてのデータベースをダンプファイルにエクスポートして、これをローカルマシン (現在のディレクトリー) にコピーします

```
$ rhc ssh <v2_application_name>
$ mysqldump --skip-lock-tables -h $OPENSHIFT_MYSQL_DB_HOST -P
${OPENSHIFT_MYSQL_DB_PORT:-3306} -u ${OPENSHIFT_MYSQL_DB_USERNAME:-
'admin'} \
 --password="$OPENSHIFT_MYSQL_DB_PASSWORD" --all-databases > ~/app-
root/data/all.sql
$ exit
```

2. **dbdump** をローカルマシンにダウンロードします。

```
$ mkdir mysqldumpdir
$ rhc scp -a <v2_application_name> download mysqldumpdir app-root/data/all.sql
```

3. テンプレートから v3 **mysql-persistent** Pod を作成します。

```
$ oc new-app mysql-persistent -p \
   MYSQL_USER=<your_V2_mysql_username> -p \
   MYSQL_PASSWORD=<your_v2_mysql_password> -p MYSQL_DATABASE=
<your_v2_database_name>
```

4. Pod の使用準備ができているかどうかを確認します。

```
$ oc get pods
```

5. Pod の実行中に、データベースのアーカイブファイルを v3 MySQL Pod にコピーします。

```
$ oc rsync /local/mysqldumpdir <mysql_pod_name>:/var/lib/mysql/data
```

6. v3 の実行中の Pod に、データベースを復元します。

```
$ oc rsh <mysql_pod>
$ cd /var/lib/mysql/data/mysqldumpdir
```

v3 では、データベースを復元するには、**root** ユーザーとして MySQL にアクセスする必要があります。

v2 では、**$OPENSHIFT_MYSQL_DB_USERNAME** には全データベースに対する完全な権限がありました。v3 では、権限をデータベースごとに **$MYSQL_USER** に割り当てる必要があります。

```
$ mysql -u root
$ source all.sql
```

<dbname> のすべての権限を **<your_v2_username>@localhost** に割り当ててから、権限をフラッシュします。

7. Pod からダンプディレクトリーを削除します。

```
$ cd ../; rm -rf /var/lib/mysql/data/mysqldumpdir
```

サポート対象の MySQL 環境変数

v2	v3
OPENSHIFT_MYSQL_DB_HOST	**[service_name]_SERVICE_HOST**
OPENSHIFT_MYSQL_DB_PORT	**[service_name]_SERVICE_PORT**
OPENSHIFT_MYSQL_DB_USERNAME	**MYSQL_USER**
OPENSHIFT_MYSQL_DB_PASSWORD	**MYSQL_PASSWORD**
OPENSHIFT_MYSQL_DB_URL	
OPENSHIFT_MYSQL_DB_LOG_DIR	
OPENSHIFT_MYSQL_VERSION	
OPENSHIFT_MYSQL_DIR	
OPENSHIFT_MYSQL_DB_SOCKET	
OPENSHIFT_MYSQL_IDENT	
OPENSHIFT_MYSQL_AIO	**MYSQL_AIO**
OPENSHIFT_MYSQL_MAX_ALLOWED_PACKET	**MYSQL_MAX_ALLOWED_PACKET**

v2	v3
OPENSHIFT_MYSQL_TABLE_OPEN_CACHE	**MYSQL_TABLE_OPEN_CACHE**
OPENSHIFT_MYSQL_SORT_BUFFER_SIZE	**MYSQL_SORT_BUFFER_SIZE**
OPENSHIFT_MYSQL_LOWER_CASE_TABLE_NAMES	**MYSQL_LOWER_CASE_TABLE_NAMES**
OPENSHIFT_MYSQL_MAX_CONNECTIONS	**MYSQL_MAX_CONNECTIONS**
OPENSHIFT_MYSQL_FT_MIN_WORD_LEN	**MYSQL_FT_MIN_WORD_LEN**
OPENSHIFT_MYSQL_FT_MAX_WORD_LEN	**MYSQL_FT_MAX_WORD_LEN**
OPENSHIFT_MYSQL_DEFAULT_STORAGE_ENGINE	
OPENSHIFT_MYSQL_TIMEZONE	
	MYSQL_DATABASE
	MYSQL_ROOT_PASSWORD
	MYSQL_MASTER_USER
	MYSQL_MASTER_PASSWORD

6.2.4. PostgreSQL

1. ギアから v2 PostgreSQL データベースをバックアップします。

```
$ rhc ssh -a <v2-application_name>
$ mkdir ~/app-root/data/tmp
$ pg_dump <database_name> | gzip > ~/app-root/data/tmp/<database_name>.gz
```

2. ローカルマシンに、バックアップファイルを展開します。

```
$ rhc scp -a <v2_application_name> download <local_dest> app-root/data/tmp/<db-name>.gz
$ gzip -d <database-name>.gz
```

注記

手順 4 とは別のフォルダーにバックアップファイルを保存します。

3. 新規サービスを作成するための v2 アプリケーションのデータベース名、ユーザー名、パスワードを使用して PostgreSQL サービスを作成します。

```
$ oc new-app postgresql-persistent -p POSTGRESQL_DATABASE=dbname -p
POSTGRESQL_PASSWORD=password -p POSTGRESQL_USER=username
```

4. Pod の使用準備ができているかどうかを確認します。

```
$ oc get pods
```

5. Pod を実行中に、バックアップディレクトリーを Pod に同期します。

```
$ oc rsync /local/path/to/dir <postgresql_pod_name>:/var/lib/pgsql/data
```

6. Pod にリモートからアクセスします。

```
$ oc rsh <pod_name>
```

7. データベースを復元します。

```
psql dbname < /var/lib/pgsql/data/<database_backup_file>
```

8. 必要のなくなったバックアップファイルをすべて削除します。

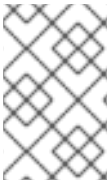
```
$ rm /var/lib/pgsql/data/<database-backup-file>
```

サポート対象の PostgreSQL 環境変数

v2	v3
OPENSHIFT_POSTGRESQL_DB_HOST	**[service_name]_SERVICE_HOST**
OPENSHIFT_POSTGRESQL_DB_PORT	**[service_name]_SERVICE_PORT**
OPENSHIFT_POSTGRESQL_DB_USERNAME	**POSTGRESQL_USER**
OPENSHIFT_POSTGRESQL_DB_PASSWORD	**POSTGRESQL_PASSWORD**
OPENSHIFT_POSTGRESQL_DB_LOG_DIR	
OPENSHIFT_POSTGRESQL_DB_PID	
OPENSHIFT_POSTGRESQL_DB_SOCKET_DIR	
OPENSHIFT_POSTGRESQL_DB_URL	
OPENSHIFT_POSTGRESQL_VERSION	
OPENSHIFT_POSTGRESQL_SHARED_BUFFERS	

v2	v3
OPENSHIFT_POSTGRESQL_MAX_CONNECTIONS	
OPENSHIFT_POSTGRESQL_MAX_PREPARED_TRANSACTIONS	
OPENSHIFT_POSTGRESQL_DATESTYLE	
OPENSHIFT_POSTGRESQL_LOCALE	
OPENSHIFT_POSTGRESQL_CONFIG	
OPENSHIFT_POSTGRESQL_SSL_ENABLED	
	POSTGRESQL_DATABASE
	POSTGRESQL_ADMIN_PASSWORD

6.2.5. MongoDB



注記

- OpenShift v3 の場合: MongoDB シェルバージョン 3.2.6
- OpenShift v2 の場合: MongoDB シェルバージョン 2.4.9

1. **ssh** コマンドを使用して、v2 アプリケーションにリモートからアクセスします。

```
$ rhc ssh <v2_application_name>
```

2. **-d <database_name> -c <collections>** で単一のデータベースを指定して、**mongodump** を実行します。このオプションがないと、データベースはすべてダンプされます。各データベースは、独自のディレクトリーにダンプされます。

```
$ mongodump -h $OPENSHIFT_MONGODB_DB_HOST -o app-root/repo/mydbdump -u
'admin' -p $OPENSHIFT_MONGODB_DB_PASSWORD
$ cd app-root/repo/mydbdump/<database_name>; tar -cvzf dbname.tar.gz
$ exit
```

3. **dbdump** を **mongodump** ディレクトリーのローカルマシンにダウンロードします。

```
$ mkdir mongodump
$ rhc scp -a <v2 appname> download mongodump \
  app-root/repo/mydbdump/<dbname>/dbname.tar.gz
```

4. v3 で MongoDB Pod を実行します。最新のイメージ (3.2.6) には **mongo-tools** が含まれないので、**mongorestore** または **mongoimport** コマンドを使用するには、デフォルトの **mongodb-**

persistent テンプレートを編集して、**mongo-tools, "mongodb:2.4"** を含むイメージタグを指定します。このため、以下の **oc export** コマンドを使用して、編集することが必要です。

```
$ oc export template mongodb-persistent -n openshift -o json > mongodb-24persistent.json
```

mongodb-24persistent.json の L80 を編集します。 **mongodb:latest** は **mongodb:2.4** に置き換えてください。

```
$ oc new-app --template=mongodb-persistent -n <project-name-that-template-was-created-
in> \
  MONGODB_USER=user_from_v2_app -p \
  MONGODB_PASSWORD=password_from_v2_db -p \
  MONGODB_DATABASE=v2_dbname -p \
  MONGODB_ADMIN_PASSWORD=password_from_v2_db
$ oc get pods
```

5. mongodb Pod の実行中に、データベースのアーカイブファイルを v3 MongoDB Pod にコピーします。

```
$ oc rsync local/path/to/mongodump <mongodb_pod_name>:/var/lib/mongodb/data
$ oc rsh <mongodb_pod>
```

6. MongoDB Pod で、復元する各データベースについて以下を実行します。

```
$ cd /var/lib/mongodb/data/mongodump
$ tar -xzvf dbname.tar.gz
$ mongorestore -u $MONGODB_USER -p $MONGODB_PASSWORD -d dbname -v
/var/lib/mongodb/data/mongodump
```

7. データベースが復元されたかどうかを確認します。

```
$ mongo admin -u $MONGODB_USER -p $MONGODB_ADMIN_PASSWORD
$ use dbname
$ show collections
$ exit
```

8. Pod から **mongodump** ディレクトリーを削除します。

```
$ rm -rf /var/lib/mongodb/data/mongodump
```

サポート対象の MongoDB 環境変数

v2	v3
OPENSHIFT_MONGODB_DB_HOST	**[service_name]_SERVICE_HOST**
OPENSHIFT_MONGODB_DB_PORT	**[service_name]_SERVICE_PORT**
OPENSHIFT_MONGODB_DB_USERNAME	**MONGODB_USER**
OPENSHIFT_MONGODB_DB_PASSWORD	**MONGODB_PASSWORD**

v2	v3
OPENSHIFT_MONGODB_DB_URL	
OPENSHIFT_MONGODB_DB_LOG_DIR	
	MONGODB_DATABASE
	MONGODB_ADMIN_PASSWORD
	MONGODB_NOPREALLOC
	MONGODB_SMALLFILES
	MONGODB_QUIET
	MONGODB_REPLICA_NAME
	MONGODB_KEYFILE_VALUE

6.3. WEB フレームワークアプリケーションの移行

6.3.1. 概要

以下のトピックでは、Python、Ruby、PHP、Perl、Node.js、WordPress、Ghost、JBoss EAP、JBoss WS (Tomcat) および Wildfly 10 (JBoss AS) の Web フレームワークアプリケーションを OpenShift version 2 (v2) から OpenShift version 3 (v3) に移行する方法を確認します。

6.3.2. Python

1. 新しい GitHub リポジトリーを設定して、そのリポジトリーをリモートのブランチとして現在のローカル v2 Git リポジトリーに追加します。

```
$ git remote add <remote-name> https://github.com/<github-id>/<repo-name>.git
```

2. ローカルの v2 ソースコードを新規リポジトリーにプッシュします。

```
$ git push -u <remote-name> master
```

3. **setup.py**、**wsgi.py**、**requirements.txt** および **etc** などの重要なファイルがすべて新規リポジトリーにプッシュされていることを確認します。
   - アプリケーションに必要なパッケージがすべて **requirements.txt** に含まれていることを確認します。
4. **oc** コマンドを使用して、ビルダーイメージとソースコードから新規の Python アプリケーションを起動します。

```
$ oc new-app --strategy=source
python:3.3~https://github.com/<github-id>/<repo-name> --name=<app-name> -e
<ENV_VAR_NAME>=<env_var_value>
```

サポート対象の Python バージョン

v2	v3
Python: 2.6, 2.7, 3.3	サポート対象のコンテナーイメージ
Django	Django-psql-example (quickstart)

6.3.3. Ruby

1. 新しい GitHub リポジトリーを設定して、そのリポジトリーをリモートのブランチとして現在のローカル v2 Git リポジトリーに追加します。

```
$ git remote add <remote-name> https://github.com/<github-id>/<repo-name>.git
```

2. ローカルの v2 ソースコードを新規リポジトリーにプッシュします。

```
$ git push -u <remote-name> master
```

3. Gemfile がなく、単純な rack アプリケーションを実行している場合には、この Gemfile ファイルをソースの root にコピーします。

```
https://github.com/sclorg/ruby-ex/blob/master/Gemfile
```

注記

Ruby 2.0 がサポートする **rack** gem の最新バージョンは 1.6.4 であるため、Gemfile は **gem 'rack', "1.6.4"** に変更する必要があります。

Ruby 2.2 以降の場合は、**rack** gem 2.0 以降を使用してください。

4. **oc** コマンドを使用して、ビルダーイメージとソースコードから新規の Ruby アプリケーションを起動します。

```
$ oc new-app --strategy=source
ruby:2.0~https://github.com/<github-id>/<repo-name>.git
```

サポート対象の Ruby バージョン

v2	v3
Ruby: 1.8, 1.9, 2.0	サポート対象のコンテナーイメージ
Ruby on Rails: 3, 4	Rails-postgresql-example (quickstart)

v2	v3
Sinatra	

6.3.4. PHP

1. 新しい GitHub リポジトリーを設定して、そのリポジトリーをリモートのブランチとして現在のローカル v2 Git リポジトリーに追加します。

```
$ git remote add <remote-name> https://github.com/<github-id>/<repo-name>
```

2. ローカルの v2 ソースコードを新規リポジトリーにプッシュします。

```
$ git push -u <remote-name> master
```

3. **oc** コマンドを使用して、ビルダーイメージとソースコードから新規の PHP アプリケーションを起動します。

```
$ oc new-app https://github.com/<github-id>/<repo-name>.git
--name=<app-name> -e <ENV_VAR_NAME>=<env_var_value>
```

サポート対象の PHP バージョン

v2	v3
PHP: 5.3, 5.4	サポート対象のコンテナーイメージ
PHP 5.4 with Zend Server 6.1	
CodeIgniter 2	
HHVM	
Laravel 5.0	
	cakephp-mysql-example (quickstart)

6.3.5. Perl

1. 新しい GitHub リポジトリーを設定して、そのリポジトリーをリモートのブランチとして現在のローカル v2 Git リポジトリーに追加します。

```
$ git remote add <remote-name> https://github.com/<github-id>/<repo-name>
```

2. ローカルの v2 ソースコードを新規リポジトリーにプッシュします。

```
$ git push -u <remote-name> master
```

3. ローカルの Git リポジトリーを編集して、変更をアップストリームにプッシュして、v3 との互換性を確保します。

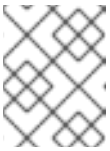
   a. v2 では、CPAN モジュールは **.openshift/cpan.txt** にあります。v3 では、s2i ビルダーは、ソースのルートディレクトリーで **cpanfile** という名前のファイルを検索します。

   ```
   $ cd <local-git-repository>
   $ mv .openshift/cpan.txt cpanfile
   ```

   cpanfile の形式が若干異なるので、これを編集します。

cpanfile の形式	cpan.txt の形式
'cpan::mod' が必要	cpan::mod
requires 'Dancer';	Dancer
requires 'YAML';	YAML

   b. **.openshift** ディレクトリーを削除します。



注記

v3 では、**action_hooks** および **cron** タスクは同じようにサポートされません。詳細情報は、「アクションフック」を参照してください。

4. **oc** コマンドを使用して、ビルダーイメージとソースコードから新規の Perl アプリケーションを起動します。

```
$ oc new-app https://github.com/<github-id>/<repo-name>.git
```

サポート対象の Perl バージョン

v2	v3
Perl: 5.10	サポート対象のコンテナーイメージ
	Dancer-mysql-example (quickstart)

6.3.6. Node.js

1. 新しい GitHub リポジトリーを設定して、そのリポジトリーをリモートのブランチとして現在のローカル Git リポジトリーに追加します。

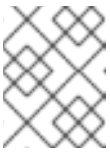
```
$ git remote add <remote-name> https://github.com/<github-id>/<repo-name>
```

2. ローカルの v2 ソースコードを新規リポジトリーにプッシュします。

```
$ git push -u <remote-name> master
```

3. ローカルの Git リポジトリーを編集して、変更をアップストリームにプッシュして、v3 との互換性を確保します。

   a. **.openshift** ディレクトリーを削除します。

   注記

   v3 では、**action_hooks** および **cron** タスクは同じようにサポートされません。詳細情報は、「アクションフック」を参照してください。

   b. **server.js** を編集します。

   - L116 server.js: 'self.app = express();'
   - L25 server.js: self.ipaddress = '0.0.0.0';
   - L26 server.js: self.port = 8080;

   注記

   Lines(L) は V2 カートリッジの **server.js** から取得されます。

4. **oc** コマンドを使用して、ビルダーイメージとソースコードから新規の Node.js アプリケーションを起動します。

```
$ oc new-app https://github.com/<github-id>/<repo-name>.git
--name=<app-name> -e <ENV_VAR_NAME>=<env_var_value>
```

サポート対象の Node.js バージョン

v2	v3
Node.js 0.10	サポート対象のコンテナーイメージ
	Nodejs-mongodb-example。このクイックスタートテンプレートは Node.js バージョン 6 のみをサポートします。

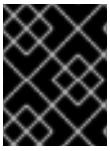
6.3.7. WordPress

重要

現時点で WordPress アプリケーションの移行はコミュニティーによるサポートのみで、Red hat のサポートはありません。

WordPress アプリケーションの OpenShift Container Platform v3 への移行に関する情報は、「OpenShift ブログ」を参照してください。

6.3.8. Ghost



重要

現時点で Ghost アプリケーションの移行はコミュニティーによるサポートのみで、Red hat のサポートはありません。

Ghost アプリケーションの OpenShift Container Platform v3 への移行に関する情報は、「OpenShift ブログ」を参照してください。

6.3.9. JBoss EAP

1. 新しい GitHub リポジトリーを設定して、そのリポジトリーをリモートのブランチとして現在のローカル Git リポジトリーに追加します。

```
$ git remote add <remote-name> https://github.com/<github-id>/<repo-name>
```

2. ローカルの v2 ソースコードを新規リポジトリーにプッシュします。

```
$ git push -u <remote-name> master
```

3. リポジトリーに事前にビルドされた **.war** ファイルが含まれている場合には、それらをリポジトリーの root ディレクトリー内の **deployments** ディレクトリーに置く必要があります。

4. JBoss EAP 7 ビルダーイメージ (jboss-eap70-openshift) と GitHub からのソースコードリポジトリーを使用して新規アプリケーションを作成します。

```
$ oc new-app --strategy=source jboss-eap70-openshift:1.6~https://github.com/<github-id>/<repo-name>.git
```

6.3.10. JBoss WS (Tomcat)

1. 新しい GitHub リポジトリーを設定して、そのリポジトリーをリモートのブランチとして現在のローカル Git リポジトリーに追加します。

```
$ git remote add <remote-name> https://github.com/<github-id>/<repo-name>
```

2. ローカルの v2 ソースコードを新規リポジトリーにプッシュします。

```
$ git push -u <remote-name> master
```

3. リポジトリーに事前にビルドされた **.war** ファイルが含まれている場合には、それらをリポジトリーの root ディレクトリー内の **deployments** ディレクトリーに置く必要があります。

4. JBoss Web Server 3 (Tomcat 7) ビルダーイメージ (jboss-webserver30-tomcat7) と GitHub からのソースコードリポジトリーを使用して新規アプリケーションを作成します。

```
$ oc new-app --strategy=source
jboss-webserver30-tomcat7-openshift~https://github.com/<github-id>/<repo-name>.git
--name=<app-name> -e <ENV_VAR_NAME>=<env_var_value>
```

6.3.11. JBoss AS (Wildfly 10)

1. 新しい GitHub リポジトリーを設定して、そのリポジトリーをリモートのブランチとして現在のローカル Git リポジトリーに追加します。

```
$ git remote add <remote-name> https://github.com/<github-id>/<repo-name>
```

2. ローカルの v2 ソースコードを新規リポジトリーにプッシュします。

```
$ git push -u <remote-name> master
```

3. ローカルの Git リポジトリーを編集して、変更をアップストリームにプッシュして v3 との互換性を確保します。

   a. **.openshift** ディレクトリーを削除します。

注記

v3 では、**action_hooks** および **cron** タスクは同じようにサポートされません。詳細情報は、「アクションフック」を参照してください。

   b. **deployments** ディレクトリーをソースリポジトリーの root に追加します。**.war** ファイルをこの「deployments」ディレクトリーに移動します。

4. **oc** コマンドを使用して、ビルダーイメージとソースコードから新規の Wildfly アプリケーションを起動します。

```
$ oc new-app https://github.com/<github-id>/<repo-name>.git
 --image-stream=”openshift/wildfly:10.0" --name=<app-name> -e
 <ENV_VAR_NAME>=<env_var_value>
```

注記

引数 **--name** はアプリケーション名を指定するためのオプションの引数です。また、**-e** は **OPENSHIFT_PYTHON_DIR** などのビルドやデプロイメントプロセスに必要な環境変数を追加するためのオプションの引数です。

6.3.12. サポート対象の JBoss バージョン

v2	v3
JBoss App Server 7	
Tomcat 6 (JBoss EWS 1.0)	サポート対象のコンテナーイメージ
Tomcat 7 (JBoss EWS 2.0)	サポート対象のコンテナーイメージ
Vert.x 2.1	
WildFly App Server 10	
WildFly App Server 8.2.1.Final	

v2	v3
WildFly App Server 9	
CapeDwarf	
JBoss Data Virtualization 6	サポート対象のコンテナーイメージ
JBoss Enterprise App Platform (EAP) 6	サポート対象のコンテナーイメージ
JBoss Unified Push Server 1.0.0.Beta1, Beta2	
JBoss BPM Suite	サポート対象のコンテナーイメージ
JBoss BRMS	サポート対象のコンテナーイメージ
	jboss-eap70-openshift: 1.3-Beta
	eap64-https-s2i
	eap64-mongodb-persistent-s2i
	eap64-mysql-persistent-s2i
	eap64-psql-persistent-s2i

6.4. クイックスタートの例

6.4.1. 概要

v2 クイックスタートから v3 クイックスタートへの明確な移行パスはありませんが、v3 では以下のクイックスタートを利用できます。データベースを含むアプリケーションがある場合には、**oc new-app** でアプリケーションを作成してから、もう一度 **oc new-app** を実行して別のデータベースサービスを起動し、これら 2 つを共通の環境変数を使用してリンクするのではなく、以下のいずれかを使用し、ソースコードを含む GitHub リポジトリーからリンクしたアプリケーションとデータベースを一度にインスタンス化できます。**oc get templates -n openshift** で利用可能なテンプレートをすべて表示することができます。

- CakePHP MySQL https://github.com/sclorg/cakephp-ex
  - テンプレート: cakephp-mysql-example
- Node.js MongoDB https://github.com/sclorg/nodejs-ex
  - テンプレート: nodejs-mongodb-example
- Django PosgreSQL https://github.com/sclorg/django-ex
  - テンプレート: django-psql-example

- Dancer MySQL https://github.com/sclorg/dancer-ex
  - テンプレート: dancer-mysql-example
- Rails PostgreSQL https://github.com/sclorg/rails-ex
  - テンプレート: rails-postgresql-example

6.4.2. ワークフロー

上記のテンプレート URL のいずれかに対して **git clone** をローカルで実行します。アプリケーションのソースコードを追加し、コミットし、GitHub リポジトリーにプッシュしてから、上記のテンプレートのいずれかで v3 クイックスタートアプリケーションを起動します。

1. アプリケーション用の GitHub リポジトリーを作成します。
2. クイックスタートテンプレートのクローンを作成して、GitHub リポジトリーをリモートとして追加します。

```
$ git clone <one-of-the-template-URLs-listed-above>
$ cd <your local git repository>
$ git remote add upstream <https://github.com/<git-id>/<quickstart-repo>.git>
$ git push -u upstream master
```

3. ソースコードを GitHub にコミットし、プッシュします。

```
$ cd <your local repository>
$ git commit -am “added code for my app”
$ git push origin master
```

4. v3 で新規アプリケーションを作成します。

```
$ oc new-app --template=<template> \
-p SOURCE_REPOSITORY_URL=<https://github.com/<git-id>/<quickstart_repo>.git> \
-p DATABASE_USER=<your_db_user> \
-p DATABASE_NAME=<your_db_name> \
-p DATABASE_PASSWORD=<your_db_password> \
-p DATABASE_ADMIN_PASSWORD=<your_db_admin_password> (1)
```

(1) MongoDB にのみ該当します。

web フレームワーク Pod とデータベース Pod の 2 つの Pod が実行されます。Web フレームワーク Pod 環境は、データベース Pod 環境と一致しているはずです。環境変数は、**oc set env pod/<pod_name> --list** で一覧表示できます。

- **DATABASE_NAME** は **<DB_SERVICE>_DATABASE** になります。
- **DATABASE_USER** は **<DB_SERVICE>_USER** になります。
- **DATABASE_PASSWORD**は **<DB_SERVICE>_PASSWORD** になります。
- **DATABASE_ADMIN_PASSWORD** は **MONGODB_ADMIN_PASSWORD** になります (MongoDB のみに該当します)。

SOURCE_REPOSITORY_URL が指定されていない場合、テンプレートはソースリポジトリーとして上記のテンプレート URL (**https://github.com/openshift/<quickstart>-ex**) を使用して、**hello-welcome** アプリケーションが起動します。

5. データベースを移行する場合は、データベースをダンプファイルにエクスポートして、新しい v3 データベース Pod にデータベースを復元します。「データベースアプリケーション」に記載の手順を参照してください。ただし、データベース Pod はすでに実行中であるため、**oc new-app** の手順は省略してください。

6.5. 継続的インテグレーションまたは継続的デプロイ (CI/CD)

6.5.1. 概要

以下のトピックでは、OpenShift バージョン 2 (v2) と OpenShift バージョン 3 (v3) 間の継続的インテグレーションおよびデプロイメント (CI/CD) アプリケーションの相違点と、これらのアプリケーションを v3 環境に移行する方法を確認します。

6.5.2. Jenkins

Jenkins アプリケーションは、アーキテクチャーの根本的な違いにより OpenShift バージョン 2 (v2) と OpenShift バージョン 3 (v3) では異なる方法で設定されます。たとえば、v2 ではアプリケーションはギアでホストされる統合型の Git リポジトリーを使用してソースコードを保存します。v3 では、ソースコードは Pod の外部でホストされるパブリックまたはプライベート Git リポジトリーに置かれます。

さらに OpenShift v3 では、Jenkins ジョブは、ソースコードの変更だけでなく、ソースコードと共にアプリケーションをビルドするために使用されるイメージの変更である ImageStream の変更によってもトリガーされます。そのため、v3 で新しい Jenkins アプリケーションを作成してから、OpenShift v3 環境に適した設定でジョブを作成し直して Jenkins アプリケーションを手動で移行することを推奨します。

Jenkins アプリケーションの作成、ジョブの設定、Jenkins プラグインの正しい使用の方法に関する詳細は、以下のリソースを参照してください。

- https://github.com/openshift/origin/blob/master/examples/jenkins/README.md
- https://github.com/openshift/jenkins-plugin/blob/master/README.md
- https://github.com/openshift/origin/blob/master/examples/sample-app/README.md

6.6. WEBHOOK およびアクションフック

6.6.1. 概要

以下のトピックでは、OpenShift バージョン 2 (v2) と OpenShift バージョン 3 (v3) 間の webhook とアクションフックの相違点と、これらのアプリケーションの v3 環境への移行方法について説明します。

6.6.2. Webhook

1. 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 を出力します。

-

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

2. 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。
3. GitHub リポジトリーで、**Settings → Webhooks & Services** から **Add Webhook** を選択します。
4. **Payload URL** フィールドに、(上記と同様の) URL の出力を貼り付けます。
5. **Content Type** を **application/json** に設定します。
6. **Add webhook** をクリックします。

webhook の設定が正常に完了したことを示す GitHub のメッセージが表示されます。

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

注記

アプリケーションを削除または再作成する場合には、GitHub の **Payload URL** フィールドを **BuildConfig** webhook url で更新する必要があります。

6.6.3. アクションフック

OpenShift バージョン 2 (v2) では、**.openshift/action_hooks** ディレクトリーに build、deploy、post_deploy および pre_build スクリプトまたは action_hooks が置かれます。v3 にはこれらのスクリプトに対応する 1 対 1 の機能マッピングはありませんが、v3 の S2I ツール には カスタム可能なスクリプト を指定の URL またはソースリポジトリーの **.s2i/bin** ディレクトリーに追加するオプションがあります。

OpenShift バージョン 3 (v3) には、イメージをビルドしてからレジストリーにプッシュするまでのイメージの基本的なテストを実行する post-build hook があります。デプロイメントフック はデプロイメント構成で設定されます。

v2 では、通常 action_hooks は環境変数を設定するために使用されます。v2 では、環境変数は以下のように渡される必要があります。

```
$ oc new-app <source-url> -e ENV_VAR=env_var
```

または

```
$ oc new-app <template-name> -p ENV_VAR=env_var
```

または、以下を使用して環境変数を追加し、変更することができます。

```
$ oc set env dc/<name-of-dc>
ENV_VAR1=env_var1 ENV_VAR2=env_var2'
```

6.7. S2I ツール

6.7.1. 概要

Source-to-Image (S2I) ツールは、アプリケーションのソースコードをコンテナーイメージに挿入します。最終成果物として、ビルダーイメージとビルド済みのソースコードが組み込まれた実行準備のできたコンテナーイメージが新たに作成されます。S2I ツールは、OpenShift Container Platform がなくても、リポジトリー から、ローカルマシンにインストールできます。

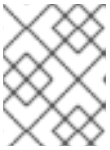
S2I ツールは、OpenShift Container Platform で使用する前にアプリケーションとイメージをローカルでテストし、検証するための非常に強力なツールです。

6.7.2. コンテナーイメージの作成

1. アプリケーションに必要なビルダーイメージを特定します。Red Hat は、Python、Ruby、Perl、PHP および Node.js など各種の言語のビルダーイメージを複数提供しています。他のイメージは コミュニティースペース から取得できます。

2. S2I は、Git リポジトリーまたはローカルのファイルシステムのソースコードからイメージをビルドできます。ビルダーイメージおよびソースコードから新しいコンテナーイメージをビルドするには、以下を実行します。

```
$ s2i build <source-location> <builder-image-name> <output-image-name>
```



注記

<source-location> には Git リポジトリーの URL、 またはローカルファイルシステムのソースコードのディレクトリーのいずれかを指定できます。

3. Docker デーモンでビルドしたイメージをテストします。

```
$ docker run -d --name <new-name> -p <port-number>:<port-number> <output-image-name>
$ curl localhost:<port-number>
```

4. 新しいイメージをOpenShiftレジストリーにプッシュします。

5. **oc** コマンドを使用して、OpenShift レジストリーのイメージから新規アプリケーションを作成します。

```
$ oc new-app <image-name>
```

6.8. サポートガイド

6.8.1. 概要

以下のトピックでは、OpenShift バージョン 2 (v2) および OpenShift バージョン 3 (v3) でサポート対象の言語、フレームワーク、データベース、マーカーについて説明します。

OpenShift Container Platform のお客様が使用する一般的な組み合わせに関する情報は、「OpenShift Container Platform tested integrations」を参照してください。

6.8.2. サポートされているデータベース

データベースアプリケーションのトピックの「サポート対象のデータベース」セクションを参照してください。

6.8.3. サポート言語

- PHP
- Python
- Perl
- Node.js
- Ruby
- JBoss/xPaaS

6.8.4. サポート対象のフレームワーク

表6.1 サポート対象のフレームワーク

v2	v3
Jenkins サーバー	jenkins-persistent
Drupal 7	
Ghost 0.7.5	
WordPress 4	
Ceylon	
Go	
MEAN	

6.8.5. サポート対象のマーカー

表6.2 Python

v2	v3
pip_install	リポジトリーに **requirements.txt** が含まれる場合には、デフォルトで pip が呼び出されます。含まれていない場合に pip は使用されません。

表6.3 Ruby

v2	v3
disable_asset_compilation	これは、**buildconfig** ストラテジー定義で **DISABLE_ASSET_COMPILATION** 環境変数を **true** に設定すると使用できます。

表6.4 Perl

v2	v3
enable_cpan_tests	これは、ビルド設定 で **ENABLE_CPAN_TEST** 環境変数を **true** に設定すると実行できます。

表6.5 PHP

v2	v3
use_composer	ソースリポジトリーの root ディレクトリーに **composer.json** が含まれる場合に、コンポーザーが常に使用されます。

表6.6 Node.js

v2	v3
NODEJS_VERSION	該当なし
use_npm	アプリケーションの起動には、**DEV_MODE** が **true** に設定されていない限り **npm** が常に使用されます。true に設定されていない場合には **nodemon** が使用されます。

表6.7 JBoss EAP, JBoss WS, WildFly

v2	v3
enable_debugging	このオプションは、デプロイメント設定で設定される **ENABLE_JPDA** 環境変数に値を設定することで制御します。
skip_maven_build	**pom.xml** がある場合には、maven が実行されます。
java7	該当なし
java8	JavaEE は JDK8 を使用します。

表6.8 Jenkins

v2	v3
enable_debugging	該当なし

表6.9 all

v2	v3
force_clean_build	v3 には同様の概念が使われています。**buildconfig** の **noCache** フィールドにより、コンテナービルドによる各層の再実行が強制的に実行されます。S2I ビルドでは、**clean build** を示す **incremental** フラグはデフォルトで false になっています。
hot_deploy	Ruby、Python、Perl、PHP、Node.js
enable_public_server_status	該当なし
disable_auto_scaling	自動スケーリングはデフォルトではオフになっていますが、pod auto-scaling でオンにすることができます。

6.8.6. サポート対象の環境変数

- MySQL
- MongoDB
- PostgreSQL

第7章 チュートリアル

7.1. 概要

以下のトピックでは、OpenShift Container Platform でアプリケーションを稼働させる方法や、さまざまな言語とフレームワークについて説明します。

7.2. クイックスタートのテンプレート

7.2.1. 概要

クイックスタートは、OpenShift Container Platform で実行するアプリケーションの基本的なサンプルです。クイックスタートはさまざまな言語やフレームワークが含まれており、サービスのセット、ビルド設定およびデプロイメント設定などで構成されるテンプレートで定義されています。このテンプレートは、必要なイメージやソースリポジトリーを参照して、アプリケーションをビルドし、デプロイします。

クイックスタートを確認するには、テンプレートからアプリケーションを作成します。管理者がすでにこれらのテンプレートを OpenShift Container Platform クラスターにインストールしている可能性がありますが、その場合には、Web コンソールからこれを簡単に選択できます。テンプレートのアップロード、作成、変更に関する情報は、テンプレートのドキュメントを参照してください。

クイックスタートは、アプリケーションのソースコードを含むソースリポジトリーを参照します。クイックスタートをカスタマイズするには、リポジトリーをフォークし、テンプレートからアプリケーションを作成する時に、デフォルトのソースリポジトリー名をフォークしたリポジトリーに置き換えます。これにより、提供されたサンプルのソースではなく、独自のソースコードを使用してビルドが実行されます。ソースリポジトリーでコードを更新し、新しいビルドを起動して、デプロイされたアプリケーションで変更が反映されていることを確認できます。

7.2.2. Web フレームワークのクイックスタートのテンプレート

以下のクイックスタートでは、指定のフレームワークおよび言語の基本アプリケーションを提供します。

- CakePHP: PHP Web フレームワーク (MySQL データベースを含む)
  - テンプレートの定義
  - Source repository
- Dancer: Perl Web フレームワーク (MySQL データベースを含む)
  - テンプレートの定義
  - Source repository
- Django: Python Web フレームワーク (PostgreSQL データベースを含む)
  - テンプレートの定義
  - Source repository
- NodeJS: NodeJS web アプリケーション (MongoDB データベースを含む)
  - テンプレートの定義

- Source repository

- Rails: Ruby Web フレームワーク (PostgreSQL データベースを含む)
  - テンプレートの定義
  - Source repository

7.3. RUBY ON RAILS

7.3.1. 概要

Ruby on Rails は Ruby で記述された一般的な Web フレームワークです。本ガイドでは、OpenShift Container Platform での Rails 4 の使用について説明します。

警告

チュートリアル全体をチェックして、OpenShift Container Platform でアプリケーションを実行するために必要なすべての手順を概観することを強く推奨します。問題に直面した場合には、チュートリアル全体を振り返り、もう一度問題に対応してください。またチュートリアルは、実行済みの手順を確認し、すべての手順が適切に実行されていることを確認するのに役立ちます。

本書では、以下があることを前提としています。

- Ruby/Rails の基本知識
- Ruby 2.0.0+、Rubygems、Bundler のローカルにインストールされたバージョン
- Git の基本知識
- OpenShift Container Platform v3 の実行インスタンス

7.3.2. ローカルのワークステーション設定

まず、OpenShift Container Platform のインスタンスが実行されており、利用できることを確認します。OpenShift Container Platform を稼働させる方法については、「Installation Methods」を確認してください。さらに、**oc** CLI クライアントがインストールされており、コマンドがコマンドシェルからアクセスできることを確認し、メールアドレスおよびパスワードを使用してログインする際にこれを使用できるようにします。

7.3.2.1. データベースの設定

Rails アプリケーションはほぼ常にデータベースと併用されます。ローカル開発の場合は、PostgreSQL データベースを選択してください。PostgreSQL データベースをインストール方法するには、以下を入力します。

```
$ sudo yum install -y postgresql postgresql-server postgresql-devel
```

次に、以下のコマンドでデータベースを初期化する必要があります。

```
$ sudo postgresql-setup initdb
```

このコマンドで **/var/lib/pgsql/data** ディレクトリーが作成され、このディレクトリーにデータが保存されます。

以下を入力してデータベースを起動します。

```
$ sudo systemctl start postgresql.service
```

データベースが実行されたら、**rails** ユーザーを作成します。

```
$ sudo -u postgres createuser -s rails
```

作成をしたユーザーのパスワードは作成されていない点に留意してください。

7.3.3. アプリケーションの作成

Rails アプリケーションをゼロからビルドするには、Rails gem を先にインストールする必要があります。

```
$ gem install rails
Successfully installed rails-4.2.0
1 gem installed
```

Rails gem のインストール後に、PostgreSQL をデータベースとして 指定して新規アプリケーションを作成します。

```
$ rails new rails-app --database=postgresql
```

次に、新規ディレクトリーに移動します。

```
$ cd rails-app
```

アプリケーションがすでにある場合には **pg** (postgresql) gem が **Gemfile** に配置されていることを確認します。配置されていない場合には、gem を追加して **Gemfile** を編集します。

```
gem 'pg'
```

すべての依存関係を含む **Gemfile.lock** を新たに生成するには、以下を実行します。

```
$ bundle install
```

pg gem で **postgresql** データベースを使用することのほかに、**config/database.yml** が **postgresql** アダプターを使用していることを確認する必要があります。

config/database.yml ファイルの **default** セクションを以下のように更新するようにしてください。

```
default: &default
  adapter: postgresql
  encoding: unicode
```

```
  pool: 5
  host: localhost
  username: rails
  password:
```

アプリケーションの開発およびテストデータベースを作成するには、以下の **rake** コマンドを使用します。

```
$ rake db:create
```

これで PostgreSQL サーバーに **development** および **test** データベースが作成されます。

7.3.3.1. Welcome ページの作成

Rails 4 では、静的な **public/index.html** ページが実稼働環境で提供されなくなったので、新たに root ページを作成する必要があります。

welcome ページをカスタマイズするには、以下の手順を実行する必要があります。

- index アクションで **コントローラー** を作成します。
- **welcome** コントローラー **index** アクションの **ビュー** ページを作成します。
- 作成した **コントローラー** と **ビュー** と共にアプリケーションの root ページを提供する **ルート** を作成します。

Rails には、これらの必要な手順をすべて実行するジェネレーターがあります。

```
$ rails generate controller welcome index
```

必要なファイルはすべて作成されたので、**config/routes.rb** ファイルの 2 行目を以下のように編集することのみが必要になります。

```
root 'welcome#index'
```

rails server を実行して、ページが利用できることを確認します。

```
$ rails server
```

ブラウザーで http://localhost:3000 に移動してページを表示してください。このページが表示されない場合は、サーバーに出力されるログを確認してデバッグを行ってください。

7.3.3.2. OpenShift Container Platform のアプリケーションの設定

アプリケーションと OpenShift Container Platform で実行されている PostgreSQL データベースサービスとを通信させるには、環境変数 を使用するように **config/database.yml** の **default** セクションを編集する必要があります。 環境変数は、後のデータベースサービスの作成時に定義します。

編集した **config/database.yml** の **default** セクションに事前定義済みの変数を入力すると、以下のようになります。

```
<% user = ENV.key?("POSTGRESQL_ADMIN_PASSWORD") ? "root" :
ENV["POSTGRESQL_USER"] %>
<% password = ENV.key?("POSTGRESQL_ADMIN_PASSWORD") ?
```

```
ENV["POSTGRESQL_ADMIN_PASSWORD"] : ENV["POSTGRESQL_PASSWORD"] %>
<% db_service = ENV.fetch("DATABASE_SERVICE_NAME","").upcase %>

default: &default
  adapter: postgresql
  encoding: unicode
  # For details on connection pooling, see rails configuration guide
  # http://guides.rubyonrails.org/configuring.html#database-pooling
  pool: <%= ENV["POSTGRESQL_MAX_CONNECTIONS"] || 5 %>
  username: <%= user %>
  password: <%= password %>
  host: <%= ENV["#{db_service}_SERVICE_HOST"] %>
  port: <%= ENV["#{db_service}_SERVICE_PORT"] %>
  database: <%= ENV["POSTGRESQL_DATABASE"] %>
```

最終的なファイルの内容のサンプルについては、「Ruby on Rails アプリケーションの例 config/database.yml」を参照してください。

7.3.3.3. アプリケーションの Git への保存

OpenShift ContainerPlatformにはgitが必要です。インストールしていない場合はインストールする必要があります。

OpenShift Container Platform でアプリケーションをビルドするには通常、ソースコードを git リポジトリーに保存する必要があるため、**git** がない場合にはインストールしてください。

ls -1 コマンドを実行して、Rails アプリケーションのディレクトリーで操作を行っていることを確認します。コマンドの出力は以下のようになります。

```
$ ls -1
app
bin
config
config.ru
db
Gemfile
Gemfile.lock
lib
log
public
Rakefile
README.rdoc
test
tmp
vendor
```

Rails app ディレクトリーでこれらのコマンドを実行して、コードを初期化して、git にコミットします。

```
$ git init
$ git add .
$ git commit -m "initial commit"
```

アプリケーションをコミットしたら、リモートのリポジトリーにプッシュする必要があります。これには、GitHub アカウント が必要です。 このアカウントで 新しいリポジトリーを作成します。

お使いの **git** リポジトリーを参照するリモートを設定します。

```
$ git remote add origin git@github.com:<namespace/repository-name>.git
```

次に、アプリケーションをリモートの git リポジトリーにプッシュします。

```
$ git push
```

7.3.4. アプリケーションの OpenShift Container Platform へのデプロイ

Ruby on Rails アプリケーションをデプロイするには、アプリケーション用に新規のプロジェクトを作成します。

```
$ oc new-project rails-app --description="My Rails application" --display-name="Rails Application"
```

rails-app プロジェクト の作成後、新規プロジェクトの namespace に自動的に切り替えられます。

OpenShift Container Platform へのアプリケーションのデプロイでは 3 つの手順を実行します。

- OpenShift Container Platform の PostgreSQL イメージ からデータベースサービスを作成します。
- OpenShift Container Platform の Ruby 2.0 ビルダーイメージ と Ruby on Rails のソースコードでフロントエンドの サービス を作成して、データベースサービスと接続します。
- アプリケーションのルートを作成します。

7.3.4.1. データベースサービスの作成

Rails アプリケーションには実行中のデータベースサービス が必要です。このサービスには、PostgeSQL データベースイメージを使用します。

データベースサービスを作成するために、oc new-app コマンドを使用します。このコマンドでは、必要な環境変数を渡す必要があります。この環境変数はデータベースコンテナー内で使用します。これらの環境変数は、ユーザー名、パスワード、およびデータベースの名前を設定するために必要です。これらの環境変数の値を任意の値に変更できます。今回設定する変数は以下の通りです。

- POSTGRESQL_DATABASE
- POSTGRESQL_USER
- POSTGRESQL_PASSWORD

これらの変数を設定すると、以下を確認できます。

- 指定の名前のデータベースが存在する
- 指定の名前のユーザーが存在する
- ユーザーは指定のパスワードで指定のデータベースにアクセスできる

以下に例を示します。

```
$ oc new-app postgresql -e POSTGRESQL_DATABASE=db_name -e
POSTGRESQL_USER=username -e POSTGRESQL_PASSWORD=password
```

データベース管理者のパスワードを設定するには、直前のコマンドに以下を追加します。

```
-e POSTGRESQL_ADMIN_PASSWORD=admin_pw
```

このコマンドの進捗を確認するには、以下を実行します。

```
$ oc get pods --watch
```

7.3.4.2. フロントエンドサービスの作成

アプリケーションを OpenShift Container Platform にデプロイするには、**oc new-app** コマンドをもう一度使用して、アプリケーションを配置するリポジトリーを指定する必要があります。 このコマンドでは、「データベースサービスの作成」で設定したデータベース関連の環境変数を指定してください。

```
$ oc new-app path/to/source/code --name=rails-app -e POSTGRESQL_USER=username -e
POSTGRESQL_PASSWORD=password -e POSTGRESQL_DATABASE=db_name -e
DATABASE_SERVICE_NAME=postgresql
```

このコマンドでは、OpenShift Container Platform は、ソースコードの取得、ビルダーイメージの設定、アプリケーションイメージの ビルド、新規作成したイメージと指定の 環境変数 のデプロイを行います。このアプリケーションは **rails-app** という名前に指定します。

rails-app DeploymentConfig の JSON ドキュメントを参照して、環境変数が追加されたかどうかを確認できます。

```
$ oc get dc rails-app -o json
```

以下のセクションが表示されるはずです。

```
env": [
    {
        "name": "POSTGRESQL_USER",
        "value": "username"
    },
    {
        "name": "POSTGRESQL_PASSWORD",
        "value": "password"
    },
    {
        "name": "POSTGRESQL_DATABASE",
        "value": "db_name"
    },
    {
        "name": "DATABASE_SERVICE_NAME",
        "value": "postgresql"
    }

],
```

ビルドプロセスを確認するには、以下を実行します。

```
$ oc logs -f build rails-app-1
```

ビルドが完了すると、OpenShift Container Platform で Pod が実行されていることを確認できます。

```
$ oc get pods
```

myapp-<number>-<hash> で始まる行が表示されますが、これは OpenShift Container Platform で実行中のアプリケーションです。

データベースの移行スクリプトを実行してデータベースを初期化してからでないと、アプリケーションは機能しません。これを実行する 2 種類の方法があります。

- 実行中のフロントエンドコンテナーから手動で実行する

最初に rsh コマンドでフロントエンドコンテナーに対して実行します。

```
$ oc rsh <FRONTEND_POD_ID>
```

コンテナー内から移行を実行します。

```
$ RAILS_ENV=production bundle exec rake db:migrate
```

development または **test** 環境で Rails アプリケーションを実行する場合には、 **RAILS_ENV** の環境変数を指定する必要はありません。

- デプロイメント前のライフサイクルフックをテンプレートに追するたとえば、Rails サンプル アプリケーションの フックのサンプル を確認します。

7.3.4.3. アプリケーションのルートの作成

www.example.com などの外部からアクセスできるホスト名を指定してサービスを公開するには、OpenShift Container Platform のルートを使用します。この場合は、以下を入力してフロントエンドサービスを公開する必要があります。

```
$ oc expose service rails-app --hostname=www.example.com
```

警告

ユーザーは指定したホスト名がルーターの IP アドレスに解決することを確認する必要があります。詳しい情報は、以下に関する OpenShift Container Platform ドキュメントを参照してください。

- ルート
- 高可用性ルートシステムの設定

7.4. MAVEN 用の NEXUS ミラーリングの設定

7.4.1. はじめに

Java および Maven でアプリケーションを開発すると、ビルドを複数回実行する可能性が非常に高くなります。Pod のビルド時間を短縮するために、Maven の依存関係をローカルの Nexus リポジトリーにキャッシュすることができます。このチュートリアルでは、クラスター上に Nexus リポジトリーを作成する方法を説明します。

このチュートリアルでは、ご利用のプロジェクトが Maven で使用できるように設定されていることを前提としています。Java プロジェクトで Maven を使用する場合は、Maven のガイドを参照することを強く推奨します。

また、アプリケーションのイメージに Maven ミラーリング機能があるか確認するようにしてください。Maven を使用するイメージの多くに **MAVEN_MIRROR_URL** 環境変数が含まれており、このプロセスを単純化するために使用できます。この機能が含まれていない場合には、Nexus ドキュメント を参照して、ビルドが正しく設定されていることを確認してください。

さらに、各 Pod が機能するように十分なリソースを割り当てるようにしてください。追加のリソースを要求するには、Nexus デプロイメント設定で Pod テンプレートを編集する必要がある場合があります。

7.4.2. Nexus の設定

1. 正式な Nexus コンテナーイメージをダウンロードし、デプロイします。

```
oc new-app sonatype/nexus
```

2. 新規作成した Nexus サービスを公開して、ルートを作成します。

```
oc expose svc/nexus
```

3. **oc get routes** を使用して、Pod の新規外部アドレスを検索します。

```
oc get routes
```

出力は以下のようになります。

```
NAME      HOST/PORT                          PATH      SERVICES   PORT       TERMINATION
nexus     nexus-myproject.192.168.1.173.xip.io            nexus      8081-tcp
```

4. ブラウザーで **HOST/PORT** の対象の URL に移動して、Nexus が実行されていることを確認します。Nexus にサインインするには、デフォルトの管理者ユーザー名 **admin**、パスワード **admin123** を使用します。

注記

Nexus は中央リポジトリー用に事前に設定されていますが、アプリケーション用に他のリポジトリーが必要な場合があります。Red Hat イメージの多くは、Maven リポジトリー に **jboss-ga リポジトリーを追加** することを推奨します。

7.4.2.1. プローブを使用した正常な実行の確認

ここで readiness プローブと liveness プローブ を設定することができます。これらのプローブは、Nexus が正しく実行されていることを定期的に確認します。

```
$ oc set probe dc/nexus \
```

```
 --liveness \
 --failure-threshold 3 \
 --initial-delay-seconds 30 \
 -- echo ok
$ oc set probe dc/nexus \
 --readiness \
 --failure-threshold 3 \
 --initial-delay-seconds 30 \
 --get-url=http://:8081/nexus/content/groups/public
```

7.4.2.2. Nexus への永続性の追加

注記

永続ストレージを必要としない場合には、Connecting to Nexus に進みます。ただし、Pod が何らかの理由で再起動された場合には、キャッシュされた依存関係および設定のカスタマイズはなくなります。

Nexus の Persistent Volume Claim (永続ボリューム要求、PVC) を作成し、サーバーを実行中の Pod を中断すると、キャッシュされた依存関係が失われないようにします。PVC にはクラスター内で利用可能な永続ボリューム (PV) が必要です。利用可能な PV がない場合や、クラスターに管理者としてのアクセス権限がない場合には、システム管理者に、読み取り/書き込み可能な永続ボリュームを作成するように依頼してください。

永続ボリュームの作成手順については、「Persistent Storage in OpenShift Container Platform 」を参照してください。

Nexus デプロイメント設定に PVC を追加します。

```
$ oc volumes dc/nexus --add \
 --name 'nexus-volume-1' \
 --type 'pvc' \
 --mount-path '/sonatype-work/' \
 --claim-name 'nexus-pv' \
 --claim-size '1G' \
 --overwrite
```

これで、デプロイメント設定の以前の **emptyDir** ボリュームが削除され、1GB 永続ストレージを **/sonatype-work** (依存関係の保存先) にマウントする要求を追加します。この設定の変更により、Nexus Pod は自動的に再デプロイされます。

Nexus が実行していることを確認するには、ブラウザーで Nexus ページを更新します。以下を使用して、デプロイメントの進捗をモニタリングすることができます。

```
$ oc get pods -w
```

7.4.3. Nexus への接続

次の手順では、新しい Nexus リポジトリーを使用するビルドを定義する方法を説明します。残りのチュートリアルでは、このリポジトリーサンプル と、ビルダーとして **wildfly-100-centos7**を使用しますが、これらの変更はどのプロジェクトでも機能します。

ビルダーイメージサンプル では、環境の一部として **MAVEN_MIRROR_URL** をサポートするため、こ

れを使用して、ビルダーイメージを Nexus リポジトリーにポイントすることができます。イメージが環境変数を使用した Mavin のミラーリングをサポートしていない場合には、Nexus ミラーリングを参照する正しい Maven 設定を指定するようにビルダーイメージを変更する必要がある場合があります。

```
$ oc new-build openshift/wildfly-100-centos7:latest~https://github.com/openshift/jee-ex.git \
 -e MAVEN_MIRROR_URL='http://nexus.<Nexus_Project>:8081/nexus/content/groups/public'
$ oc logs build/jee-ex-1 --follow
```

<Nexus_Project> は Nexus リポジトリーのプロジェクト名に置き換えます。これが使用するアプリケーションと同じプロジェクトに含まれる場合には、**<Nexus_Project>.** を削除できます。OpenShift Container Platform の DNS 解決について参照してください。

7.4.4. 正常な実行の確認

web ブラウザーで、**http://<NexusIP>:8081/nexus/content/groups/public** に移動して、アプリケーションの依存関係を保存したことを確認します。また、ビルドログを確認して Maven が Nexus ミラーリングを使用しているかどうかをチェックできます。正常にミラーリングされている場合には、URL **http://nexus:8081** を参照する出力が表示されるはずです。

7.4.5. その他のリソース

- OpenShift Container Platform でのボリュームの管理
- OpenShift Container Platform での Java ビルドの構築時間の改善
- Nexus リポジトリーのドキュメント

7.5. OPENSHIFT PIPELINE ビルド

7.5.1. はじめに

シンプルな web サイトを作成する場合も、複雑なマイクロサービス web を作成する場合も、OpenShift Pipeline を使用して、OpenShift でアプリケーションをビルド、テスト、デプロイ、プロモートを実行します。

標準の Jenkins Pipeline 構文のほかにも、OpenShift Jenkins イメージは (OpenShift Jenkins Client プラグインを使用して) OpenShift の Domain Specific Language (DSL) を提供します。 これは、OpenShift API サーバーと高度な対話を行う、読み取り可能でコンパクトで総合的な、かつ Fluent (流れるような) 構文を提供することを目的とし、OpenShift クラスターのアプリケーションのビルド、デプロイメント、プロモートのより詳細な制御が可能になります。

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

7.5.2. Jenkins Master の作成

Jenkins master を作成するには以下を実行します。

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

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

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

注記

Jenkins の自動プロビジョニングがクラスターで有効化されており、Jenkins master をカスタマイズする必要がない場合には、以前の手順を省略できます。

Jenkins自動プロビジョニングの詳細については、「Configuring Pipeline Execution」を参照してください。

7.5.3. Pipeline のビルド設定

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

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。

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

Pipeline ビルドストラテジーに関する情報は、「Pipeline ストラテジーオプション」を参照してください。

7.5.4. Jenkinsfile

jenkinsPipelineStrategy で BuildConfig を作成したら、インラインの **jenkinsfile** を使用して、Pipeline に指示を出します。この例では、アプリケーションに Git リポジトリーを設定しません。

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

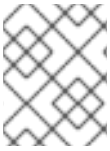
完了した BuildConfig は、OpenShift Origin リポジトリーの examples ディレクトリーの **nodejs-sample-pipeline.yaml** で確認できます。

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

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



注記

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

7.5.5. パイプラインの作成

OpenShift の BuildConfig を作成するには、以下を実行します。

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

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。

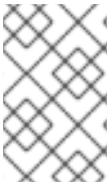
```
$ oc create -f
https://raw.githubusercontent.com/openshift/origin/master/examples/jenkins/pipeline/nodejs-sample-pipeline.yaml
```

使用する OpenShift DSL 構文の情報は、「OpenShift Jenkins クライアントプラグイン」を参照してください。

7.5.6. パイプラインの起動

以下のコマンドでパイプラインを起動します。

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



注記

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

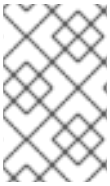
パイプラインが起動したら、以下のアクションがプロジェクト内で実行されるはずです。

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

        - パイプラインは、デプロイメントが完了して次のステージをトリガーするまで待機します。

    - ビルドとデプロイに成功すると、**nodejs-mongodb-example:latest** イメージが **nodejs-mongodb-example:stage** としてトリガーされます。

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



注記

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

7.5.7. OpenShift Pipeline の詳細オプション

OpenShift Pipeline では、Jenkins を 1 つのプロジェクトで起動してから、OpenShift Sync プラグインに開発者が作業をするプロジェクトのグループをモニタリングさせることができます。以下のセクションでは、このプロセスを完了する手順を説明します。

- Jenkins auto = Provisioningを無効にするには、「Configuring Pipeline Execution」を参照してください。
- JenkinsサービスアカウントがOpenShiftPipelinesを実行する各プロジェクトにアクセスできるようにするには、「Cross Project Access」を参照してください。
- モニタリングするプロジェクトを追加するには、以下のいずれかを行います。
    - Jenkins コンソールにログインします。
        - **Manage Jenkins** から、**Configure System** に移動します。
        - **OpenShift Jenkins Sync** の **Namespace** フィールドを更新します。
    - または、S2I 拡張オプションを使用して OpenShift Jenkins イメージを拡張して Jenkins 設定ファイルを更新します。

注記

OpenShift Sync プラグインを実行する複数の Jenkins デプロイメントから、同じプロジェクトのモニタリングがされないようにします。これらのインスタンスは相互に連携していないため、予期せぬ結果が発生する可能性があります。

7.6. バイナリービルド

7.6.1. はじめに

OpenShift のバイナリービルドの機能では、開発者はビルドで Git リポジトリーの URL からソースをプルするのではなく、ソースまたはアーティファクトをビルドに直接アップロードします。ソース、Docker またはカスタムのストラテジーが指定された BuildConfig はバイナリービルドとして起動できます。ローカルのアーティファクトからビルドを起動する場合は、既存のソース参照をローカルユーザーのマシンのソースに置き換えます。

ソースは複数の方法で提供できます。 これは、start-build コマンドの使用時に利用可能な引数に相当します。

- ファイルから (**--from-file**): これは、ビルドのソース全体が単一ファイルで構成されている場合です。たとえば、Docker ビルドは **Dockerfile**、Wildfly ビルドは **pom.xml**、Ruby ビルドは **Gemfile** です。

- ディレクトリーから (**--from-directory**): ソースがローカルのディレクトリーにあり、Git リポジトリーにコミットされていない場合に使用します。**start-build** コマンドは指定のディレクトリーのアーカイブを作成して、ビルダーにソースとしてアップロードします。

- アーカイブから (**--from-archive**): ソースが含まれるアーカイブがすでに存在する場合に使用します。アーカイブは**tar**、**tar.gz** または **zip** 形式のいずれかを使用できます。

- Git リポジトリーから (**--from-repo**): これはソースがユーザーのローカルマシンで Git リポジトリーの一部となっている場合に使用します。現在のリポジトリーの HEAD コミットがアーカイブされ、ビルド用に OpenShift に送信されます。

7.6.1.1. 使用例

バイナリービルドの場合は、ビルドでソースを既存の git リポジトリーからプルする必要がありません。バイナリービルドを使用する理由は以下のとおりです。

- ローカルコードの変更をビルドし、テストする。パブリックリポジトリーからのソースはクローンでき、ローカルの変更を OpenShift にアップロードしてビルドできます。ローカルの変更はコミットまたはプッシュする必要はありません。

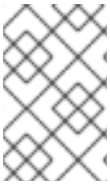
- プロイベートコードをビルドする。新規ビルドをゼロからバイナリービルドとして起動することができます。ソースは、SCM にチェックインする必要なく、ローカルのワークステションから OpenShift に直接アップロードできます。

- 別のソースからアーティファクトを含むイメージをビルドする。Jenkins Pipeline では、Maven または C コンパイラー、これらのビルドを活用するランタイムイメージなどのツールでビルドしたアーティファクトを組み合わせる場合に、バイナリービルドが役立ちます。

7.6.1.2. 制限

- バイナリービルドは反復できません。バイナリービルドは、ビルドの開始時にアーティファクトをアップロードするユーザーに依存するため、そのユーザーが毎回同じアップロードを繰り返さない限り、OpenShift は同じビルドを反復できません。

- バイナリービルドは自動的にトリガーできません。バイナリービルドは、ユーザーが必要なバイナリーアーティファクトをアップロードする時にのみ手動で起動できます。



注記

バイナリービルドとして起動したビルドには設定済みのソース URL が含まれる場合があります。その場合、トリガーでビルドが正常に起動しますが、ソースはビルドの最終実行時にユーザーが指定した URL ではなく、設定済みのソース URL から取得されます。

7.6.2. チュートリアルの概要

以下のチュートリアルは、OpenShift クラスターが利用可能であり、アーティファクトを作成できるプロジェクトが用意されていることを前提としています。このチュートリアルでは、**git** と **oc** がローカルで使用できる必要があります。

7.6.2.1. チュートリアル: ローカルコードの変更のビルド

1. 既存のソースリポジトリーをベースにして新規アプリケーションを作成し、そのルートを作成します。

```
$ oc new-app https://github.com/openshift/ruby-hello-world.git
$ oc expose svc/ruby-hello-world
```

2. 初期ビルドが完了するまで待機し、ルートのホストに移動してアプリケーションのページを表示します。Welcome ページが表示されるはずです。

```
$ oc get route ruby-hello-world
```

3. リポジトリーをローカルにクローンします。

```
$ git clone https://github.com/openshift/ruby-hello-world.git
$ cd ruby-hello-world
```

4. アプリケーションのビューに変更を加えます。任意のエディターで **views/main.rb** を編集します。 **<body>** タグを **<body style="background-color:blue">** に変更します。

5. ローカルで変更したソースで新規ビルドを起動します。リポジトリーのローカルディレクトリーから、以下を実行します。

```
----
$ oc start-build ruby-hello-world --from-dir="." --follow
----
```

ビルドが完了し、アプリケーションが再デプロイされたら、アプリケーションのルートホストに移動すると、青のバックグラウンドのページが表示されるはずです。

ローカルでさらに変更を加えて、**oc start-build --from-dir** でコードをビルドします。

また、コードのブランチを作成し、変更をローカルでコミットし、リポジトリーの HEAD をビルドのソースとして使用します。

```
$ git checkout -b my_branch
$ git add .
$ git commit -m "My changes"
$ oc start-build ruby-hello-world --from-repo="." --follow
```

7.6.2.2. チュートリアル: プライベートコードのビルド

1. コードを保存するローカルディレクトリーを作成します。

```
$ mkdir myapp
$ cd myapp
```

2. このディレクトリーで、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 centos:centos7

EXPOSE 8080
```

```
COPY index.html /var/run/web/index.html

CMD cd /var/run/web && python -m SimpleHTTPServer 8080
```

3. 以下の内容を含む **index.html** という名前のファイルを作成します。

```
<html>
  <head>
    <title>My local app</title>
  </head>
  <body>
    <h1>Hello World</h1>
    <p>This is my local application</p>
  </body>
</html>
```

4. アプリケーションの新規ビルドを作成します。

```
$ oc new-build --strategy docker --binary --docker-image centos:centos7 --name myapp
```

5. ローカルディレクトリーの内容を使用して、バイナリービルドを起動します。

```
$ oc start-build myapp --from-dir . --follow
```

6. **new-app** を使用してアプリケーションをデプロイしてから、そのルートを作成します。

```
$ oc new-app myapp
$ oc expose svc/myapp
```

7. ルートのホスト名を取得して、そこに移動します。

```
$ oc get route myapp
```

コードをビルドし、デプロイした後に、ローカルファイルに変更を加えて、**oc start-build myapp --from-dir** を呼び出して新規ビルドを起動します。ビルドされると、コードが自動的にデプロイされ、ページを更新すると、変更がブラウザーに反映されます。

7.6.2.3. チュートリアル: パイプラインからのバイナリーアーティファクト

OpenShift の Jenkins では、適切なツールでスレーブイメージを使用して、コードをビルドすることができます。たとえば、**maven** スレーブを使用して、コードリポジトリーから WAR をビルドできます。ただし、このアーティファクトがビルドされたら、コードを実行するための適切なランタイムアーティファクトが含まれるイメージにコミットする必要があります。これらのアーティファクトをランタイムイメージに追加するために、バイナリービルドが使用される場合があります。以下のチュートリアルでは、**maven** スレーブで WAR をビルドし、**Dockerfile** でバイナリービルドを使用してこの WAR を WIldfly のランタイムイメージに追加するように Jenkins パイプラインを作成します。

1. アプリケーションの新規ディレクトリーを作成します。

```
$ mkdir mavenapp
$ cd mavenapp
```

2. WAR を wildfly イメージ内の適切な場所にコピーする **Dockerfile** を作成します。以下を **Dockerfile** という名前のローカルファイルにコピーします。

```
FROM wildfly:latest
COPY ROOT.war /wildfly/standalone/deployments/ROOT.war
CMD  $STI_SCRIPTS_PATH/run
```

3. Dockerfile の新規 BuildConfig を作成します。

注記

これにより、ビルドが自動的に起動しますが、**ROOT.war** アーティファクトがまだ利用できないので初回は失敗します。以下のパイプラインでは、バイナリービルドを使用してその WAR をビルドに渡します。

```
$ cat Dockerfile | oc new-build -D - --name mavenapp
```

4. Jenkins Pipeline で BuildConfig を作成します。この BuildConfig では WAR をビルドし、以前に作成した **Dockerfile** を使用してこの WAR でイメージをビルドします。ツールのセットでバイナリーアーティファクトをビルドしてから、最終的なパッケージ用に別のランタイムイメージと組み合わせる場合など、同じパターンを別のプラットフォームでも使用できます。 以下のコードを **mavenapp-pipeline.yml** に保存します。

```
apiVersion: v1
kind: BuildConfig
metadata:
  name: mavenapp-pipeline
spec:
  strategy:
    jenkinsPipelineStrategy:
      jenkinsfile: |-
        pipeline {
          agent { label "maven" }
          stages {
            stage("Clone Source") {
              steps {
                checkout([$class: 'GitSCM',
                          branches: [[name: '*/master']],
                          extensions: [
                            [$class: 'RelativeTargetDirectory', relativeTargetDir: 'mavenapp']
                          ],
                          userRemoteConfigs: [[url: 'https://github.com/openshift/openshift-jee-
sample.git']]
                        ])
              }
            }
            stage("Build WAR") {
              steps {
                dir('mavenapp') {
                  sh 'mvn clean package -Popenshift'
                }
              }
            }
            stage("Build Image") {
```

```
            steps {
              dir('mavenapp/target') {
                sh 'oc start-build mavenapp --from-dir . --follow'
              }
            }
          }
        }
      }
  type: JenkinsPipeline
triggers: []
```

5. Pipeline ビルドを作成します。Jenkins がプロジェクトにデプロイされていない場合は、パイプラインが含まれる BuildConfig を作成すると、Jenkins がデプロイされます。Jenkins がパイプラインをビルドする準備ができるまで、2 分ほどかかる場合があります。Jenkins のロールアウトの状況を確認するには、**oc rollout status dc/jenkins** を起動します。

```
$ oc create -f ./mavenapp-pipeline.yml
```

6. Jenkins の準備ができたら、以前に定義したパイプラインを起動します。

```
$ oc start-build mavenapp-pipeline
```

7. パイプラインがビルドを完了した時点で、new-app で新規アプリケーションをデプロイし、ルートを公開します。

```
$ oc new-app mavenapp
$ oc expose svc/mavenapp
```

8. ブラウザーで、アプリケーションのルートに移動します。

```
$ oc get route mavenapp
```

第8章 ビルド

8.1. ビルドの仕組み

8.1.1. ビルドの概要

OpenShift Container Platform での **ビルド** とは、入力パラメーターをオブジェクトに変換するプロセスのことです。多くの場合に、ビルドを使用して、ソースコードを実行可能なコンテナーイメージに変換します。

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。ストラテジーは前述のプロセスを決定し、ソースは入力内容を提供します。

ビルドストラテジーには、以下が含まれます。

- Source-to-Image (S2I) (説明、オプション)
- Pipeline (説明、オプション)
- Docker (説明、オプション)
- カスタム (説明、オプション)

ビルド入力として指定できるソースは 6 種類あります。

- Git
- Dockerfile
- バイナリー
- イメージ
- 入力シークレット
- 外部アーティファクト

ビルドストラテジーごとに、特定タイプのソースを検討するか、または無視するかや、そのソースタイプの使用方法が決まります。バイナリーおよび Git のソースタイプは併用できません。Dockerfile とイメージは、そのまま単体で使用することも、2 つを併用することも、 Git またはバイナリーと組み合わせることも可能です。バイナリーのソースタイプは、他のオプションと比べると システムへの指定方法の面で独特のタイプです。

8.1.2. BuildConfig の概要

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 トリガーの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。

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** を構成する部分や利用可能なオプションを理解しておくと、後に設定を手動で調整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: "v1"
metadata:
  name: "ruby-sample-build" 1
spec:
  runPolicy: "Serial" 2
  triggers: 3
    -
      type: "GitHub"
      github:
        secret: "secret101"
    - type: "Generic"
      generic:
        secret: "secret101"
    -
      type: "ImageChange"
  source: 4
    git:
      uri: "https://github.com/openshift/ruby-hello-world"
  strategy: 5
    sourceStrategy:
      from:
        kind: "ImageStreamTag"
        name: "ruby-20-centos7:latest"
  output: 6
    to:
      kind: "ImageStreamTag"
      name: "origin-ruby-sample:latest"
  postCommit: 7
      script: "bundle exec rake test"
```

1. この仕様は、**ruby-sample-build** という名前の新規の **BuildConfig** を作成します。
2. **runPolicy** フィールドは、このビルド設定に基づいて作成されたビルドを同時に実行できるかどうかを制御します。デフォルトの値は **Serial** です。 これは新規ビルドが同時にではなく、順番に実行されることを意味します。
3. 新規ビルドを作成するトリガーの一覧を指定できます。
4. **source** セクションでは、ビルドのソースを定義します。ソースの種類は入力の主なソースを決定し、**Git** (コードのリポジトリーの場所を参照)、**Dockerfile** (インラインの Dockerfile からビルド) または **Binary** (バイナリーペイロードを受け入れる) のいずれかとなっています。複数のソースを一度に指定できます。 詳細は、各ソースタイプのドキュメントを参照してください。
5. **strategy** セクションでは、ビルドの実行に使用するビルドストラテジーを記述します。ここでは **Source**、**Docker** または **Custom** ストラテジーを指定できます。上記の例では、Source-To-Image がアプリケーションのビルドに使用する **ruby-20-centos7** コンテナーイメージを使用します。
6.

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

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

8.2. 基本的なビルド操作

8.2.1. ビルドの開始

以下のコマンドを使用して、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します。

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

--from-build フラグを使用してビルドを再度実行します。

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

--follow フラグを指定して、stdout のビルドのログをストリームします。

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

--env フラグを指定して、ビルドに任意の環境変数を設定します。

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

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。

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

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

8.2.2. 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、ビルドを手動でキャンセルします。

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

複数のビルドを同時にキャンセルします。

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

ビルド設定から作成されたビルドすべてをキャンセルします。

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

特定の状態にあるビルドをすべてキャンセルします (例: **new** または **pending**)。 この際、他の状態のビルドは無視されます。

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

8.2.3. BuildConfig の削除

以下のコマンドで **BuildConfig** を削除します。

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

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

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

8.2.4. 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。

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

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。

8.2.5. 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。

ビルドを直接使用してログをストリームするには、以下を実行します。

```
$ oc logs -f build/<build_name>
```

ビルド設定の最新ビルドのログをストリームするには、以下を実行します。

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

ビルド設定で指定されているバージョンのビルドに関するログを返すには、以下を実行します。

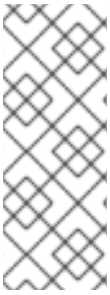
```
$ oc logs --version=<number> bc/<buildconfig_name>
```

ログの詳細レベル

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。



注記

プラットフォームの管理者は、**BuildDefaults** 受付コントローラーの **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。これはデフォルトの設定です。
レベル 1	実行したプロセスに関する基本情報を生成します。
レベル 2	実行したプロセスに関する詳細情報を生成します。
レベル 3	実行したプロセスに関する詳細情報と、アーカイブコンテンツの一覧を生成します。
レベル 4	現時点ではレベル 3 と同じ情報を生成します。
レベル 5	これまでのレベルで記載したすべての内容と docker のプッシュメッセージを提供します。

8.3. ビルド入力

8.3.1. ビルド入力の仕組み

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。OpenShift Cotainer Platform では複数の方法でソースを提供します。以下に優先順に記載します。

- 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

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

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

ビルドが実行されるたびに、以下が行われます。

1. 作業ディレクトリーが作成され、すべての入力内容がその作業ディレクトリーに配置されます。たとえば、入力 Git リポジトリーのクローンはこの作業ディレクトリーに作成され、入力イメージから指定されたファイルはターゲットのパスを使用してこの作業ディレクトリーにコピーされます。
2. ビルドプロセスによりディレクトリーが **contextDir** に変更されます (定義されている場合)。
3. インライン Dockerfile がある場合は、現在のディレクトリーに書き込まれます。

4. 現在の作業ディレクトリーにある内容が Dockerfile、カスタムビルダーのロジック、または **assemble** スクリプトが参照するビルドプロセスに提供されます。つまり、ビルドでは **contextDir** 内にない入力コンテンツは無視されます。

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

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

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

8.3.2. Dockerfile ソース

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

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。

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。

8.3.3. 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。入力イメージは **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 (ある場合) のクローンが作成されるディレクトリーと同じです。ソースパスの末尾は /. であり、ディレクトリーのコンテンツがコピーされますが、ディレクトリー自体は宛先で作成されません。

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

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

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

注記

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

8.3.4. Git ソース

指定されている場合には、ソースコードが指定先の場所からフェッチされます。

インラインの Dockerfile がサポートされる場合には、git リポジトリー **contextDir** 内にある **Dockerfile** (ある場合) が上書きされます。

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

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

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

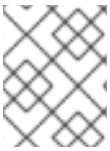
3. オプションの **dockerfile** フィールドがある場合は、Dockerfile を含む文字列を指定してください。 この文字列は、ソースリポジトリーに存在する可能性のある Dockerfile を上書きします。

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**)。

8.3.4.1. プロキシーの使用

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

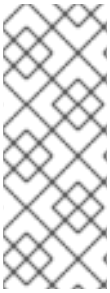


注記

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

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

クラスター管理者は、Ansible を使用して Git クローンにグローバルプロキシーを設定する こともできます。



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。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。

8.3.4.2. 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。

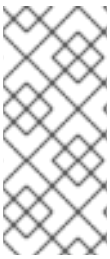
- .gitconfig ファイル
- Basic 認証
- SSH キー認証
- 信頼されている認証局

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。

ビルドは **builder** サービスアカウントで実行されます。この builder アカウントには、使用するソースクローンのシークレットに対するアクセスが必要です。以下のコマンドを使用してアクセスを付与できます。

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



注記

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

8.3.4.2.1. 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 は自動的にソースクローンのシークレット参照を生成します。この動作により、追加の設定なしに、作成される **Builds** が参照される **Secret** に保存された認証情報を自動的に使用して、リモート git リポジトリーへの認証を行います。

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

URI パターンには以下を含める必要があります。

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

上記のいずれの場合でも、* 文字はワイルドカードと見なされます。

重要

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

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

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

複数の **Secrets** 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/*'
```

8.3.4.2.2. ソースクローンシークレットの手動による追加

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

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

注記

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。

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

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

8.3.4.2.3. .gitconfig ファイル

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してからこれをビルダーサービスアカウントに追加し、**BuildConfig** に追加できます。

.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
```

8.3.4.2.4. セキュアな git 用の .gitconfig ファイル

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。

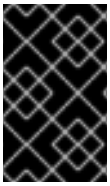
1. **client.crt**、**cacert.crt**、および **client.key** ファイルをアプリケーションソースコードの **/var/run/secrets/openshift.io/source/** フォルダーに追加します。
2. サーバーの **.gitconfig** ファイルに、以下の例のように **[http]** セクションを追加します。

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

3. シークレットを作成します。

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

1. ユーザーの Git ユーザー名
2. このユーザーのパスワード



重要

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

8.3.4.2.5. Basic 認証

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

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

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

トークンで 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。

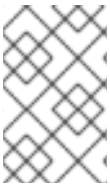
```
$ oc create secret generic <secret_name> \
    --from-literal=password=<token> \
    --type=kubernetes.io/basic-auth
```

8.3.4.2.6. SSH キー認証

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。以下のコマンドで、SSH キーの認証情報を生成します。

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



注記

SSH キーのパスフレーズを作成すると、OpenShift 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。

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

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

8.3.4.2.7. 信頼された認証局

git clone の操作時に信頼される TLS 認証局のセットは 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り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。

以下のプロセスの 1 つを完了します。

- CA 証明書ファイルでシークレットを作成する (推奨)

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

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

  b. シークレットを作成します。

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

  1 **ca.crt** というキーの名前を使用する必要があります。

- git TLS 検証を無効にします。
  ビルド設定の適切なストラテジーセクションで **GIT_SSL_NO_VERIFY** 環境変数を **true** に設定します。**BuildConfig** 環境変数を管理するには、**oc set env** コマンドを使用できます。

8.3.4.2.8. 組み合わせ

ここでは、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方法についての例を紹介します。

a. **.gitconfig** ファイルで 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
```

b. **.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>
```

c. CA 証明書ファイルで 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=ca.crt=</path/to/file> \
    --type=kubernetes.io/basic-auth
```

d. **.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
```

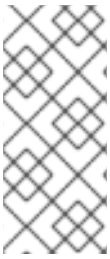
e. **.gitconfig** ファイルと CA 証明書ファイルを合わせて Basic 認証シークレットを作成するには、以下を実行します。

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

8.3.5. 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方法は、**Binary** タイプのビルドと呼ばれています。このビルドについての **BuildConfig.spec.source.type** の対応する値は **Binary** です。

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



注記

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

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

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

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

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

上記のそれぞれの例では、以下のようになります。

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

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

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

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

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

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

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

8.3.6. 入力シークレット

シナリオによっては、ビルド操作において、依存するリソースにアクセスするために認証情報が必要になる場合がありますが、この認証情報をビルドで生成される最終的なアプリケーションイメージで利用可能にすることは適切ではありません。このため、**入力シークレット** を定義することができます。

たとえば、Node.js アプリケーションのビルド時に、Node.js モジュールのプライベートミラーを設定できます。プライベートミラーからモジュールをダウンロードするには、URL、ユーザー名、パスワードを含む、ビルド用のカスタム **.npmrc** ファイルを指定する必要があります。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 Node.js 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。

8.3.6.1. 入力シークレットの追加

入力シークレットを既存の **BuildConfig** に追加するには、以下を実行します。

1. シークレットがない場合は作成します。

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

これにより、**secret-npmrc** 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、**~/.npmrc** ファイルの base64 でエンコードされたコンテンツが含まれます。

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

```
source:
  git:
    uri: https://github.com/openshift/nodejs-ex.git
```

```
    secrets:
      - secret:
          name: secret-npmrc
```

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

```
$ oc new-build \
    openshift/nodejs-010-centos7~https://github.com/openshift/nodejs-ex.git \
    --build-secret secret-npmrc
```

ビルド時に、**.npmrc** 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** をシークレット定義に追加します。

```
source:
  git:
    uri: https://github.com/openshift/nodejs-ex.git
  secrets:
    - secret:
        name: secret-npmrc
      destinationDir: /etc
```

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

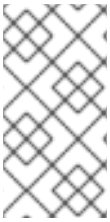
```
$ oc new-build \
    openshift/nodejs-010-centos7~https://github.com/openshift/nodejs-ex.git \
    --build-secret “secret-npmrc:/etc”
```

いずれの場合も、**.npmrc** ファイルがビルド環境の **/etc** ディレクトリーに追加されます。Docker ストラテジー の場合は、宛先のディレクトリーは相対パスでなければならない点に注意してください。

8.3.6.2. Source-to-Image ストラテジー

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

destinationDir が相対パスの場合に同じルールが使用されます。シークレットは、イメージの作業ディレクトリーに対する相対的なパスに配置されます。**destinationDir** が存在する必要があり、存在しない場合はエラーが生じます。コピープロセスでディレクトリーパスは作成されません。



注記

現時点で、これらのシークレットが含まれるすべてのファイル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。

8.3.6.3. Docker ストラテジー

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

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

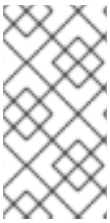
例8.1 シークレットデータを参照する Dockerfile の例

```
FROM centos/ruby-22-centos7

USER root
ADD ./secret-dir /secrets
COPY ./secret2 /

# Create a shell script that will output secrets when the image is run
RUN echo '#!/bin/sh' > /secret_report.sh
RUN echo '(test -f /secrets/secret1 && echo -n "secret1=" && cat /secrets/secret1)' >>
/secret_report.sh
RUN echo '(test -f /secret2 && echo -n "relative-secret2=" && cat /secret2)' >> /secret_report.sh
RUN chmod 755 /secret_report.sh

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



注記

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

8.3.6.4. カスタムストラテジー

Custom ストラテジーを使用する場合、定義された入力シークレットはすべて、**/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。カスタムビルドイメージは、これらのシークレットを適切に使用する必要があります。また、**Custom** ストラテジーを使用すると、カスタムストラテジーのオプションで記載されているようにシークレットを定義できます。

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

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

8.3.7. 外部アーティファクトの使用

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。

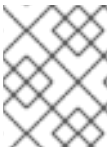
Source ビルドストラテジーの場合は、**assemble** スクリプトに適切なシェルコマンドを設定する必要があります。

.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
```



注記

Source ビルドが使用する **assemble** および **run** スクリプトを制御する方法に関する情報は、「ビルダーイメージスクリプトの上書き」を参照してください。

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

Dockerfile の抜粋

```
FROM jboss/base-jdk:8

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

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

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

環境変数の定義には複数の方法があり、いずれかの方法を選択できます。

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

8.3.8. 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 Docker 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 Docker 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 Docker 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 Docker 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 Docker 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。

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

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

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

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

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

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

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

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

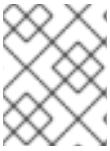
   ```
   $ oc set build-secret --push bc/sample-build dockerhub
   ```

4. ビルドストラテジー定義に含まれる **pullSecret** を指定して、プライベート Docker 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、ビルド設定にプルする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。

8.4. ビルドの出力

8.4.1. ビルド出力の概要

Docker または **Source** 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"
```

8.4.2. アウトプットイメージの環境変数

Docker および **Source** 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

さらに、**Source** または **Docker** ストラテジーオプションで設定されるユーザー定義の環境変数は、アウトプットイメージの環境変数一覧にも含まれます。

8.4.3. アウトプットイメージのラベル

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

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

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

ビルドイメージに適用されるカスタムラベル

```
spec:
  output:
    to:
      kind: "ImageStreamTag"
      name: "my-image:latest"
    imageLabels:
    - name: "vendor"
```

```
      value: "MyCompany"
    - name: "authoritative-source-url"
      value: "registry.mycompany.com"
```

8.4.4. アウトプットイメージのダイジェスト

ビルドイメージは、ダイジェストで一意に識別して、後に現在のタグとは無関係にこれを使用してダイジェスト別にイメージをプルすることができます。

Docker および **Source** ビルドは、イメージがレジストリーにプッシュされた後に **Build.status.output.to.imageDigest** にダイジェストを保存します。ダイジェストはレジストリーで処理されます。そのため、これはレジストリーがダイジェストを返さない場合や、ビルダーイメージで形式が認識されない場合など、存在しないことがあります。

レジストリーへのプッシュに成功した後のビルドイメージのダイジェスト

```
status:
  output:
    to:
      imageDigest:
sha256:29f5d56d12684887bdfa50dcd29fc31eea4aaf4ad3bec43daf19026a7ce69912
```

8.4.5. プライベートレジストリーでの docker 認証情報の使用

シークレットを使用して認証情報を指定することで、プライベート Docker レジストリーにイメージをプッシュすることができます。方法については、「ビルド入力」を参照してください。

8.5. ビルドストラテジーのオプション

8.5.1. Source-to-Image ストラテジーのオプション

以下のオプションは、S2I ビルドストラテジー に固有のオプションです。

8.5.1.1. 強制プル

ビルド設定で指定したビルドイメージがノードでローカルに利用できる場合には、デフォルトではそのイメージが使用されます。ただし、ローカルイメージを上書きして、イメージストリームが参照するレジストリーからイメージを更新する場合には、 **forcePull** フラグを **true** に設定して **BuildConfig** を作成します。

```
strategy:
  sourceStrategy:
    from:
      kind: "ImageStreamTag"
      name: "builder-image:latest" 1
    forcePull: true 2
```

1 使用するビルダーイメージ。 ノードのローカルバージョンは、イメージストリームが参照するレジストリーのバージョンと同様の最新の状態でない可能性があります。

2

このフラグがあると、ローカルのビルダーイメージが無視され、イメージストリームが参照するレジストリーから新しいバージョンがプルされます。**forcePull** を **false** に設定すると、デフォルト

8.5.1.2. 増分ビルド

S2I は増分ビルドを実行できるので、以前にビルドされたイメージからのアーティファクトが再利用されます。増分ビルドを作成するには、ストラテジー定義に以下の変更を加えて **BuildConfig** を作成します。

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

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

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

注記

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に関する説明は、「S2I Requirements」を参照してください。

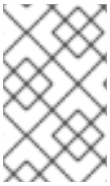
8.5.1.3. ビルダーイメージのスクリプトの上書き

ビルダーイメージが提供する **assemble**、**run**、および **save-artifacts** S2I スクリプト は、2 種類のいずれかの方法で上書きできます。次のいずれかになります。

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

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

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



注記

scripts URL に配置されているファイルは、ソースリポジトリーの **.s2i/bin** に配置されているファイルよりも優先されます。S2I スクリプトがどのように使用されるかについては、S2I 要件 のトピックおよび S2I ドキュメント を参照してください。

8.5.1.4. 環境変数

ソースビルド のプロセスと生成されるイメージで環境変数を利用できるようにする方法として、2 種類 (環境ファイルおよび BuildConfig 環境の値の使用) 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。

8.5.1.4.1. 環境ファイル

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。このファイルで指定された環境変数は、ビルドプロセスとアウトプットイメージに存在します。サポートされる環境変数の完全な一覧は、各イメージの ドキュメント にあります。

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

たとえば、Rails アプリケーションのアセットのコンパイルを無効にする場合には、**.s2i/environment** ファイルに **DISABLE_ASSET_COMPILATION=true** を追加して、ビルド時にアセットのコンパイルがスキップされるようにします。

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、**.s2i/environment** ファイルに **RAILS_ENV=development** を追加して、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します。

8.5.1.4.2. BuildConfig 環境

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

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例:

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

ビルド環境のセクションでは、より詳細な説明を提供します。

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

8.5.1.5. 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. Source-to-Image(S2I)ビルド設定 を作成します。

4. ビルド設定のエディターページや、Web コンソール の **create app from builder image** ページで、**Source Secret** を設定します。

5. **Save** ボタンをクリックします。

8.5.1.5.1. 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にプルできるようにするには、ビルド設定に **Pull Secret** を設定し、プッシュを有効にするには **Push Secret** を設定します。

8.5.1.6. ソースファイルの無視

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

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

8.5.2. Docker ストラテジーのオプション

以下のオプションは、「 Docker ビルドストラテジー 」に固有のオプションです。

8.5.2.1. FROM イメージ

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"
```

8.5.2.2. Dockerfile パス

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

dockerfilePath フィールドでは、異なるパスを使用して Dockerfile の場所 (**BuildConfig.spec.source.contextDir** フィールドへの相対パス) を特定します。デフォルトの **Dockerfile** (例: **MyDockerfile**) とは異なる名前や、サブディレクトリーにある Dockerfile へのパス (例: **dockerfiles/app1/Dockerfile**) などを単純に設定できます。

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

8.5.2.3. キャッシュなし

Docker ビルドは通常、ビルドを実行するホスト上のキャッシュ階層を再利用します。**noCache** オプションを **true** に設定すると、ビルドがキャッシュ階層を無視して、**Dockerfile** のすべての手順を再実行します。

```
strategy:
  dockerStrategy:
    noCache: true
```

8.5.2.4. 強制プル

ビルド設定で指定したビルドイメージがノードでローカルに利用できる場合には、デフォルトではそのイメージが使用されます。ただし、ローカルイメージを上書きして、イメージストリームが参照するレジストリーからイメージを更新する場合には、 **forcePull** フラグを **true** に設定して **BuildConfig** を作成します。

```
strategy:
  dockerStrategy:
    forcePull: true (1)
```

(1) このフラグがあると、ローカルのビルダーイメージが無視され、イメージストリームが参照するレジストリーから新しいバージョンがプルされます。**forcePull** を **false** に設定すると、デフォルトの動作として、ローカルに保存されたイメージが使用されます。

8.5.2.5. 環境変数

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

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

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

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/"
```

クラスター管理者は、Ansible を使用してグローバルビルド設定を設定 することもできます。

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。

8.5.2.6. 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. docker ビルド設定 を作成します。
4. ビルド設定のエディターページまたは、Web コンソール の **fromimage** ページで、**Source Secret** を設定します。
5. **Save** ボタンをクリックします。

8.5.2.7. Docker ビルド引数

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

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

ビルド引数は、ビルドの開始時に Docker に渡されます。

8.5.2.7.1. 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にプルできるようにするには、ビルド設定に **Pull Secret** を設定し、プッシュを有効にするには **Push Secret** を設定します。

8.5.3. カスタムストラテジーのオプション

以下のオプションは、「 カスタムビルドストラテジー 」に固有のオプションです。

8.5.3.1. FROM イメージ

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します。

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

8.5.3.2. Docker ソケットの公開

コンテナー内から Docker コマンドを実行して、コンテナーイメージをビルドできるようにするには、アクセス可能なソケットにビルドコンテナーをバインドする必要があります。これには、**exposeDockerSocket** オプションを **true** に設定します。

```
strategy:
  customStrategy:
    exposeDockerSocket: true
```

8.5.3.3. Secret

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。

各シークレットは、特定の場所にマウントできます。

```
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。

8.5.3.3.1. 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. カスタムビルド設定 を作成します。
4. ビルド設定のエディターページまたは、Web コンソール の **fromimage** ページで、**Source Secret** を設定します。
5. **Save** ボタンをクリックします。

8.5.3.3.2. 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にプルできるようにするには、ビルド設定に **Pull Secret** を設定し、プッシュを有効にするには **Push Secret** を設定します。

8.5.3.4. 強制プル

ビルド Pod を設定する場合に、ビルドコントローラーはデフォルトで、ビルド設定で指定したイメージがローカルで使用できるかどうかを確認します。ローカルで利用できる場合にはそのイメージが使用されます。ただし、ローカルイメージを上書きして、イメージストリームが参照するレジストリーからイメージを更新する場合には、 **forcePull** フラグを **true** に設定して **BuildConfig** を作成します。

```
strategy:
  customStrategy:
    forcePull: true 1
```

1 このフラグがあると、ローカルのビルダーイメージが無視され、イメージストリームが参照するレジストリーから新しいバージョンがプルされます。**forcePull** を **false** に設定すると、デフォルトの動作として、ローカルに保存されたイメージが使用されます。

8.5.3.5. 環境変数

環境変数を カスタムビルド プロセスで利用可能にするには、環境変数を **BuildConfig** の **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。

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

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

クラスター管理者は、Ansible を使用してグローバルビルド設定を設定 することもできます。

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。

8.5.4. パイプラインストラテジーのオプション

以下のオプションは、「 Pipeline ビルドストラテジー 」に固有のオプションです。

8.5.4.1. Jenkinsfile の提供

Jenkinsfile は、以下の 2 つの方法のどちらかで提供できます。

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

埋め込み定義

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

Git リポジトリーへの参照

-

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

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

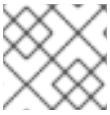
8.5.4.2. 環境変数

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

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

以下に例を示します。

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



注記

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。

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

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

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

Jenkins ジョブのビルドを開始する方法により、パラメーターの設定方法が決まります。**oc start-build** で開始された場合には、**BuildConfig** の環境変数の値は対応するジョブインスタンスに設定するパラメーターになります。Jenkins コンソールからパラメーターのデフォルト値に変更を加えても無視されます。**BuildConfig** の値が優先されます。

oc start-build -e で開始すると、**-e** オプションで指定した環境変数の値が優先されます。また、**BuildConfig** に記載されていない環境変数を指定した場合には、Jenkins ジョブのパラメーター定義として追加されます。また、Jenkins コンソールから環境変数に対応するパラメーターに加える変更

は無視されます。**BuildConfig** および **oc start-build -e** て指定する内容が優先されます。

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

8.6. ビルド環境

8.6.1. 概要

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、以下のような例外があります。

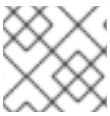
注記

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。

8.6.2. 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。

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



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。

8.6.3. 環境変数としてのコンテナーリソースの使用

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。

8.6.4. 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。

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

```
    secretKeyRef:
      key: myval
      name: mysecret
```

8.7. ビルドのトリガー

8.7.1. ビルドトリガーの概要

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

- Webhook
- イメージの変更
- 設定の変更

8.7.2. 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、Webhook 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) 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、イベント内のブランチ参照が、対応の **BuildConfig** のブランチ参照と一致しているかどうか確認されます。一致する場合には、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、OpenShift Container Platform ビルド用にチェックアウトされます。一致しない場合には、ビルドはトリガーされません。

注記

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

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

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

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

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

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

8.7.2.1. GitHub Webhooks

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

GitHub Webhook の定義例:

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

注記

Webhook トリガーの設定で使用されるシークレットは、GitHub UI で Webhook 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 は以下のように構成されます。

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

GitHub Webhook を設定するには以下を実行します。

1. GitHub リポジトリーから **BuildConfig** を作成した後に、以下を実行します。

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

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

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

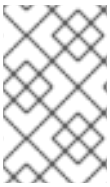
2. GitHub の Web コンソールから、この URL を GitHub にカットアンドペーストします。
3. GitHub リポジトリーで、**Settings → Webhooks & Services** から **Add Webhook** を選択します。
4. **Payload URL** フィールドに、(上記と同様の) URL の出力を貼り付けます。

5. **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。

6. **Add webhook** をクリックします。

webhook の設定が正常に完了したことを示す GitHub のメッセージが表示されます。

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



注記

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

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

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

8.7.2.2. GitLab Webhooks

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

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

oc describe コマンドは、ペイロード URL を GitLab Webhook URL として返します (「Webhook URL の表示」を参照)。 ペイロード URL は以下のように構成されます。

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

GitLab Webhook を設定するには以下を実行します。

1. ビルド設定を記述して、webhook URL を取得します。

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

2. 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。

3. 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。

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

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

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

8.7.2.3. Bitbucket Webhook

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

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

oc describe コマンドは、ペイロード URL を Bitbucket Webhook URL として返します (「Webhook URL の表示」を参照)。ペイロード URL は以下のように構成されます。

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

Bitbucket Webhook を設定するには以下を実行します。

1. ビルド設定を記述して、webhook URL を取得します。

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

2. 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。
3. 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。

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

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

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

8.7.2.4. Generic Webhook

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

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

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

手動で Webhook を呼び出すには、**curl** を使用します。

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

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

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

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

注記

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

8.7.2.5. 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 を表示します。

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

上記のコマンドで webhook URL が表示されない場合には、対象のビルド設定に webhook トリガーが定義されていないことになります。トリガーを手動で追加するには、「トリガーの設定」を参照してください。

8.7.3. イメージ変更のトリガー

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。

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 **docker-registry** の名前で、サービスとして定義されます。

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

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

```
      kind: "ImageStreamTag"
      name: "ruby-20-centos7:latest"
```

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

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

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

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

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

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

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

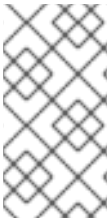
ストラテジーにイメージ変更トリガーが含まれる上記の例では、作成されるビルドは以下のようになります。

```
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。

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。



注記

v1 Docker 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 Docker 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。

8.7.4. 設定変更のトリガー

設定変更のトリガーは、**BuildConfig** がされると同時に自動的にビルドが呼び出されます。以下の例は、**BuildConfig** 内のトリガー定義の YAML です。

```
type: "ConfigChange"
```

注記

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

8.7.4.1. トリガーの手動設定

トリガーは、**oc set triggers** でビルド設定に対して追加/削除できます。たとえば、GitHub webhook トリガーをビルド設定に追加するには以下を使用します。

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

イメージ変更トリガーを設定するには以下を使用します。

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

トリガーを削除するには **--remove** を追加します。

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

注記

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

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

8.8. ビルドフック

8.8.1. ビルドフックの概要

ビルドフックを使用すると、ビルドプロセスに動作を挿入できます。

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

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

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

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

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

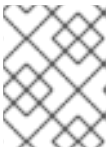
8.8.2. コミット後のビルドフックの設定

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。

- 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。

- デフォルトのエントリーポイントに渡す引数:

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

この形式では、**args** はイメージのデフォルトエントリーポイントに渡される引数一覧です。イメージのエントリーポイントは、引数を処理できる必要があります。

- 引数を指定したシェルスクリプト:

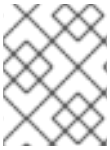
```
postCommit:
  script: "bundle exec rake test $1"
  args: ["--verbose"]
```

引数を渡す必要があるが、シェルスクリプトで正しく引用するのが困難な場合に、この形式を使用します。上記の **script** では、**$0** は "/bin/sh" で、**$1**、**$2** などは **args** の位置引数となります。

- 引数のあるコマンド:

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

この形式は **command** に引数を追加するのと同じです。



注記

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

8.8.2.1. CLI の使用

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。

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
```

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"
```

8.9. ビルド実行ポリシー

8.9.1. ビルド実行ポリシーの概要

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。

- **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。

8.9.2. 順次実行ポリシー

runPolicy フィールドを **Serial** に設定すると、**Build** ビルドから作成される新しいビルドはすべて 順次実行になります。つまり、1度に実行されるビルドは1つだけで、新しいビルドは、前のビルドが完了するまで待機します。このポリシーを使用すると、一貫性があり、予測可能なビルドが出力されます。これは、デフォルトの **runPolicy** です。

Serial ポリシーで sample-build 設定から 3 つのビルドをトリガーすると以下のようになります。

```
NAME           TYPE     FROM         STATUS    STARTED         DURATION
sample-build-1   Source   Git@e79d887   Running   13 seconds ago   13s
sample-build-2   Source   Git           New
sample-build-3   Source   Git           New
```

sample-build-1 ビルドが完了すると、sample-build-2 ビルドが実行されます。

```
NAME           TYPE     FROM         STATUS    STARTED         DURATION
sample-build-1   Source   Git@e79d887   Completed 43 seconds ago   34s
sample-build-2   Source   Git@1aa381b   Running   2 seconds ago    2s
sample-build-3   Source   Git           New
```

8.9.3. SerialLatestOnly 実行ポリシー

runPolicy フィールドを **SerialLatestOnly** に設定すると、**Serial** 実行ポリシーと同様に、**Build** 設定から作成される新規ビルドすべてが順次実行されます。相違点は、現在実行中のビルドの完了後に、実行される次のビルドが作成される最新ビルドになるという点です。言い換えると、キューに入っているビルドはスキップされるので、これらの実行を待機しないということです。スキップされたビルドは Cancelled としてマークされます。このポリシーは、反復的な開発を迅速に行う場合に使用できます。

SerialLatestOnly ポリシーで sample-build 設定から 3 つのビルドをトリガーすると以下のようになります。

```
NAME           TYPE     FROM         STATUS    STARTED         DURATION
sample-build-1   Source   Git@e79d887   Running   13 seconds ago   13s
sample-build-2   Source   Git           Cancelled
sample-build-3   Source   Git           New
```

sample-build-2 のビルドはキャンセル (スキップ) され、sample-build-1 の完了後に、sample-build-3 ビルドが次のビルドとして実行されます。

```
NAME           TYPE     FROM         STATUS    STARTED         DURATION
sample-build-1   Source   Git@e79d887   Completed 43 seconds ago   34s
sample-build-2   Source   Git           Cancelled
sample-build-3   Source   Git@1aa381b   Running   2 seconds ago    2s
```

8.9.4. 並列実行ポリシー

runPolicy フィールドを **Parallel** に設定すると、**Build** 設定から作成される新規ビルドはすべて並列で実行されます。この設定では、最初に作成されるビルドが完了するのが最後になる可能性があり、最新

のイメージで生成され、プッシュされたコンテナーイメージが先に完了してしまい、置き換わる可能性があるので、結果が予想できません。

ビルドの完了する順番が問題とはならない場合には、並列実行ポリシーを使用してください。

Parallel ポリシーで **sample-build** 設定から 3 つのビルドをトリガーすると、3 つのビルドが同時に実行されます。

```
NAME           TYPE     FROM         STATUS    STARTED         DURATION
sample-build-1   Source   Git@e79d887   Running   13 seconds ago   13s
sample-build-2   Source   Git@a76d881   Running   15 seconds ago   3s
sample-build-3   Source   Git@689d111   Running   17 seconds ago   3s
```

完了する順番は保証されません。

```
NAME           TYPE     FROM         STATUS    STARTED         DURATION
sample-build-1   Source   Git@e79d887   Running   13 seconds ago   13s
sample-build-2   Source   Git@a76d881   Running   15 seconds ago   3s
sample-build-3   Source   Git@689d111   Completed 17 seconds ago   5s
```

8.10. 高度なビルド操作

8.10.1. 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。プロジェクトのデフォルトのコンテナー制限に、リソースの制限を指定すると、これらのリソースを制限できます。

ビルド設定の一部にリソース制限を指定して、リソースの使用を制限することも可能です。以下の例では、**resources**、**cpu** および **memory** の各パラメーターはオプションです。

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

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

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

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

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

```
resources:
  requests: 1
    cpu: "100m"
```

```
    memory: "256Mi"
```

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

- プロジェクトで定義される 制限の範囲。**LimitRange** オブジェクト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。

8.10.2. 最長期間の設定

BuildConfig 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。

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。

8.10.3. 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。

```
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。

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** 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。

詳細情報は、「 グローバルビルドのデフォルト設定および上書きの 設定」を参照してください。

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。

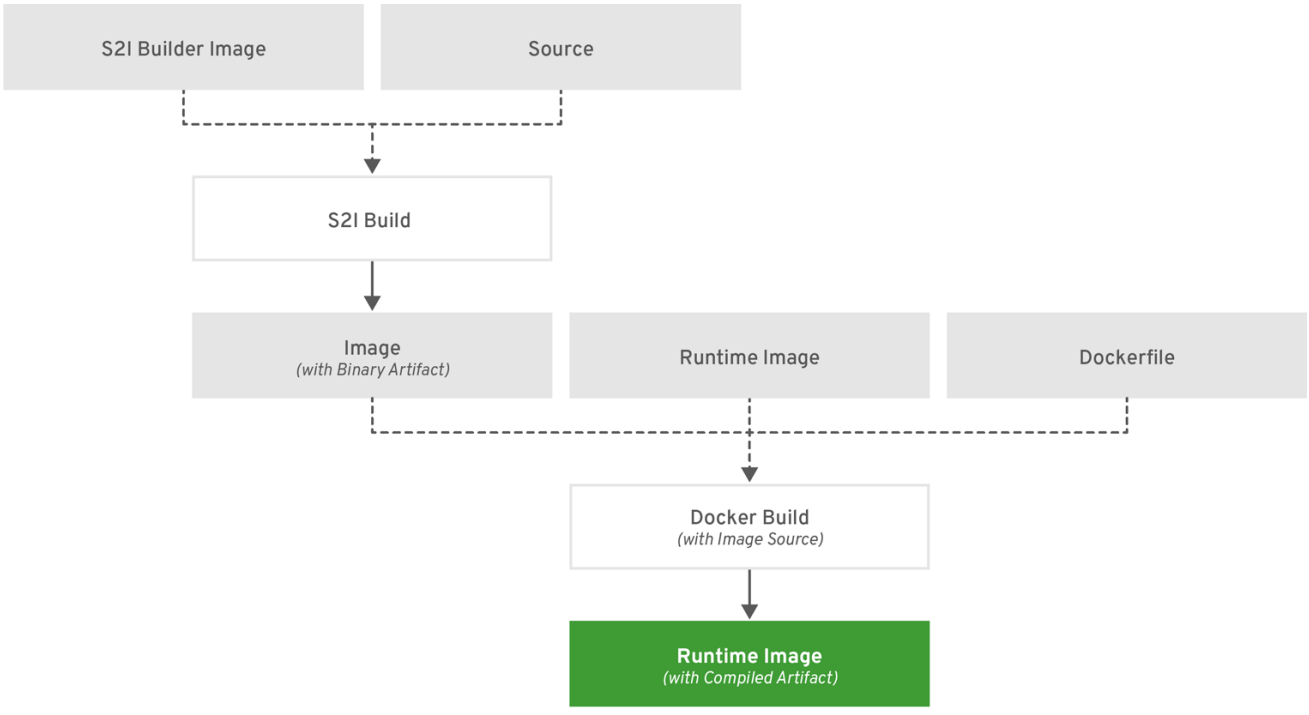
8.10.4. チェーンビルド

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 ビルドが Docker ビルドと組み合わせて、アーティファクトをコンパイルし、別のランタイムイメージに配置します。

注記

この例では、Source-to-Image 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、任意の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可能な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 Source-to-Image 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: 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
    type: Git
  strategy:
    sourceStrategy:
      from:
        kind: ImageStreamTag
        name: wildfly:10.1
        namespace: openshift
    type: Source
```

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、その WAR ファイルをランタイムイメージにコピーします。

```
apiVersion: v1
kind: BuildConfig
metadata:
  name: image-build
spec:
  output:
    to:
      kind: ImageStreamTag
      name: image-build:latest
  source:
    type: Dockerfile
    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
    type: Docker
  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。

8.10.5. ビルドのプルーニング

デフォルトでは、ライフサイクルが完了したビルドは、無限に永続します。以下のビルド設定例にあるように、**successfulBuildsHistoryLimit** または **failedBuildsHistoryLimit** を正の整数に指定すると、以前のビルドを保持する数を制限することができます。

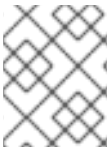
```
apiVersion: "v1"
kind: "BuildConfig"
metadata:
  name: "sample-build"
spec:
  successfulBuildsHistoryLimit: 2 (1)
  failedBuildsHistoryLimit: 2 (2)
```

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。

ビルドプルーニングは、以下のアクションによりトリガーされます。

- ビルド設定が更新された場合
- ビルドのライフサイクルが完了した場合

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' オブジェクトの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。

8.11. ビルドのトラブルシューティング

8.11.1. 拒否されたリソースへのアクセス要求

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
```

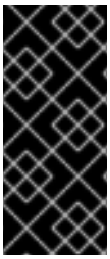
第9章 デプロイメント

9.1. デプロイメントの仕組み

9.1.1. デプロイメントの概要

OpenShift Container Platform デプロイメントでは、一般的なユーザーアプリケーションに対して詳細にわたる管理ができます。デプロイメントは、3 つの異なる API オブジェクトを使用して記述します。

- デプロイメント設定。 Pod テンプレートとして、アプリケーションの特定のコンポーネントに対する状態を記述します。
- 1 つまたは複数のレプリケーションコントローラー。 このコントローラーには、Pod テンプレートとしてデプロイメント設定のある時点の状態が含まれます。
- 1 つまたは複数の Pod。 特定バージョンのアプリケーションのインスタンスを表します。



重要

デプロイメント設定が所有するレプリケーションコントローラーまたは Pod を操作する必要はありません。デプロイメントシステムにより、デプロイメント設定への変更は適切に伝搬されます。既存のデプロイメントストラテジーがユースケースに適さない場合や、デプロイメントのライフサイクルで手動の手順を実行する必要がある場合には、「カスタムストラテジー」の作成を検討してください。

デプロイメント設定を作成すると、レプリケーションコントローラーが、デプロイメント設定の Pod テンプレートとして作成されます。デプロイメント設定が変更されると、最新の Pod テンプレートで新しいレプリケーションコントローラーが作成され、デプロイメントプロセスが実行され、以前のレプリケーションコントローラーにスケールダウンされるか、新しいレプリケーションコントローラーにスケールアップされます。

アプリケーションのインスタンスは、作成時にサービスローダーバランサーやルーターに対して自動的に追加/削除されます。アプリケーションが正常なシャットダウン機能をサポートしている限り、アプリケーションが **TERM** シグナルを受け取ると、実行中のユーザー接続が通常通り完了できるようにすることができます。

デプロイメントシステムで提供される機能:

- 実行中のアプリケーションのテンプレートとなるデプロイメント設定。
- イベントへの対応として自動化されたデプロイメントを駆動するトリガー。
- 以前のバージョンから新しいバージョンに移行するための、ユーザーによるカスタマイズが可能なストラテジー。ストラテジーは、デプロイメントプロセスと一般的に呼ばれる Pod 内で実行されます。
- デプロイメントのライフサイクル中のさまざまなポイントで、カスタムの動作を実行するための フック セット。
- デプロイメントの失敗時に手動または自動でロールバックをサポートするためのアプリケーションのバージョン管理。
- レプリケーションの手動および自動スケーリング。

9.1.2. デプロイメント設定の作成

デプロイメント設定は、OpenShift Container Platform API リソースの **deploymentConfig** で、他のリソースのように **oc** コマンドで管理できます。以下は、**deploymentConfig** リソースの例です。

```
kind: "DeploymentConfig"
apiVersion: "v1"
metadata:
  name: "frontend"
spec:
  template: 1
    metadata:
      labels:
        name: "frontend"
    spec:
      containers:
        - name: "helloworld"
          image: "openshift/origin-ruby-sample"
          ports:
            - containerPort: 8080
              protocol: "TCP"
  replicas: 5 2
  triggers:
    - type: "ConfigChange" 3
    - type: "ImageChange" 4
      imageChangeParams:
        automatic: true
        containerNames:
          - "helloworld"
        from:
          kind: "ImageStreamTag"
          name: "origin-ruby-sample:latest"
  strategy: 5
    type: "Rolling"
  paused: false 6
  revisionHistoryLimit: 2 7
  minReadySeconds: 0 8
```

1. 単純な Ruby アプリケーションを記述する **frontend** デプロイメント設定の Pod テンプレート。
2. **frontend** のレプリカは 5 つとなります。
3. Pod テンプレートが変更されるたびに、新規レプリケーションコントローラーが作成されるようにする設定変更トリガー
4. origin-ruby-sample:latest イメージストリームタグの新規バージョンが利用できるようになると、新しいレプリケーションコントローラーが作成されるようにする**イメージ変更トリガー**
5. ローリングストラテジーは、Pod をデプロイするデフォルトの方法です。このストラテジーは、Pod をデプロイするデフォルトの方法で、省略可能です。
6. デプロイメント設定を一時停止します。これにより、すべてのトリガー機能が無効になり、実際にロールアウトされる前に Pod テンプレートに複数の変更を加えることができます。
7. 改訂履歴の制限。ロールバック用に保持する、以前のレプリケーションコントローラー数の上限です。これは省略可能です。省略した場合には、以前のレプリケーションコントローラーは消去され

ません。

8 (Readiness チェックにパスした後) Pod が利用可能とみなされるまでに待機する最低期間 (秒)。デフォルト値は 0 です。

9.2. 基本のデプロイメント操作

9.2.1. デプロイメントの開始

Web コンソールまたは CLI を使用して手動で新規デプロイメントプロセスを開始できます。

```
$ oc rollout latest dc/<name>
```

注記

デプロイメントプロセスが進行中の場合には、このコマンドを実行すると、メッセージが表示され、新規レプリケーションコントローラーがデプロイされません。

9.2.2. デプロイメントの表示

アプリケーションで利用可能な全リビジョンの基本情報を取得します。

```
$ oc rollout history dc/<name>
```

このコマンドでは、現在実行中のデプロイメントプロセスなど、指定したデプロイメント設定用に、最近作成されたすべてのレプリケーションコントローラーの詳細を表示します。

--revision フラグを使用すると、リビジョン固有の詳細情報が表示されます。

```
$ oc rollout history dc/<name> --revision=1
```

デプロイメント設定および最新のリビジョンに関する詳細情報は、以下を実行してください。

```
$ oc describe dc <name>
```

注記

Web コンソール では、**Browse** タブにデプロイメントが表示されます。

9.2.3. デプロイメントのロールバック

ロールバックすると、アプリケーションを以前のリビジョンに戻します。この操作は、REST API、CLI または Web コンソールで実行できます。

最後にデプロイして成功した設定のリビジョンにロールバックするには、以下を実行します。

```
$ oc rollout undo dc/<name>
```

デプロイメント設定のテンプレートは、undo コマンドで指定してデプロイメントのリビジョンと一致するように元に戻され、新しいレプリケーションコントローラーが起動します。**--to-revision** でリビジョンが指定されていない場合には、最後に成功したデプロイメントのバージョンが使用されます。

ロールバックの完了直後に新規デプロイメントプロセスが誤って開始されないように、ロールバックの一部として、デプロイメント設定のイメージ変更トリガーは無効になります。イメージ変更トリガーを再度有効にするには、以下を実行します。

```
$ oc set triggers dc/<name> --auto
```

注記

最新のデプロイメントプロセスに失敗した場合に、デプロイメント設定は、最後に成功したリビジョンの設定に自動的にロールバックする機能をサポートします。この場合に、デプロイに失敗した最新のテンプレートはシステムで修正されないので、設定の修正はユーザーが行う必要があります。

9.2.4. コンテナー内でのコマンドの実行

コンテナーにコマンドを追加して、イメージの **ENTRYPOINT** を却下して、コンテナーの起動動作を変更することができます。これは、指定したタイミングでデプロイメントごとに 1 回実行できるライフサイクルフックとは異なります。

command パラメーターを、デプロイメントの **spec** フィールドを追加します。**command** コマンドを変更する **args** フィールドも追加できます (または **command** が存在しない場合には、**ENTRYPOINT**)。

```
...
spec:
  containers:
    -
    name: <container_name>
    image: 'image'
    command:
      - '<command>'
    args:
      - '<argument_1>'
      - '<argument_2>'
      - '<argument_3>'
...
```

たとえば、**-jar** および **/opt/app-root/springboots2idemo.jar** 引数を指定して、**java** コマンドを実行するには、以下を実行します。

```
...
spec:
  containers:
    -
    name: example-spring-boot
    image: 'image'
    command:
      - java
    args:
```

```
      - '-jar'
      - /opt/app-root/springboots2idemo.jar
...
```

9.2.5. デプロイメントログの表示

指定のデプロイメント設定に関する最新リビジョンのログをストリームします。

```
$ oc logs -f dc/<name>
```

最新のリビジョンが実行中または失敗した場合には、**oc logs** は、Pod のデプロイを行うプロセスのログが返されます。成功した場合には、**oc logs** は、アプリケーションの Pod からのログを返します。

以前に失敗したデプロイメントプロセスからのログを表示することも可能です。 ただし、これらのプロセス (以前のレプリケーションコントローラーおよびデプロイヤーの Pod) が存在し、手動でプルーニングまたは削除されていない場合に限ります。

```
$ oc logs --version=1 dc/<name>
```

ログの取得に関する他のオプションについては、以下を参照してください。

```
$ oc logs --help
```

9.2.6. デプロイメントトリガーの設定

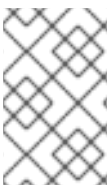
デプロイメント設定には、クラスター内のイベントに対応する新規デプロイメントプロセスの作成を駆動するトリガーを含めることができます。

警告

トリガーがデプロイメント設定に定義されていない場合は、**ConfigChange** トリガーがデフォルトで追加されます。トリガーが空のフィールドとして定義されている場合には、デプロイメントは手動で起動する必要があります。

9.2.6.1. 設定変更トリガー

ConfigChange トリガーにより、デプロイメント設定の Pod テンプレートに変更があると検出されるたびに、新規のレプリケーションコントローラーが作成されます。



注記

ConfigChange トリガーがデプロイメント設定に定義されている場合は、デプロイメント設定自体が作成された直後に、最初のレプリケーションコントローラーは自動的に作成され、一時停止されません。

例9.1 ConfigChange Trigger

```
triggers:
  - type: "ConfigChange"
```

9.2.6.2. ImageChange Trigger

ImageChange トリガーにより、イメージストリームタグの内容が変更されるたびに、（イメージ の新規バージョンがプッシュされるタイミングで）新規レプリケーションコントローラーが作成されます。

例9.2 ImageChange トリガー

```
triggers:
  - type: "ImageChange"
    imageChangeParams:
      automatic: true (1)
      from:
        kind: "ImageStreamTag"
        name: "origin-ruby-sample:latest"
        namespace: "myproject"
      containerNames:
        - "helloworld"
```

(1) **imageChangeParams.automatic** フィールドが **false** に設定されると、トリガーが無効になります。

上記の例では、**origin-ruby-sample** イメージストリームの **latest** タグの値が変更され、新しいイメージの値がデプロイメント設定の **helloworld** コンテナーに指定されている現在のイメージと異なる場合に、**helloworld** コンテナーの新規イメージを使用して、新しいレプリケーションコントローラーが作成されます。

注記

ImageChange トリガーがデプロイメント設定 (**ConfigChange** トリガーと **automatic=false**、または **automatic=true**) で定義されていて、**ImageChange** トリガーで参照されている **ImageStreamTag** がまだ存在していない場合には、ビルドにより、イメージが、**ImageStreamTag** にインポートまたはプッシュされた直後に初回のデプロイメントプロセスが自動的に開始されます。

9.2.6.2.1. コマンドラインの使用するには、以下を行います。

oc set triggers コマンドは、デプロイメント設定のデプロイメントトリガーを設定するために使用できます。上記の例では、次のコマンドを使用して **ImageChangeTrigger** を設定できます。

```
$ oc set triggers dc/frontend --from-image=myproject/origin-ruby-sample:latest -c helloworld
```

詳細は以下を参照してください。

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

9.2.7. デプロイメントリソースの設定

デプロイメントは、ノードでリソース (メモリーおよび CPU) を消費する Pod を使用して完了します。デフォルトで、Pod はバインドされていないノードのリソースを消費します。ただし、プロジェクトにデフォルトのコンテナー制限が指定されている場合には、Pod はその上限までリソースを消費します。

デプロイメントストラテジーの一部としてリソース制限を指定して、リソースの使用を制限することも可能です。デプロイメントリソースは、Recreate (再作成)、Rolling (ローリング) または Custom (カスタム) のデプロイメントストラテジーで使用できます。

以下の例では、**recources**、**cpu**、および **memory** はそれぞれ任意です。

```
type: "Recreate"
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:

```
type: "Recreate"
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。

コンピュートリソースや、要求と制限の相違点についての詳しい情報は、「クォータと制限の範囲」を参照してください。

- プロジェクトで定義される 制限の範囲。**LimitRange** オブジェクトのデフォルト値がデプロイメントプロセス時に作成される Pod に適用されます。

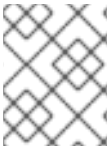
適用されない場合は、クォータ基準を満たさないために失敗したというメッセージが出され、デプロイメントの Pod 作成は失敗します。

9.2.8. 手動のスケーリング

ロールバック以外に、Web コンソールまたは **oc scale** コマンドを使用して、レプリカの数を細かく管理できます。たとえば、以下のコマンドは、デプロイメント設定の **frontend** を 3 に設定します。

```
$ oc scale dc frontend --replicas=3
```

レプリカの数は最終的に、デプロイメント設定の **frontend** で設定した希望のデプロイメントの状態と現在のデプロイメントの状態に伝搬されます。



注記

Pod は **oc autoscale** コマンドを使用して自動スケーリングすることも可能です。詳細は「Pod の自動スケーリング」を参照してください。

9.2.9. 特定のノードへの Pod の割り当て

ラベル付きのノードと合わせてノードセレクターを使用し、Pod の割り当てを制御することができます。

注記

OpenShift Container Platform 管理者は通常 インストール(Advanced installation)時にラベル を割り当てるか、または インストール後にノードに追加 できます。

クラスター管理者 は、プロジェクトに対して デフォルトのノードセレクターを設定 して 特定のノードに Pod の配置を制限できます。OpenShift Container Platform の開発者は、Pod 設定にノードセレクターを設定して、ノードをさらに制限することができます。

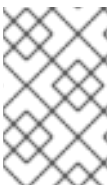
Pod の作成時にセレクターを追加するには、Pod 設定を編集し、**nodeSelector** の値を追加します。これは、単一の Pod 設定や、Pod テンプレートに追加できます。

```
apiVersion: v1
kind: Pod
spec:
  nodeSelector:
    disktype: ssd
...
```

ノードセレクターが有効な場合に作成される Pod は指定されたラベルを持つノードに割り当てられます。

ここで指定したラベルは、クラスター管理者が追加したラベルと併用さ れます。

たとえば、プロジェクトに **type=user-node** と **region=east** のラベルがクラスター管理者により追加され、上記の **disktype: ssd** ラベルを Pod に追加した場合に、Pod は 3 つのラベルすべてが含まれるノードにのみスケジュールされます。



注記

ラベルには値を 1 つしか設定できないので、**region=east** が管理者によりデフォルト設定されている Pod 設定に **region=west** のノードセレクターを設定すると、Pod が全くスケジュールされなくなります。

9.2.10. 異なるサービスアカウントでの Pod の実行

デフォルト以外のサービスアカウントで Pod を実行できます。

1. デプロイメント設定を編集します。

```
$ oc edit dc/<deployment_config>
```

2. **serviceAccount** と **serviceAccountName** パラメーターを **spec** フィールドに追加し、使用するサービスアカウントを指定します。

```
spec:
  securityContext: {}
  serviceAccount: <service_account>
  serviceAccountName: <service_account>
```

9.2.11. Web コンソールを使用してデプロイメント設定にシークレットを追加する手順

プライベートリポジトリーにアクセスできるように、デプロイメント設定にシークレットを追加します。

1. 新規の OpenShift Container Platform プロジェクトを作成します。
2. プライベートのイメージ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. デプロイメント設定を作成します。
4. デプロイメント設定のエディターページまたは、Web コンソール の **fromimage** ページで、**Pull Secret** を設定します。
5. **Save** ボタンをクリックします。

9.3. デプロイメントストラテジー

9.3.1. デプロイメントストラテジーの概要

デプロイメントストラテジーは、アプリケーションを変更またはアップグレードする 1 つの方法です。この目的は、ユーザーには改善が加えられていることが分からないように、ダウンタイムなしに変更を加えることにあります。

最も一般的なストラテジーとして blue-green デプロイメント を使用します。新規バージョン (blue バージョン) を、テストと評価用に起動しつつ、安定版 (green バージョン) をユーザーが継続して使用します。準備が整ったら、blue バージョンに切り替えられます。問題が発生した場合には、green バージョンに戻すことができます。

一般的な別のストラテジーとして、A/B バージョンがいずれも、同時にアクティブな状態で、A バージョンを使用するユーザーも、B バージョンを使用するユーザーもいるという方法があります。これは、ユーザーインターフェースや他の機能の変更をテストして、ユーザーのフィードバックを取得するために使用できます。また、ユーザーに対する問題の影響が限られている場合に、実稼働のコンテキストで操作が正しく行われていることを検証するのに使用することもできます。

カナリアデプロイメントでは、新規バージョンをテストしますが、問題が検出されると、すぐに以前のバージョンにフォールバックされます。これは、上記のストラテジーどちらでも実行できます。

ルートベースのデプロイメントストラテジーでは、サービス内の Pod 数はスケーリングされません。希望するパフォーマンスの特徴を維持するには、デプロイメント設定をスケーリングする必要があります。

デプロイメントストラテジーを選択する場合に、考慮するべき事項があります。

- 長期間実行される接続は正しく処理される必要があります。
- データベースの変換は複雑になる可能性があり、アプリケーションと共に変換し、ロールバックする必要があります。

- アプリケーションがマイクロサービスと従来のコンポーネントを使用するハイブリッドの場合には、移行の完了時にダウンタイムが必要になる場合があります。
- これを実行するためのインフラストラクチャーが必要です。
- テスト環境が分離されていない場合は、新規バージョンと以前のバージョン両方が破損してしまう可能性があります。

通常、エンドユーザーはルーターが取り扱うルート経由でアプリケーションにアクセスするので、デプロイメントストラテジーは、デプロイメント設定機能またはルーティング機能にフォーカスできます。

デプロイメント設定にフォーカスするストラテジーは、アプリケーションを使用するすべてのルートに影響を与えます。ルーター機能を使用するストラテジーは個別のルートにターゲットを設定します。

デプロイメントストラテジーの多くは、デプロイメント設定でサポートされ、追加のストラテジーはルーター機能でサポートされます。このセクションでは、デプロイメント設定をベースにするストラテジーについて説明します。

- ローリングストラテジー およびカナリアデプロイメント
- 再作成ストラテジー
- カスタムストラテジー
- ルートを使用した Blue-Green デプロイメント
- ルートを使用した A/B デプロイメント およびカナリアデプロイメント
- 1 サービス、複数のデプロイメント設定

ローリングストラテジー は、ストラテジーがデプロイメント設定に指定されていない場合にデフォルトで使用するストラテジーです。

デプロイメントストラテジーは、readiness チェック を使用して、新しい Pod の使用準備ができているかを判断します。Readiness チェックに失敗すると、デプロイメント設定は、タイムアウトするまで Pod の実行を再試行します。デフォルトのタイムアウトは、**10m** で、値は **dc.spec.strategy.*params** の **TimeoutSeconds** で設定します。

9.3.2. ローリングストラテジー

ローリングデプロイメントは、以前のバージョンのアプリケーションインスタンスを、新しいバージョンのアプリケーションインスタンスに徐々に置き換えます。ローリングデプロイメントは通常、新規 Pod が **readiness チェック** によって **ready** になるのを待機してから、古いコンポーネントをスケールダウンします。大きな問題が発生した場合には、ローリングデプロイメントは中断される可能性があります。

9.3.2.1. カナリアデプロイメント

OpenShift Container Platform におけるすべてのローリングデプロイメントは **カナリアデプロイメント** です。新規バージョン (カナリア) はすべての古いインスタンスが置き換えられる前にテストされます。Readiness チェックに成功しない場合には、カナリアリリースのインスタンスが削除され、デプロイメント設定は自動的にロールバックされます。Readiness チェックはアプリケーションコードの一部で、新規インスタンスの使用準備が確実に整うように、必要に応じて改善されます。より複雑なアプリケーションチェックを実装する必要がある場合には (新規インスタンスに実際のユーザーワークロードを送信するなど)、カスタムデプロイメントの実装や、blue-green デプロイメントストラテジーの使用を検討してください。

9.3.2.2. ローリングデプロイメントの使用のタイミング

- ダウンタイムを発生させずに、アプリケーションの更新を行う場合
- 以前のコードと新しいコードの同時実行がアプリケーションでサポートされている場合

ローリングデプロイメントとは、以前のバージョンと新しいバージョンのコードを同時に実行するという意味です。これは通常、アプリケーションで N-1 互換性に対応する必要があります。

以下は、ローリングストラテジーの例です。

```
strategy:
  type: Rolling
  rollingParams:
    updatePeriodSeconds: 1 (1)
    intervalSeconds: 1 (2)
    timeoutSeconds: 120 (3)
    maxSurge: "20%" (4)
    maxUnavailable: "10%" (5)
    pre: {} (6)
    post: {}
```

(1) 各 Pod が次に更新されるまで待機する時間。指定されていない場合、デフォルト値は **1** となります。

(2) 更新してからデプロイメントステータスをポーリングするまでの間待機する時間。指定されていない場合、デフォルト値は **1** となります。

(3) イベントのスケーリングを中断するまでの待機時間。この値はオプションです。デフォルトは **600** です。ここでの **中断** とは、自動的に以前の完全なデプロイメントにロールバックされるという意味です。

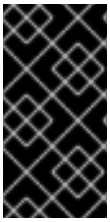
(4) **maxSurge** はオプションで、指定されていない場合には、デフォルト値は **25%** となります。以下の手順の次にある情報を参照してください。

(5) **maxUnavailable** はオプションで、指定されていない場合には、デフォルト値は **25%** となります。以下の手順の次にある情報を参照してください。

(6) **pre** および **post** はどちらもライフサイクルフックです。

ローリングストラテジーは以下を行います。

1. **pre** ライフサイクルフックを実行します。
2. サージ数に基づいて新しいレプリケーションコントローラーをスケールアップします。
3. 最大利用不可数に基づいて以前のレプリケーションコントローラーをスケールダウンします。
4. 新しいレプリケーションコントローラーが希望のレプリカ数に到達して、以前のレプリケーションコントローラーの数がゼロになるまで、このスケーリングを繰り返します。
5. **post** ライフサイクルフックを実行します。



重要

スケールダウン時には、ローリングストラテジーは Pod の準備ができるまで待機し、スケーリングを行うことで可用性に影響が出るかどうかを判断します。Pod をスケールアップしたにもかかわらず、準備が整わない場合には、デプロイメントプロセスは最終的にタイムアウトして、デプロイメントに失敗します。

maxUnavailable パラメーターは、更新時に利用できない Pod の最大数です。**maxSurge** パラメーターは、元の Pod 数を超えてスケジュールできる Pod の最大数です。どちらのパラメーターも、パーセント (例: **10%**) または絶対値 (例: **2**) のいずれかに設定できます。両方のデフォルト値は **25%** です。

以下のパラメーターを使用して、デプロイメントの可用性やスピードを調整できます。以下は例になります。

- **maxUnavailable=0** および **maxSurge=20%** が指定されていると、更新時および急速なスケールアップ時に完全なキャパシティーが維持されるようになります。
- **maxUnavailable=10%** および **maxSurge=0** が指定されていると、追加のキャパシティーを使用せずに更新を実行します (インプレース更新)。
- **maxUnavailable=10%** および **maxSurge=10%** の場合は、キャパシティーが失われる可能性がありますが、迅速にスケールアップおよびスケールダウンします。

一般的に、迅速にロールアウトする場合は **maxSurge** を使用します。リソースのクォータを考慮して、一部に利用不可の状態が発生してもかまわない場合には、**maxUnavailable** を使用します。

9.3.2.3. ローリングの例

OpenShift Container Platform では、ローリングデプロイメントはデフォルト設定です。ローリングアップデートを行うには、以下の手順に従います。

1. DockerHub にあるデプロイメントイメージの例を基にアプリケーションを作成します。

```
$ oc new-app openshift/deployment-example
```

ルーターをインストールしている場合は、ルートを使用してアプリケーションを利用できるようにしてください (または、サービス IP を直接使用してください)。

```
$ oc expose svc/deployment-example
```

deployment-example.<project>.<router_domain> でアプリケーションを参照し、**v1** イメージが表示されることを確認します。

2. レプリカが最大 3 つになるまで、デプロイメント設定をスケーリングします。

```
$ oc scale dc/deployment-example --replicas=3
```

3. 新しいバージョンの例を **latest** とタグ付けして、新規デプロイメントを自動的にトリガーします。

```
$ oc tag deployment-example:v2 deployment-example:latest
```

4. ブラウザーで、**v2** イメージが表示されるまでページを更新します。

5. CLI を使用している場合は、以下のコマンドで、バージョン 1 に Pod がいくつあるか、バージョン 2 にはいくつあるかを表示します。Web コンソールでは、徐々に v2 に追加される Pod、v1 から削除される Pod が確認できるはずです。

```
$ oc describe dc deployment-example
```

デプロイメントプロセスで、新しいレプリケーションコントローラーが漸増的にスケールアップします。新しい Pod が (readiness チェックをパスして) **ready** とマークされたら、デプロイメントプロセスが継続されます。Pod の準備が整わない場合には、プロセスが中断され、デプロイメント設定が以前のバージョンにロールバックされます。

9.3.3. 再作成ストラテジー

再作成ストラテジーは、基本的なロールアウト動作で、デプロイメントプロセスにコードを挿入するためのライフサイクルフックをサポートします。

以下は、再作成ストラテジーの例です。

```
strategy:
  type: Recreate
  recreateParams: 1
    pre: {} 2
    mid: {}
    post: {}
```

1 **recreateParams** はオプションです。

2 **pre**、**mid**、および **post** はライフサイクルフックです。

再作成ストラテジーは以下を行います。

1. **pre** ライフサイクルフックを実行します。
2. 以前のデプロイメントをゼロにスケールダウンします。
3. **mid** ライフサイクルフックを実行します。
4. 新規デプロイメントをスケールアップします。
5. **post** ライフサイクルフックを実行します。

重要

スケールアップ中に、デプロイメントのレプリカ数が複数ある場合は、デプロイメントの最初のレプリカが準備できているかどうかが検証されてから、デプロイメントが完全にスケールアップされます。最初のレプリカの検証に失敗した場合には、デプロイメントは失敗とみなされます。

9.3.3.1. 再作成デプロイメントの使用のタイミング

- 新規コードを起動する前に、移行または他のデータの変換を行う必要がある場合
- 以前のバージョンと新しいバージョンのアプリケーションコードの同時使用をサポートしていない場合

- 複数のレプリカ間での共有がサポートされていない、RWO ボリュームを使用する場合

再作成デプロイメントでは、短い期間にアプリケーションのインスタンスが実行されなくなるので、ダウンタイムが発生します。ただし、以前のコードと新しいコードは同時には実行されません。

9.3.4. カスタムストラテジー

カスタムストラテジーでは、独自のデプロイメントの動作を提供できるようになります。

以下は、カスタムストラテジーの例です。

```
strategy:
  type: Custom
  customParams:
    image: organization/strategy
    command: [ "command", "arg1" ]
    environment:
      - name: ENV_1
        value: VALUE_1
```

上記の例では、**organization/strategy** コンテナーイメージにより、デプロイメントの動作が提供されます。オプションの **command** 配列は、イメージの **Dockerfile** で指定した **CMD** ディレクティブを上書きします。指定したオプションの環境変数は、ストラテジープロセスの実行環境に追加されます。

さらに、OpenShift Container Platform は以下の環境変数もデプロイメントプロセスに提供します。

環境変数	説明
OPENSHIFT_DEPLOYMENT_NAME	新規デプロイメント名 (レプリケーションコントローラー)
OPENSHIFT_DEPLOYMENT_NAMESPACE	新規デプロイメントの namespace

新規デプロイメントのレプリカ数は最初はゼロです。ストラテジーの目的は、ユーザーのニーズに最適な仕方で対応するロジックを使用して新規デプロイメントをアクティブにすることにあります。

詳細は、高度なデプロイメントストラテジーを参照してください。

または **customParams** を使用して、カスタムのデプロイメントロジックを、既存のデプロイメントストラテジーに挿入します。カスタムのシェルロジックを指定して、**openshift-deploy** バイナリーを呼び出します。カスタムのデプロイヤーコンテナーイメージを用意する必要はありません。 ここでは、代わりにデフォルトの OpenShift Container Platform デプロイヤーイメージが使用されます。

```
strategy:
  type: Rolling
  customParams:
    command:
    - /bin/sh
    - -c
    - |
      set -e
      openshift-deploy --until=50%
```

```
    echo Halfway there
    openshift-deploy
    echo Complete
```

このストラテジーの設定では、以下のようなデプロイメントになります。

```
Started deployment #2
--> Scaling up custom-deployment-2 from 0 to 2, scaling down custom-deployment-1 from 2 to 0
(keep 2 pods available, don't exceed 3 pods)
    Scaling custom-deployment-2 up to 1
--> Reached 50% (currently 50%)
Halfway there
--> Scaling up custom-deployment-2 from 1 to 2, scaling down custom-deployment-1 from 2 to 0
(keep 2 pods available, don't exceed 3 pods)
    Scaling custom-deployment-1 down to 1
    Scaling custom-deployment-2 up to 2
    Scaling custom-deployment-1 down to 0
--> Success
Complete
```

カスタムデプロイメントストラテジーのプロセスでは、OpenShift Container Platform API または Kubernetes API へのアクセスが必要な場合には、ストラテジーを実行するコンテナーは、認証用のコンテナーで利用可能なサービスアカウントのトークンを使用できます。

9.3.5. ライフサイクルフック

再作成 および ローリング ストラテジーは、ストラテジーで事前に定義したポイントでデプロイメントプロセスに動作を挿入できるようにするライフサイクルフックをサポートします。

以下は **pre** ライフサイクルフックの例です。

```
pre:
  failurePolicy: Abort
  execNewPod: {} 1
```

1 **execNewPod** は Pod ベースのライフサイクルフック です。

フックにはすべて、フックに問題が発生した場合にストラテジーが取るべきアクションを定義する **failurePolicy** が含まれます。

Abort	フックに失敗すると、デプロイメントプロセスも失敗とみなされます。
Retry	フックの実行は、成功するまで再試行されます。
Ignore	フックの失敗は無視され、デプロイメントは続行されます。

フックには、フックの実行方法を記述するタイプ固有のフィールドがあります。現在、フックタイプとしてサポートされているのは Pod ベースのフック のみで、このフックは e **execNewPod** フィールドで指定されます。

9.3.5.1. Pod ベースのライフサイクルフック

Pod ベースのライフサイクルフックは、デプロイメント設定のテンプレートをベースとする新しい Pod でフックコードを実行します。

以下のデプロイメント設定例は簡素化されており、この例では ローリングストラテジーを使用します。簡潔にまとめられるように、トリガーおよびその他の詳細は省略しています。

```
kind: DeploymentConfig
apiVersion: v1
metadata:
  name: frontend
spec:
  template:
    metadata:
      labels:
        name: frontend
    spec:
      containers:
        - name: helloworld
          image: openshift/origin-ruby-sample
  replicas: 5
  selector:
    name: frontend
  strategy:
    type: Rolling
    rollingParams:
      pre:
        failurePolicy: Abort
        execNewPod:
          containerName: helloworld (1)
          command: [ "/usr/bin/command", "arg1", "arg2" ] (2)
          env: (3)
            - name: CUSTOM_VAR1
              value: custom_value1
          volumes:
            - data (4)
```

(1) **helloworld** の名前は **spec.template.spec.containers[0].name** を参照します。

(2) この **command** は、**openshift/origin-ruby-sample** イメージで定義される **ENTRYPOINT** を上書きします。

(3) **env** は、フックコンテナーの環境変数です (任意)。

(4) **volumes** は、フックコンテナーのボリューム参照です (任意)。

この例では、**pre** フックは、**helloworld** コンテナーからの **openshift/origin-ruby-sample** イメージを使用して新規 Pod で実行されます。フック Pod には以下のプロパティーが設定されます。

- フックコマンドは **/usr/bin/command arg1 arg2** となります。
- フックコンテナーには **CUSTOM_VAR1=custom_value1** 環境変数が含まれます。
- フックの失敗ポリシーは **Abort** で、フックが失敗するとデプロイメントプロセスも失敗します。

- フック Pod は、設定 Pod から **data** ボリュームを継承します。

9.3.5.2. コマンドラインの使用するには、以下を行います。

oc set deployment-hook コマンドは、デプロイメント構成にデプロイメントフックを設定するのに使用できます。上記の例では、以下のコマンドでプリデプロイメントフックを設定できます。

```
$ oc set deployment-hook dc/frontend --pre -c helloworld -e CUSTOM_VAR1=custom_value1 \
  -v data --failure-policy=abort -- /usr/bin/command arg1 arg2
```

9.4. 高度なデプロイメントストラテジー

9.4.1. 高度なデプロイメントストラテジー

デプロイメントストラテジーは、アプリケーションを進化させる手段として使用します。一部のストラテジーは デプロイメント設定 を使用して変更を加えます。これらの変更は、アプリケーションを解決する全ルートのユーザーに表示されます。これらの変更は、アプリケーションを解決する全ルートのユーザーに表示されます。 ここで説明している他のストラテジーは、ルート機能を使用して固有のルートに影響を与えます。

9.4.2. Blue-Green デプロイメント

Blue-green デプロイメントでは、同時に 2 つのバージョンを実行し、実稼働版 (green バージョン) からより新しいバージョン (blue バージョン) にトラフィックを移動します。ルートでは、ローリングストラテジーまたは切り替えサービスを使用できます。

注記

多くのアプリケーションは永続データに依存するので、N-1 互換性 をサポートするアプリケーションが必要です。 つまり、データを共有して、データ層を 2 つ作成し、データベース、ストアまたはディスク間のライブマイグレーションを実装します。

新規バージョンのテストに使用するデータについて考えてみてください。実稼働データの場合には、新規バージョンのバグにより、実稼働版を破損してしまう可能性があります。

9.4.2.1. Blue-Green デプロイメントの使用

Blue-Green デプロイメントは 2 つのデプロイメント設定を使用します。いずれも実行され、実稼働のデプロイメントはルートが指定するルートによって変わります。この際、各デプロイメント設定は異なるサービスに公開されます。準備ができたら、実稼働ルートのサービスが新規サービスを参照するように変更します。 新規 (blue) バージョンは有効になります。

必要に応じて以前のバージョンにサービスを切り替えて、以前の green バージョンにロールバックすることができます。

ルートと 2 つのサービスの使用
以下の例は、2 つのデプロイメント設定を行います。 1 つは、安定版 (green バージョン) で、もう 1 つは新規バージョン (blue バージョン) です。

ルートは、サービスを参照し、いつでも別のサービスを参照するように変更できます。開発者は、実稼働トラフィックが新規サービスにルーティングされる前に、新規サービスに接続して、コードの新規バージョンをテストできます。

ルートは、Web (HTTP および HTTPS) トラフィックを対象としているので、この手法は Web アプリケーションに最適です。

1. アプリケーションサンプルの 2 つのコピーを作成します。

```
$ oc new-app openshift/deployment-example:v1 --name=example-green
$ oc new-app openshift/deployment-example:v2 --name=example-blue
```

上記のコマンドにより、独立したアプリケーションコンポーネントが 2 つ作成されます。1 つは、**example-green** サービスで **v1** イメージを実行するコンポーネントと、もう 1 つは **example-blue** サービスで **v2** イメージを実行するコンポーネントです。

2. 以前のサービスを参照するルートを作成します。

```
$ oc expose svc/example-green --name=bluegreen-example
```

3. **bluegreen-example.<project>.<router_domain>** でアプリケーションを参照し、**v1** イメージが表示されることを確認します。

注記

v3.0.1 より前のバージョンの OpenShift Container Platform では、このコマンドは上記の場所ではなく、**example-green.<project>.<router_domain** > にルートを生成します。

4. ルートを編集して、サービス名を **example-blue** に変更します。

```
$ oc patch route/bluegreen-example -p '{"spec":{"to":{"name":"example-blue"}}}'
```

5. ルートが変更されたことを確認するには、**v2** イメージが表示されるまで、ブラウザーを更新します。

9.4.3. A/B デプロイメント

A/B デプロイメントストラテジーでは、新しいバージョンのアプリケーションを実稼働環境での制限された方法で試すことができます。実稼働バージョンは、ユーザーの要求の大半に対応し、要求の一部が新しいバージョンに移動されるように指定できます。各バージョンへの要求の部分を制御できるので、テストが進むにつれ、新しいバージョンへの要求を増やし、最終的に以前のバージョンの使用を停止することができます。各バージョン要求負荷を調整するにつれ、期待どおりのパフォーマンスを出せるように、各サービスの Pod 数もスケーリングする必要があります。

ソフトウェアのアップグレードに加え、この機能を使用してユーザーインターフェースのバージョンを検証することができます。以前のバージョンを使用するユーザーと、新しいバージョンを使用するユーザーが出てくるので、異なるバージョンに対するユーザーの反応を評価して、設計上の意思決定を知らせることができます。

このデプロイメントの効果を発揮するには、以前のバージョンと新しいバージョンは同時に実行できるほど類似している必要があります。これは、バグ修正リリースや新機能が以前の機能と干渉しないようにする場合の一般的なポイントになります。これらのバージョンが正しく連携するには N-1 互換性 が必要です。

OpenShift Container Platform は、Web コンソールとコマンドラインインターフェースで N-1 互換性をサポートします。

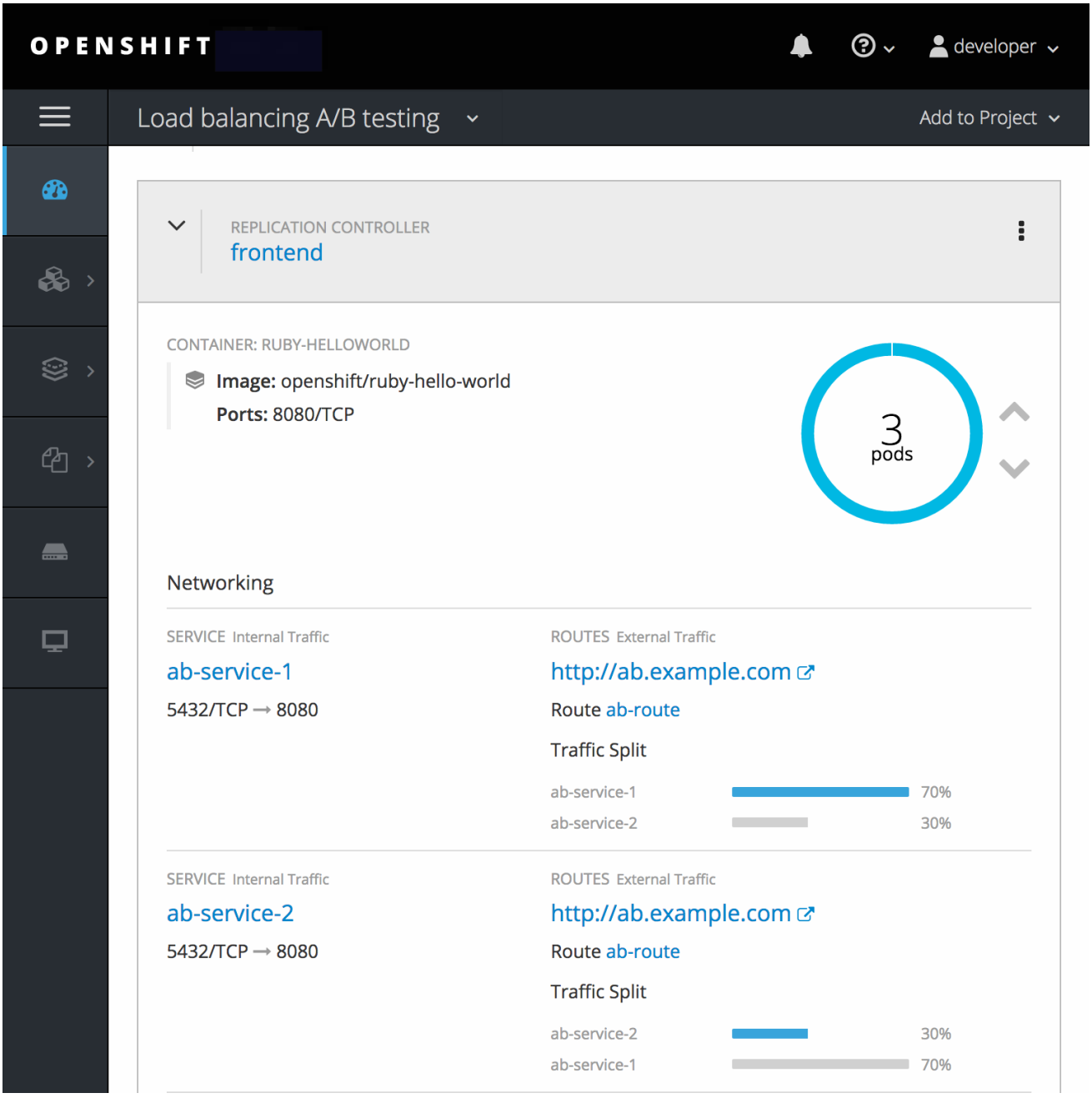
9.4.3.1. A/B テスト用の負荷分散

ユーザーは 複数のサービスでルート を設定します。各サービスは、アプリケーションの 1 つのバージョンを処理します。

各サービスには **weight** が割り当てられ、各サービスへの要求の部分については **service_weight** を **sum_of_weights** で除算します。エンドポイントの **weights** の合計がサービスの **weight** になるように、サービスごとの **weight** がサービスのエンドポイントに分散されます。

ルートにはサービスを最大で 4 つ含めることができます。サービスの **weight** は、**0** から **256** の間で指定してください。**weight** が **0** の場合、新しい要求はサービスには送信されませんが、既存の接続はアクティブのままになります。サービスの **weight** が **0** でない場合は、エンドポイントの最小 **weight** は **1** となります。これにより、エンドポイントが多数含まれるサービスは、最終的に **weight** は必要な値よりも大きくなる可能性があります。このような場合は、負荷分散の **weight** を必要なレベルに下げるために Pod の数を減らします。詳細は、Alternate Backends and Weights セクションを参照してください。

Web コンソールでは、重みを設定したり、各サービス間の重みの分散を表示したりできます。



A/B 環境を設定するには以下を行います。

1. 2 つのアプリケーションを作成して、異なる名前を指定します。それぞがデプロイメント設定を作成します。これらのアプリケーションは同じアプリケーションのバージョンであり、通常 1 つは現在の実稼働バージョンで、もう 1 つは提案される新規バージョンとなります。

```
$ oc new-app openshift/deployment-example1 --name=ab-example-a
$ oc new-app openshift/deployment-example2 --name=ab-example-b
```

2. デプロイメント設定を公開してサービスを作成します。

```
$ oc expose dc/ab-example-a --name=ab-example-A
$ oc expose dc/ab-example-b --name=ab-example-B
```

この時点で、いずれのアプリケーションもデプロイ、実行され、サービスが追加されています。

3. ルート経由でアプリケーションを外部から利用できるようにします。この時点でサービスを公開できます。 現在の実稼働バージョンを公開してから、後でルートを編集して新規バージョンを追加すると便利です。

```
$ oc expose svc/ab-example-A
```

ab-example.<project>.<router_domain> でアプリケーションを参照して、希望とするバージョンが表示されていることを確認します。

4. ルートをデプロイする場合には、ルーターはサービスに指定 した **weights** に従ってトラフィックを分散 します。この時点では、デフォルトの **weight=1** と指定されたサービスが 1 つ存在するので、すべての要求がこのサービスに送られます。他のサービスを **alternateBackends** として追加し、**weights** を調整すると、A/B 設定が機能するようになります。これは、 **oc set route-backends** コマンドを実行するか、ルートを編集して実行できます。

注記

ルートに変更を加えると、さまざまなサービスへのトラフィックの部分だけが変更されます。デプロイメント設定をスケーリングして、必要な負荷を処理できるように Pod 数を調整する必要がある場合があります。

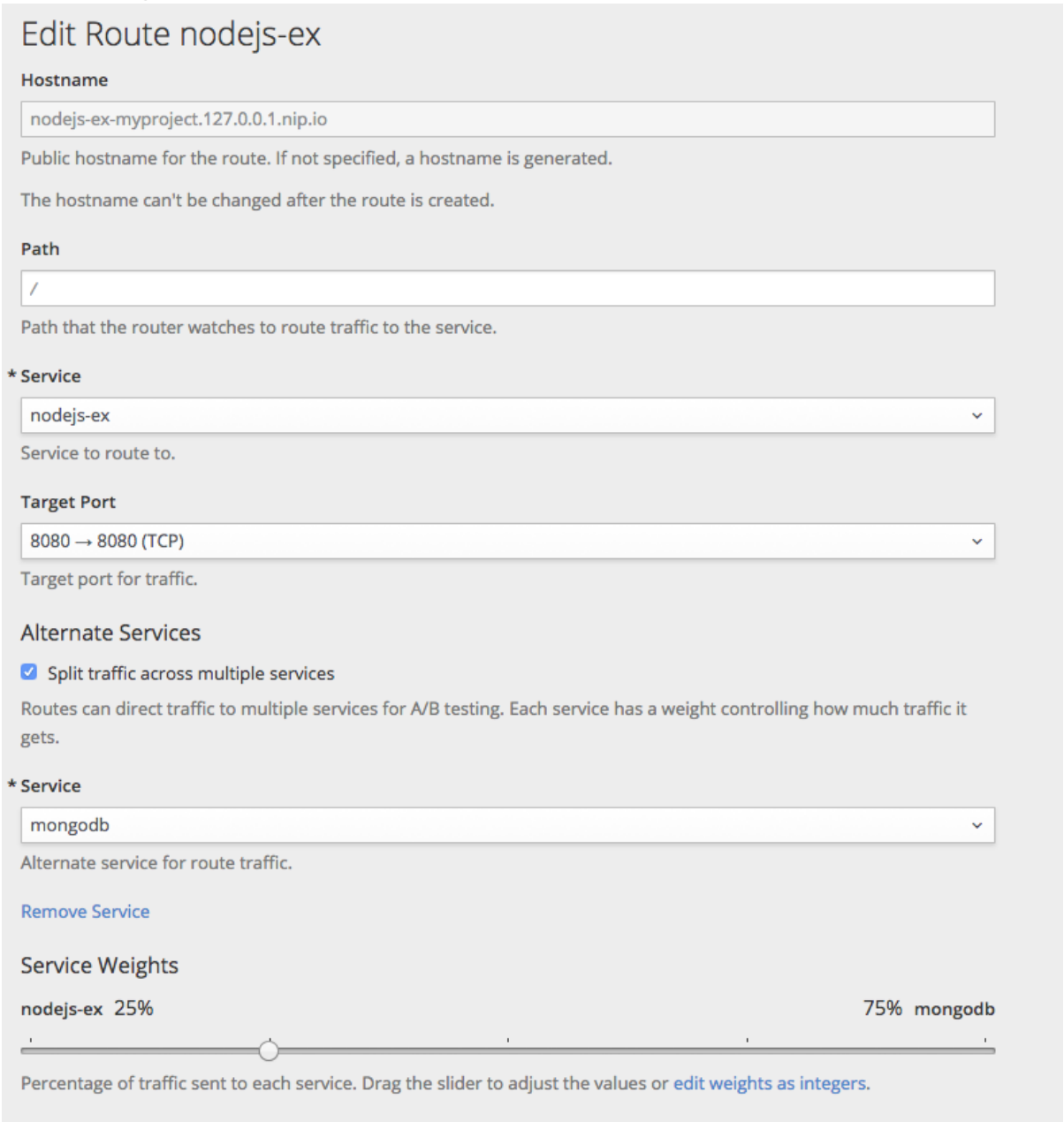
ルートを編集するには、以下を実行します。

```
$ oc edit route <route-name>
...
metadata:
  name: route-alternate-service
  annotations:
    haproxy.router.openshift.io/balance: roundrobin
spec:
  host: ab-example.my-project.my-domain
  to:
    kind: Service
    name: ab-example-A
    weight: 10
  alternateBackends:
  - kind: Service
```

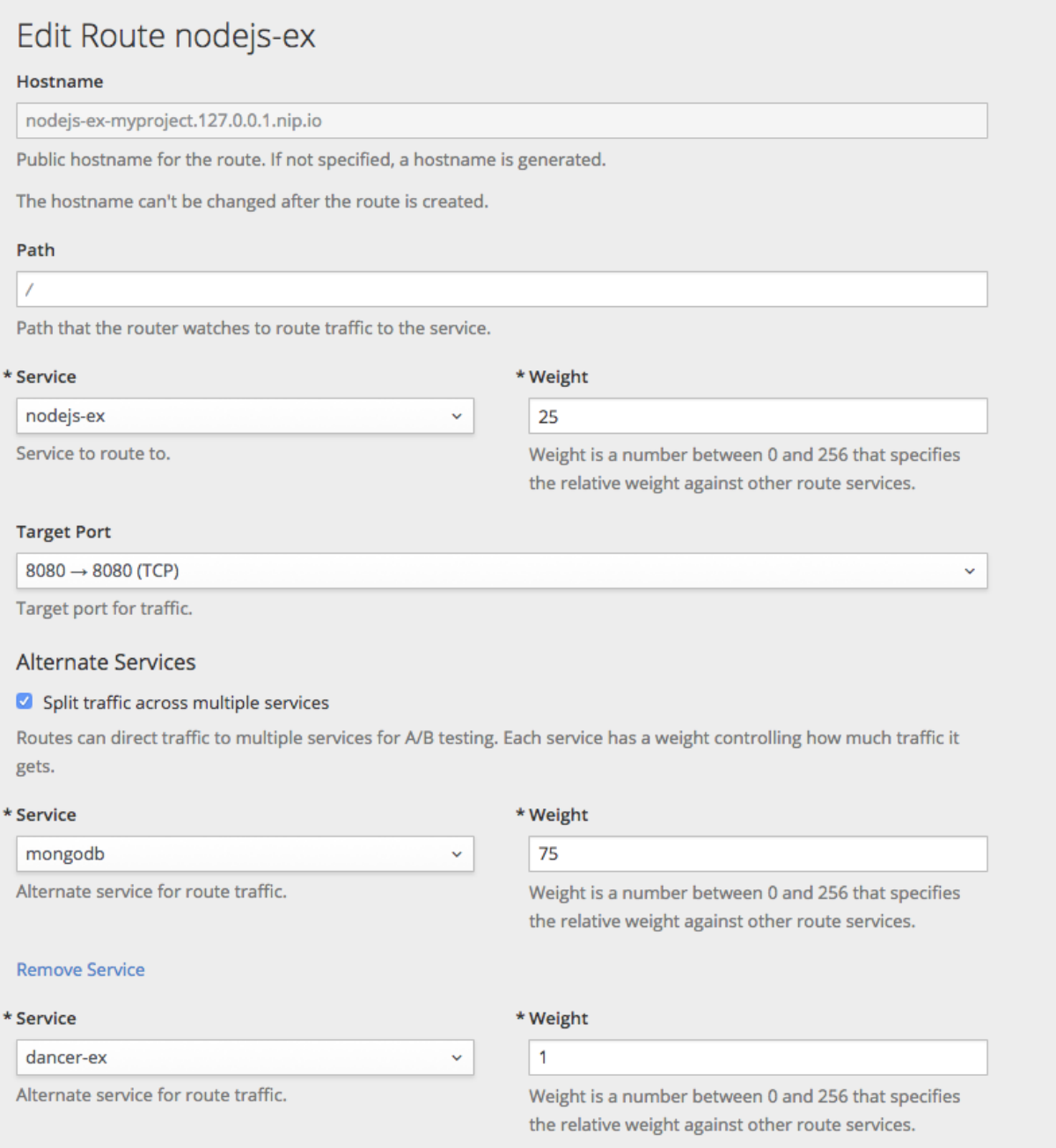
```
    name: ab-example-B
    weight: 15
...
```

9.4.3.1.1. Web コンソールを使用した重みの管理

1. Route の詳細ページ (アプリケーション/ルート) に移動します。
2. Actions メニューから **Edit** を選択します。
3. **Split traffic across multiple services**にチェックを入れます。
4. **Service Weights** スライダーで、各サービスに送信するトラフィックの割合を設定します。



2 つ以上のサービスにトラフィックを分割する場合には、各サービスに 0 から 256 の整数を使用して、相対的な重みを指定します。



トラフィックの重みは、トラフィックを分割したアプリケーションの行を展開すると **Overview** に表示されます。

9.4.3.1.2. CLI を使用した重みの管理

このコマンドは、ルートでサービスと対応する重みの 負荷分散 を管理します。

```
$ oc set route-backends ROUTENAME [--zero|--equal] [--adjust] SERVICE=WEIGHT[%] [...] [options]
```

たとえば、以下のコマンドは **ab-example-A** に **weight=198** を指定して主要なサービスとし、**ab-example-B** に **weight=2** を指定して 1 番目の代用サービスとして設定します。

```
$ oc set route-backends web ab-example-A=198 ab-example-B=2
```

つまり、99% のトラフィックはサービス **ab-example-A** に、1% はサービス **ab-example-B** に送信されます。

このコマンドでは、デプロイメント設定はスケーリングされません。要求の負荷を処理するのに十分な Pod がある状態でこれを実行する必要があります。

フラグなしのコマンドでは、現在の設定が表示されます。

```
$ oc set route-backends web
NAME                KIND     TO          WEIGHT
routes/web          Service  ab-example-A 198 (99%)
routes/web          Service  ab-example-B 2   (1%)
```

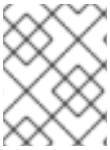
--adjust フラグを使用すると、個別のサービスの重みを、それ自体に対して、または主要なサービスに対して相対的に変更できます。割合を指定すると、主要サービスまたは 1 番目の代用サービス (主要サービスを設定している場合) に対して相対的にサービスを調整できます。他にバックエンドがある場合には、重みは変更に比例した状態になります。

```
$ oc set route-backends web --adjust ab-example-A=200 ab-example-B=10
$ oc set route-backends web --adjust ab-example-B=5%
$ oc set route-backends web --adjust ab-example-B=+15%
```

--equal フラグでは、全サービスの **weight** が 100 になるように設定します。

```
$ oc set route-backends web --equal
```

--zero フラグは、全サービスの **weight** を 0 に設定します。すべての要求に対して 503 エラーが返されます。



注記

ルートによっては、複数のバックエンドたは重みが設定されたバックエンドをサポートしないものがあります。

9.4.3.1.3.1 サービス、複数のデプロイメント設定

ルーターをインストールしている場合は、ルートを使用してアプリケーションを利用できるようにしてください (または、サービス IP を直接使用してください)。

```
$ oc expose svc/ab-example
```

ab-example.<project>.<router_domain> でアプリケーションを参照し、**v1** イメージが表示されることを確認します。

1. 1 つ目のシャードと同じだが別のバージョンがタグ付けされたソースイメージを基に 2 つ目のシャードを作成して、一意の値を設定します。

   ```
   $ oc new-app openshift/deployment-example:v2 --name=ab-example-b --labels=ab-example=true SUBTITLE="shard B" COLOR="red"
   ```

2. 新たに作成したシャードを編集して、全シャードに共通の **ab-example=true** ラベルを設定します。

   ```
   $ oc edit dc/ab-example-b
   ```

エディターで、**spec.selector** および **spec.template.metadata.labels** のトに、既存の **deploymentconfig=ab-example-b** ラベルと一緒に **ab-example: "true"** の行を追加します。保存してからエディターを終了します。

3. 2 番目のシャードの再デプロイメントをトリガーして、新規ラベルを取得します。

```
$ oc rollout latest dc/ab-example-b
```

4. この時点で、いずれの Pod のセットもルートで提供されます。しかし、両ブラウザー (接続を開放) とルーター (デフォルトでは cookie を使用) で、バックエンドサーバーへの接続を維持しようとするので、シャードが両方返されない可能性があります。1 つまたは他のシャードに対してブラウザーを強制的に実行するには、scale コマンドを使用します。

```
$ oc scale dc/ab-example-a --replicas=0
```

ブラウザーを更新すると、 **v2** および **shard B** (赤字) が表示されているはずです。

```
$ oc scale dc/ab-example-a --replicas=1; oc scale dc/ab-example-b --replicas=0
```

ブラウザーを更新すると、**v1** と **shard A** (青字) が表示されているはずです。

いずれかのシャードでデプロイメントをトリガーした場合には、そのシャード内の Pod のみが影響を受けます。いずれかのデプロイメント設定で **SUBTITLE** 環境変数を変更して (**oc edit dc/ab-example-a** または **oc edit dc/ab-example-b**)、デプロイメントを簡単にトリガーできます。ステップ 5-7 を繰り返すと、別のシャードを追加できます。

注記

これらの手順は、今後の OpenShift Container Platform バージョンでは簡素化される予定です。

9.4.4. プロキシーシャード/トラフィックスプリッター

実稼働環境で、特定のシャードに到達するトラフィックの分散を正確に制御できます。多くのインスタンスを扱う場合は、各シャードに相対的なスケールを使用して、割合ベースのトラフィックを実装できます。これは、他の場所で実行中の別のサービスやアプリケーションに転送または分割する **プロキシーシャード** とも適切に統合されます。

最も単純な設定では、プロキシーは要求を変更せずに転送します。より複雑な設定では、受信要求を複製して、別のクラスターだけでなく、アプリケーションのローカルインスタンスにも送信して、結果を比較することができます。他のパターンとしては、DR のインストールのキャッシュを保持したり、分析目的で受信トラフィックをサンプリングすることができます。

実装がこの例のスコープ外の場合でも、TCP (または UDP) のプロキシーは必要なシャードで実行できます。**oc scale** コマンドを使用して、プロキシーシャードで要求に対応するインスタンスの相対数を変更してください。より複雑なトラフィックを管理する場合には、OpenShift Container Platform ルーターを比例分散機能でカスタマイズすることを検討してください。

9.4.5. N-1 互換性

新規コードと以前のコードが同時に実行されるアプリケーションの場合は、新規コードで記述されたデータが、以前のコードで読み込みや処理 (または正常に無視) できるように注意する必要があります。これは、**スキーマの進化** と呼ばれ、複雑な問題です。

これは、ディスクに保存したデータ、データベース、一時的なキャッシュ、ユーザーのブラウザーセッ

ションの一部など、多数の形式を取ることができます。多くの Web アプリケーションはローリングデプロイメントをサポートできますが、アプリケーションをテストし、設計してこれに対応させることが重要です。

アプリケーションによっては、新旧のコードが並行的に実行されている期間が短いため、バグやユーザーのトランザクションに失敗しても許容範囲である場合があります。別のアプリケーションでは失敗したパターンが原因で、アプリケーション全体が機能しなくなる場合もあります。

N-1 互換性を検証する 1 つの方法として、A/B デプロイメントがあります。制御されたテスト環境で、以前のコードと新しいコードを同時に実行して、新規デプロイメントに流れるトラフィックが以前のデプロイメントで問題を発生させないかを確認します。

9.4.6. 正常な終了

OpenShift Container Platform および Kubernetes は、負荷分散のローテーションから削除する前にアプリケーションインスタンスがシャットダウンする時間を設定します。ただし、アプリケーションでは、終了前にユーザー接続が正常に中断されていることを確認する必要があります。

シャットダウン時に、OpenShift Container Platform はコンテナーのプロセスに **TERM** シグナルを送信します。**SIGTERM** を受信すると、アプリケーションコードは、新規接続の受け入れを停止する必要があります。こうすることで、ロードバランサーにより、他のアクティブなインスタンスにトラフィックをルーティングされるようになります。アプリケーションコードは、開放されている接続がすべて終了する (または、次の機会に個別接続が正常に終了される) まで待機してから終了します。

正常に終了する期間が終わると、終了されていないプロセスに **KILL** シグナルが送信され、プロセスが即座に終了されます。Pod の **terminationGracePeriodSeconds** 属性または Pod テンプレートが正常に終了する期間 (デフォルト 30 秒) を制御し、必要に応じてアプリケーションごとにカスタマイズすることができます。

9.5. KUBERNETES デプロイメントサポート

9.5.1. デプロイメントオブジェクトタイプ

Kubernetes には、OpenShift Container Platform では **デプロイメント** と呼ばれるファーストクラスのオブジェクトタイプがあります。このオブジェクトタイプ (ここでは区別するために **Kubernetes デプロイメント** と呼びます) は、デプロイメント設定オブジェクトタイプの派生タイプとして機能します。

デプロイメント設定と同様に、Kubernetes デプロイメントは Pod テンプレートとして、アプリケーションの特定のコンポーネントの必要とされる状態を記述します。Kubernetes デプロイメント **は レプリカセット** （ レプリケーションコントローラーの反復）を作成して、Pod ライフサイクルをオーケストレーションします。

たとえば、Kubernetes デプロイメントのこの定義は、レプリカセットを作成して **hello-openshift** Pod を 1 つ起動します。

例: Kubernetes デプロイメント定義 hello-openshift-deployment.yaml

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: hello-openshift
spec:
  replicas: 1
  selector:
    matchLabels:
```

```
    app: hello-openshift
template:
  metadata:
    labels:
      app: hello-openshift
  spec:
    containers:
    - name: hello-openshift
      image: openshift/hello-openshift:latest
      ports:
      - containerPort: 80
```

ローカルファイルに定義を保存した後に、Kubernetes デプロイメントの作成にこのファイルを使用できます。

```
$ oc create -f hello-openshift-deployment.yaml
```

CLI を使用して、**get** や **describe** などの 一般的な操作で説明されているように、他のオブジェクトタイプなどの Kubernetes デプロイメントおよびレプリカセットを検証し、操作 できます。オブジェクトタイプの場合、Kubernetes デプロイメントには **deployments** または **deploy** を、レプリカセットには **replicasets** または **rs** を使用します。

デプロイメント および レプリカセット に関する詳細は、Kubernetes のドキュメントを参照してください。 CLI の使用方法の例で、**oc** を **kubectl** に置き換えてください。

9.5.2. Kubernetes デプロイメント 対 デプロイメント設定

デプロイメントが Kurbernetes 1.2 に追加される前にデプロイメント設定が OpenShift Container Platform に存在していたために、Kurbenetes のオブジェクトタイプは OpenShift Container Platform の設定とは若干異なっています。OpenShift Container Platform の長期的な目標は、Kubernetes デプロイメントと全く同等な機能を実現し、アプリケーションの詳細な管理を可能にする単一オブジェクトタイプとしてそれらのデプロイメントを使用する方法に切り替えることにあります。

新規オブジェクトタイプを使用するアップストリームのプロジェクトや例が OpenShift Container Platform でスムーズに実行できるように、Kubernetes デプロイメントはサポートされます。Kubernetes デプロイメントの現在の機能を考慮すると、特に以下のいずれかを使用する予定がない場合には、OpenShift Container Platform デプロイメント設定の代わりに、Kubernetes デプロイメントを使用すると良いでしょう。

- イメージストリーム
- ライフサイクルフック
- カスタムデプロイメントストラテジー

以下のセクションでは、2 つのオブジェクトタイプの相違点に関してさらに扱います。 これは、デプロイメント設定ではなく、Kubernetes デプロイメントを使用する場合を判別するのに役立ちます。

9.5.2.1. デプロイメント設定固有の機能

9.5.2.1.1. 自動ロールバック

Kubernetes デプロイメントは、問題が発生した場合に、最後に正常にデプロイされたレプリカセットに自動的にロールバックされません。この機能は近日追加される予定です。

9.5.2.1.2. トリガー

Kubernetes デプロイメントには、デプロイメントの Pod テンプレートに変更があるたびに、新しいロールアウトが自動的にトリガーされるので、暗黙的な **ConfigChange** トリガーが含まれています。Pod テンプレートの変更時に新たなロールアウトが不要な場合には、デプロイメントを以下のように停止します。

```
$ oc rollout pause deployments/<name>
```

現在、Kubernetes デプロイメントでは **ImageChange** トリガーはサポートされません。汎用的なトリガーの仕組みがアップストリームでは提案されていますが、この提案が受け入れられるのか、また受け入れられるタイミングは不明です。最終的には、OpenShift Container Platform 固有の仕組みが、Kubernetes デプロイメントの階層の上に実装される可能性がありますが、Kubernetes コアの一部として存在させる方が適しています。

9.5.2.1.3. ライフサイクルフック

Kubernetes デプロイメントではライフサイクルフックがサポートされません。

9.5.2.1.4. カスタムストラテジー

Kubernetes デプロイメントでは、ユーザーが指定するカスタムデプロイメントストラテジーはまだサポートされていません。

9.5.2.1.5. カナリアデプロイメント

Kubernetes デプロイメントでは、新規ロールアウトの一部としてカナリアリリースは実行されません。

9.5.2.1.6. テストデプロイメント

Kubernetes デプロイメントでは、実行中のテストトラックはサポートされません。

9.5.2.2. Kubernetes デプロイメント固有の機能

9.5.2.2.1. ロールオーバー

Kubernetes デプロイメントのデプロイメントプロセスは、コントローラーループで駆動されますが、デプロイメント設定は、新しいロールアウトごとにデプロイヤー Pod を使用します。つまり、Kubernetes デプロイメントはできるだけアクティブなレプリカセットを指定することができ、最終的にデプロイメントコントローラーが以前のレプリカセットをスケールダウンし、最新のものをスケールアップします。

デプロイメント設定で、実行できるデプロイヤー Pod は最大 1 つとなっています。デプロイヤーが 2 つある場合は、他のデプロイヤーと競合して、それぞれが最新のレプリケーションコントローラーであると考えるコントローラーをスケールアップしようとします。これにより、一度にアクティブにできるのは、レプリケーションコントローラー 2 つだけで、最終的に、Kubernetes デプロイヤーのロールアウトが加速します。

9.5.2.2.2. 比例スケーリング

Kubernetes デプロイメントコントローラーのみがデプロイメントが所有する新旧レプリカセットのサイズについての信頼できる情報源であるため、継続中のロールアウトのスケーリングできます。追加のレプリカはレプリカセットのサイズに比例して分散されます。

デプロイメント設定は、デプロイメント設定コントローラーが新規レプリケーションコントローラーのサイズに関してデプロイヤープロセスと競合するためにロールアウトが続行されている場合はスケーリングできません。

9.5.2.2.3. ロールアウト中の一時停止

Kubernetes デプロイメントはいつでも一時停止できます。つまり、継続中のロールアウトも一時停止できます。 反対に、デプロイヤー Pod は現時点で一時停止できないので、ロールアウト時にデプロイメント設定を一時停止しようとしても、デプロイヤープロセスはこの影響を受けず、完了するまで続行されます。

第10章 TEMPLATES (テンプレート)

10.1. 概要

テンプレートでは、パラメーター化や処理が可能な一連の オブジェクト を記述し、OpenShift Container Platform で作成するためのオブジェクトの一覧を生成します。テンプレートは、サービス、ビルド設定および デプロイメント設定 など、プロジェクト内で作成パーミッションがあるすべてのものを作成するために処理できます。また、テンプレートでは ラベル のセットを定義して、これをテンプレート内に定義されたすべてのオブジェクトに適用できます。

オブジェクトの一覧は CLI を使用してテンプレートから作成することも、プロジェクトまたはグローバルテンプレートライブラリーにテンプレートがアップロードされている場合、Web コンソールを使用することもできます。キュレートされたテンプレートの場合は、OpenShift イメージストリームおよびテンプレートライブラリー を参照してください。

10.2. テンプレートのアップロード

テンプレートを定義する JSON または YAML ファイルがある場合は、この例にあるように、CLI を使用してプロジェクトにテンプレートをアップロードできます。こうすることで、プロジェクトにテンプレートが保存され、対象のプロジェクトに対して適切なアクセスを持つユーザーが繰り返し使用できます。独自のテンプレートの記述については、このトピックで後ほど説明します。

現在のプロジェクトのテンプレートライブラリーにテンプレートをアップロードするには、JSON または YAML ファイルを以下のコマンドで渡します。

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

-n オプションを使用してプロジェクト名を指定することで、別のプロジェクトにテンプレートをアップロードできます。

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

テンプレートは、Web コンソールまたは CLI を使用して選択できるようになりました。

10.3. WEB コンソールを使用してテンプレートから作成する手順

「Web コンソール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。

10.4. CLI を使用してテンプレートから作成する手順

CLI を使用して、テンプレートを処理し、オブジェクトを作成するために生成された設定を使用できます。

10.4.1. ラベル

ラベル は、Pod などの生成されたオブジェクトを管理し、整理するために使用されます。テンプレートで指定されるラベルは、テンプレートから生成されるすべてのオブジェクトに適用されます。

コマンドラインからテンプレートにラベルを追加する機能もあります。

```
$ oc process -f <filename> -l name=otherLabel
```

10.4.2. パラメーター

上書きできるパラメーターの一覧は、テンプレートの **parameters** セクションに表示されます。以下のコマンドで使用するファイルを指定して、CLI でパラメーター一覧を追加できます。

```
$ oc process --parameters -f <filename>
```

または、テンプレートがすでにアップロードされている場合には、以下を実行します。

```
$ oc process --parameters -n <project> <template_name>
```

たとえば、デフォルトの **openshift** プロジェクトにあるクイックスタートテンプレートのいずれかに対してパラメーターを一覧表示する場合に、以下のような出力が表示されます。

```
$ oc process --parameters -n openshift rails-postgresql-example
NAME                     DESCRIPTION
GENERATOR          VALUE
SOURCE_REPOSITORY_URL        The URL of the repository with your application source code
https://github.com/sclorg/rails-ex.git
SOURCE_REPOSITORY_REF        Set this to a branch name, tag or other ref of your repository if
you are not using the default branch
CONTEXT_DIR                  Set this to the relative path to your project if it is not in the root of your
repository
APPLICATION_DOMAIN           The exposed hostname that will route to the Rails service
rails-postgresql-example.openshiftapps.com
GITHUB_WEBHOOK_SECRET        A secret string used to configure the GitHub webhook
expression         [a-zA-Z0-9]{40}
SECRET_KEY_BASE              Your secret key for verifying the integrity of signed cookies
expression         [a-z0-9]{127}
APPLICATION_USER             The application user that is used within the sample application to
authorize access on pages                    openshift
APPLICATION_PASSWORD         The application password that is used within the sample
application to authorize access on pages                     secret
DATABASE_SERVICE_NAME        Database service name
postgresql
POSTGRESQL_USER              database username
expression         user[A-Z0-9]{3}
POSTGRESQL_PASSWORD          database password
expression         [a-zA-Z0-9]{8}
POSTGRESQL_DATABASE          database name
root
POSTGRESQL_MAX_CONNECTIONS   database max connections
10
POSTGRESQL_SHARED_BUFFERS    database shared buffers
12MB
```

この出力から、テンプレートの処理時に正規表現のようなジェネレーターで生成された複数のパラメーターを特定できます。

10.4.3. オブジェクト一覧の生成

CLI を使用して、標準出力にオブジェクト一覧を返すテンプレートを定義するファイルを処理できます。

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

または、テンプレートがすでに現在のプロジェクトにアップロードされている場合には以下を実行します。

```
$ oc process <template_name>
```

テンプレートを処理し、**oc create** の出力をパイプして、テンプレートからオブジェクトを作成することができます。

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

または、テンプレートがすでに現在のプロジェクトにアップロードされている場合には以下を実行します。

```
$ oc process <template> | oc create -f -
```

上書きする **<name>=<value>** の各ペアに **-p** オプションを追加することで、ファイルに定義された parameter の値を上書きできます。パラメーター参照は、テンプレートアイテム内のテキストフィールドに表示される場合があります。

たとえば、テンプレートの以下の **POSTGRESQL_USER** および **POSTGRESQL_DATABASE** パラメーターを上書きし、カスタマイズされた環境変数の設定を出力します。

例10.1 テンプレートからのオブジェクト一覧の作成

```
$ oc process -f my-rails-postgresql \
    -p POSTGRESQL_USER=bob \
    -p POSTGRESQL_DATABASE=mydatabase
```

JSON ファイルは、ファイルにリダイレクトすることも、**oc create** コマンドで処理済みの出力をパイプして、テンプレートをアップロードせずに直接適用することも可能です。

```
$ oc process -f my-rails-postgresql \
    -p POSTGRESQL_USER=bob \
    -p POSTGRESQL_DATABASE=mydatabase \
    | oc create -f -
```

多数のパラメーターがある場合は、それらをファイルに保存してからそのファイルを **oc process** に渡すことができます。

```
$ cat postgres.env
POSTGRESQL_USER=bob
POSTGRESQL_DATABASE=mydatabase
$ oc process -f my-rails-postgresql --param-file=postgres.env
```

--param-file の引数として **"-"** を使用して、標準入力から環境を読み込むこともできます。

```
$ sed s/bob/alice/ postgres.env | oc process -f my-rails-postgresql --param-file=-
```

10.5. アップロードしたテンプレートの変更

以下のコマンドを使用して、すでにプロジェクトにアップロードされているテンプレートを編集できます。

```
$ oc edit template <template>
```

10.6. インスタントアプリおよびクイックスタートテンプレートの使用

OpenShift Container Platform では、デフォルトで、インスタントアプリとクイックスタートテンプレートを複数提供しており、各種言語で簡単に新規アプリの構築を開始できます。Rails (Ruby)、Django (Python)、Node.js、CakePHP (PHP) および Dancer (Perl) 用のテンプレートを利用できます。クラスター管理者は、これらのテンプレートを利用できるようにデフォルトのグローバル **openshift** プロジェクトにこれらのテンプレートを作成しているはずです。以下のように、利用可能なデフォルトのインスタントアプリとクイックスタートテンプレートを一覧表示できます。

```
$ oc get templates -n openshift
```

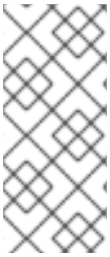
見つからない場合には、クラスター管理者に「Loading the Default Image Streams and Templates」のトピックを参照してもらうようにしてください。

デフォルトで、テンプレートビルドは必要なアプリケーションコードが含まれる GitHub の公開ソースリポジトリーを使用して行われます。ソースを変更して、独自のバージョンのアプリケーションをビルドするには、以下を実行する必要があります。

1. テンプレートのデフォルト **SOURCE_REPOSITORY_URL** パラメーターが参照するリポジトリーをフォークします。
2. テンプレートから作成する場合には、**SOURCE_REPOSITORY_URL** パラメーターの値を上書きします。 デフォルト値ではなく、フォークを指定してください。

これにより、テンプレートで作成したビルド設定はアプリケーションコードのフォークを参照するようになり、コードを更新して、自由にアプリケーションをリビルドできます。

Web コンソールを使用してこのプロセスを行う場合は、「Getting Started for Developers: Web Console」を参照してください。



注記

インスタンスアプリおよびクイックスタートアプリのテンプレートで、データベースのデプロイメント設定 を定義します。テンプレートが定義する設定では、データベースコンテツ用に一時ストレージを使用します。データベース Pod が何らかの理由で再起動されると、データベースの全データが失われてしまうので、これらのテンプレートは、デモ目的でのみ使用する必要があります。

10.7. テンプレートの記述

アプリケーションの全オブジェクトを簡単に再作成するために、新規テンプレートを定義できます。テンプレートでは、作成するオブジェクトと、これらのオブジェクトの作成をガイドするメタデータを定義します。

例10.2 単純なテンプレートオブジェクト定義 (YAML)

```
apiVersion: v1
kind: Template
metadata:
  name: redis-template
  annotations:
    description: "Description"
    iconClass: "icon-redis"
    tags: "database,nosql"
objects:
- apiVersion: v1
  kind: Pod
  metadata:
    name: redis-master
  spec:
    containers:
    - env:
      - name: REDIS_PASSWORD
        value: ${REDIS_PASSWORD}
      image: dockerfile/redis
      name: master
      ports:
      - containerPort: 6379
        protocol: TCP
parameters:
- description: Password used for Redis authentication
  from: '[A-Z0-9]{8}'
  generate: expression
  name: REDIS_PASSWORD
labels:
  redis: master
```

10.7.1. 詳細

テンプレートの説明では、テンプレートの内容に関する情報を提供でき、Web コンソールでの検索時に役立ちます。テンプレート名以外のメタデータは任意ですが、使用できると便利です。メタデータには、一般的な説明などの情報以外にタグのセットも含まれます。便利なタグにはテンプレートで使用する言語名などがあります (例: **java**、**php**、**ruby**)。

例10.3 テンプレート記述メタデータ

```
kind: Template
apiVersion: v1
metadata:
  name: cakephp-mysql-example 1
  annotations:
    openshift.io/display-name: "CakePHP MySQL Example (Ephemeral)" 2
    description: >-
      An example CakePHP application with a MySQL database. For more information
      about using this template, including OpenShift considerations, see
      https://github.com/sclorg/cakephp-ex/blob/master/README.md.


      WARNING: Any data stored will be lost upon pod destruction. Only use this
```

```
    template for testing." 3
  openshift.io/long-description: >-
    This template defines resources needed to develop a CakePHP application,
    including a build configuration, application deployment configuration, and
    database deployment configuration.  The database is stored in
    non-persistent storage, so this configuration should be used for
    experimental purposes only. 4
  tags: "quickstart,php,cakephp" 5
  iconClass: icon-php 6
  openshift.io/provider-display-name: "Red Hat, Inc." 7
  openshift.io/documentation-url: "https://github.com/sclorg/cakephp-ex" 8
  openshift.io/support-url: "https://access.redhat.com" 9
message: "Your admin credentials are ${ADMIN_USERNAME}:${ADMIN_PASSWORD}" 10
```

1. テンプレートの一意の名前。
2. ユーザーインターフェースで利用できるように、ユーザーに分かりやすく、簡単な名前。
3. テンプレートの説明。デプロイされる内容、デプロイ前に知っておく必要のある注意点をユーザーができるように詳細を追加します。**README** など、追加情報へのリンクも追加できます。パラグラフを作成するには、改行を追加できます。
4. 追加の説明。たとえば、サービスカタログに表示されます。
5. 検索およびグループ化を実行するためにテンプレートに関連付けられるタグ。指定のカタログカテゴリーの 1 つに含まれるように、タグを追加します。コンソールの定数ファイルの **CATALOG_CATEGORIES** で **id** および **categoryAliases** を参照してください。カテゴリーはクラスター全体に対して カスタマイズ することもできます。
6. Web コンソールでテンプレートと一緒に表示されるアイコン。可能な場合は、既存のロゴアイコンから選択します。また、FontAwesome および PatternFly からもアイコンを使用できます。または、テンプレートを使用する OpenShift Container Platform クラスターに CSS カスタマイズ を追加できるので、CSS カスタマイズ経由でアイコンを提供します。存在するアイコンクラスを指定するようにしてください。 指定しないと、汎用アイコンにフォールバックできなくなります。
7. テンプレートを提供する人または組織の名前
8. テンプレートに関する他のドキュメントを参照する URL
9. テンプレートに関するサポートを取得できる URL
10. テンプレートがインスタンス化された時に表示される説明メッセージ。このフィールドで、新規作成されたリソースの使用方法をユーザーに通知します。生成された認証情報や他のパラメーターを出力に追加できるように、メッセージの表示前にパラメーターの置換が行われます。ユーザーが従うべき次の手順が記載されたドキュメントへのリンクを追加してください。

10.7.2. ラベル

テンプレートには ラベル のセットを追加できます。これらのラベルは、テンプレートがインスタンス化される時に作成されるオブジェクトごとに追加します。このようにラベルを定義すると、特定のテンプレートから作成された全オブジェクトの検索、管理が簡単になります。

例10.4 テンプレートオブジェクトのラベル

```
kind: "Template"
apiVersion: "v1"
...
labels:
  template: "cakephp-mysql-example" 1
  app: "${NAME}" 2
```

1 このテンプレートから作成する全オブジェクトに適用されるラベル

2 パラメーター化されたラベル。このラベルは、このテンプレートを基に作成された全オブジェクトに適用されます。パラメーターは、ラベルキーおよび値の両方で拡張されます。

10.7.3. パラメーター

パラメーターにより、テンプレートがインスタンス化される時に値を生成するか、ユーザーが値を指定できるようになります。パラメーターが参照されると、値が置換されます。参照は、オブジェクト一覧フィールドであればどこでも定義できます。これは、無作為にパスワードを作成したり、テンプレートのカスタマイズに必要なユーザー固有の値やホスト名を指定したりできるので便利です。パラメーターは、2 種類の方法で参照可能です。

- 文字列の値として、テンプレートの文字列フィールドに **${PARAMETER_NAME}** の形式で配置する
- json/yaml の値として、テンプレートのフィールドに **${{PARAMETER_NAME}}** の形式で配置する

${PARAMETER_NAME}** 構文を使用すると、複数のパラメーター参照を 1 つのフィールドに統合でき、**"http://${PARAMETER_1}${PARAMETER_2}" などのように、参照を固定データ内に埋め込むことができます。どちらのパラメーター値も置換されて、引用された文字列が最終的な値になります。

${{PARAMETER_NAME}} 構文のみを使用する場合は、単一のパラメーター参照のみが許可され、先頭文字や終了文字は使用できません。結果の値は、置換後に結果が有効な json オブジェクトの場合は引用されません。結果が有効な json 値でない場合に、結果の値は引用され、標準の文字列として処理されます。

単一のパラメーターは、テンプレート内で複数回参照でき、1 つのテンプレート内で両方の置換構文を使用して参照することができます。

デフォルト値を指定でき、ユーザーが別の値を指定していない場合に使用されます。

例10.5 デフォルト値として明示的な値の設定

```
parameters:
  - name: USERNAME
    description: "The user name for Joe"
    value: joe
```

パラメーター値は、パラメーター定義に指定したルールを基に生成することも可能です。

例10.6 パラメーター値の生成

```
parameters:
  - name: PASSWORD
    description: "The random user password"
    generate: expression
    from: "[a-zA-Z0-9]{12}"
```

上記の例では、処理後に、大文字、小文字、数字すべてを含む 12 文字長のパスワードが無作為に作成されます。

利用可能な構文は、完全な正規表現構文ではありません。ただし、**\w**、**\d**、および **\a** 修飾子を使用できます。

- **[\w]{10}** は、10 桁の英字、数字、およびアンダースコアを生成します。これは PCRE 標準に準拠し、**[a-zA-Z0-9_]{10}** に相当します。
- **[\d]{10}** は 10 桁の数字を生成します。これは **[0-9]{10}** に相当します。
- **[\a]{10}** は 10 桁の英字を生成します。これは **[a-zA-Z]{10}** に相当します。

以下は、パラメーター定義と参照を含む完全なテンプレートの例です。

例10.7 パラメーター定義と参照を含む完全なテンプレート

```
kind: Template
apiVersion: v1
metadata:
  name: my-template
objects:
  - kind: BuildConfig
    apiVersion: v1
    metadata:
      name: cakephp-mysql-example
      annotations:
        description: Defines how to build the application
    spec:
      source:
        type: Git
        git:
          uri: "${SOURCE_REPOSITORY_URL}" 1
          ref: "${SOURCE_REPOSITORY_REF}"
        contextDir: "${CONTEXT_DIR}"
  - kind: DeploymentConfig
    apiVersion: v1
    metadata:
      name: frontend
    spec:
      replicas: "${{REPLICA_COUNT}}" 2
parameters:
  - name: SOURCE_REPOSITORY_URL 3
    displayName: Source Repository URL 4
    description: The URL of the repository with your application source code 5
```

```
    value: https://github.com/sclorg/cakephp-ex.git 6
    required: true 7
  - name: GITHUB_WEBHOOK_SECRET
    description: A secret string used to configure the GitHub webhook
    generate: expression 8
    from: "[a-zA-Z0-9]{40}" 9
  - name: REPLICA_COUNT
    description: Number of replicas to run
    value: "2"
    required: true
message: "... The GitHub webhook secret is ${GITHUB_WEBHOOK_SECRET} ..." 10
```

1. この値は、テンプレートがインスタンス化された時点で **SOURCE_REPOSITORY_URL** パラメーターに置き換えられます。
2. この値は、テンプレートがインスタンス化された時点で、**REPLICA_COUNT** パラメーターの引用なしの値に置き換えられます。
3. パラメーター名。この値は、テンプレート内でパラメーターを参照するのに使用します。
4. 分かりやすいパラメーターの名前。これは、ユーザーに表示されます。
5. パラメーターの説明。期待値に対する制約など、パラメーターの目的を詳細にわたり説明します。説明には、コンソールのテキスト標準に従い、完結した文章を使用するようにしてください。表示名と同じ内容を使用しないでください。
6. テンプレートをインスタンス化する時に、ユーザーにより値が上書きされない場合に使用されるパラメーターのデフォルト値。パスワードなどのデフォルト値の使用を避けるようにしてください。 シークレットと組み合わせた生成パラメーターを使用するようにしてください。
7. このパラメーターが必須であることを示します。つまり、ユーザーは空の値で上書きできません。パラメーターでデフォルト値または生成値が指定されていない場合には、ユーザーは値を指定する必要があります。
8. 値が生成されるパラメーター
9. ジェネレーターへの入力。この場合、ジェネレーターは、大文字、小文字を含む 40 桁の英数字の値を生成します。
10. パラメーターはテンプレートメッセージに含めることができます。これにより、生成された値がユーザーに通知されます。

10.7.4. オブジェクト一覧

テンプレートの主な部分は、テンプレートがインスタンス化される時に作成されるオブジェクトの一覧です。これには、**BuildConfig**、**DeploymentConfig**、**Service** などの 有効な API オブジェクト を使用できます。オブジェクトは、ここで定義された通りに作成され、パラメーターの値は作成前に置換されます。これらのオブジェクトの定義では、以前に定義したパラメーターを参照できます。

```
kind: "Template"
apiVersion: "v1"
metadata:
  name: my-template
```

```
objects:
  - kind: "Service" 1
    apiVersion: "v1"
    metadata:
      name: "cakephp-mysql-example"
      annotations:
        description: "Exposes and load balances the application pods"
    spec:
      ports:
        - name: "web"
          port: 8080
          targetPort: 8080
      selector:
        name: "cakephp-mysql-example"
```

1 **Service** の定義。 このテンプレートにより作成されます。

注記

オブジェクト定義のメタデータに **namespace** フィールドの固定値が含まれる場合、フィールドはテンプレートのインスタンス化の際に定義から取り除かれます。**namespace** フィールドにパラメーター参照が含まれる場合には、通常のパラメーター置換が行われ、パラメーター置換が値を解決した namespace で、オブジェクトが作成されます。 この際、ユーザーは対象の namespace でオブジェクトを作成するパーミッションがあることが前提です。

10.7.5. バインド可能なテンプレートの作成

テンプレートサービスブローカーは、認識されているテンプレートオブジェクトごとに、カタログ内にサービスを 1 つ公開します。デフォルトでは、これらのサービスはそれぞれ「バインド可能」として公開され、エンドユーザーがプロビジョニングしたサービスに対してバインドできるようにします。

テンプレートの作成者は、**template.openshift.io/bindable: "false"** のアノテーションをテンプレートに追加して、エンドユーザーが、指定のテンプレートからプロビジョニングされるサービスをバインドできないようにできます。

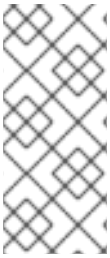
10.7.6. オブジェクトフィールドの公開

テンプレートの作成者は、テンプレートに含まれる特定のオブジェクトのフィールドを公開すべきかどうかを指定できます。テンプレートサービスのブローカーは、ConfigMap、Secret、Service、Route オブジェクトに公開されたフィールドを認識し、ユーザーがブローカーでバックされているサービスをバンドした場合に公開されたフィールドの値を返します。

オブジェクトのフィールドを 1 つまたは複数公開するには、テンプレート内のオブジェクトに、プレフィックスが **template.openshift.io/expose-** または **template.openshift.io/base64-expose-** のアノテーションを追加します。

各アノテーションキーは、**bind** 応答のキーになるように、プレフィックスが削除されてパススルーされます。

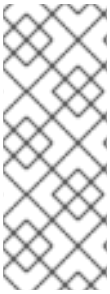
各アノテーションの値は Kubernetes JSONPath 式 の値であり、バインド時に解決され、**bind** 応答で返される値が含まれるオブジェクトフィールドを指定します。



注記

Bind 応答のキー/値のペアは、環境変数として、システムの他の場所で使用できます。そのため、アノテーションキーでプレフィックスを取り除いた値を有効な環境変数名として使用することが推奨されます。先頭に**A-Z**、**a-z** またはアンダースコアを指定して、その後に、ゼロか、他の文字 **A-Z**、**a-z**、**0-9** またはアンダースコアを指定してください。

template.openshift.io/expose- アノテーションを使用して、文字列としてフィールドの値を返します。これは、任意のバイナリーデータを処理しないものの、便利な方法です。バイナリーデータを返す場合には、バイナリーデータを返す前にデータのエンコードに base64b を使用するのではなく、**template.openshift.io/base64-expose-** アノテーションを使用します。



注記

バックスラッシュでエスケープしない限り、Kubernetes の JSONPath 実装は表現内のどの場所に使用されていても、**.**、**@** などはメタ文字として解釈されます。そのため、たとえば、**my.key** という名前の **ConfigMap** のデータを参照するには、JSONPath 式は **{.data['my\.key']}** とする必要があります。JSONPath 式が YAML でどのように記述されているかによって、**"{.data['my\\.key']}"** などのように、追加でバックスラッシュが必要になる場合があります。

以下は、公開されるさまざまなオブジェクトのフィールドの例です。

```
kind: Template
apiVersion: v1
metadata:
  name: my-template
objects:
- kind: ConfigMap
  apiVersion: v1
  metadata:
    name: my-template-config
    annotations:
      template.openshift.io/expose-username: "{.data['my\\.username']}"
  data:
    my.username: foo
- kind: Secret
  apiVersion: v1
  metadata:
    name: my-template-config-secret
    annotations:
      template.openshift.io/base64-expose-password: "{.data['password']}"
  stringData:
    password: bar
- kind: Service
  apiVersion: v1
  metadata:
    name: my-template-service
    annotations:
      template.openshift.io/expose-service_ip_port: "{.spec.clusterIP}:{.spec.ports[?
(.name==\"web\")].port}"
  spec:
    ports:
    - name: "web"
```

```
      port: 8080
- kind: Route
  apiVersion: v1
  metadata:
    name: my-template-route
    annotations:
      template.openshift.io/expose-uri: "http://{.spec.host}{.spec.path}"
  spec:
    path: mypath
```

上記の部分的なテンプレートでの **bind** 操作に対する応答例は以下のようになります。

```
{
  "credentials": {
    "username": "foo",
    "password": "YmFy",
    "service_ip_port": "172.30.12.34:8080",
    "uri": "http://route-test.router.default.svc.cluster.local/mypath"
  }
}
```

10.7.7. テンプレートの準備ができるまで待機

テンプレートの作成者は、テンプレート内の特定のオブジェクトがサービスカタログ、Template Service Broker または TemplateInstance API によるテンプレートのインスタンス化が完了したとされるまで待機する必要があるかを指定できます。

この機能を使用するには、テンプレート内の **Build**、**BuildConfig**、**Deployment**、**DeploymentConfig**、**Job** または **StatefulSet** のオブジェクト 1 つ以上に、次のアノテーションでマークを付けてください。

```
"template.alpha.openshift.io/wait-for-ready": "true"
```

テンプレートのインスタンス化は、アノテーションのマークが付けられたすべてのオブジェクトが準備できたと報告されるまで、完了しません。同様に、アノテーションが付けられたオブジェクトが失敗したと報告されるか、固定タイムアウトである 1 時間以内にテンプレートの準備が整わなかった場合に、テンプレートのインスタンス化は失敗します。

インスタンス化の目的で、各オブジェクトの種類の準備状態および失敗は以下のように定義されます。

種類	準備状態 (Readines)	失敗 (Failure)
Build	オブジェクトが Complete (完了) フェーズを報告する	オブジェクトが Canceled (キャンセル)、Error (エラー)、または Failed (失敗) を報告する
BuildConfig	関連付けられた最新のビルドオブジェクトが Complete (完了) フェーズを報告する	関連付けられた最新のビルドオブジェクトが Canceled (キャンセル)、Error (エラー)、または Failed (失敗) を報告する

種類	準備状態 (Readines)	失敗 (Failure)
Deployment	オブジェクトが新しい ReplicaSet やデプロイメントが利用可能であることを報告する (これはオブジェクトに定義された readiness プローブに従います)	オブジェクトで、Progressing (進捗中) の状態が false であると報告される
DeploymentConfig	オブジェクトが新しい ReplicaController やデプロイメントが利用可能であると報告する (これはオブジェクトに定義された readiness プローブに従います)	オブジェクトで、Progressing (進捗中) の状態が false であると報告される
Job	オブジェクトが完了 (completion) を報告する	オブジェクトが 1 つ以上の失敗が発生したことを報告する
StatefulSet	オブジェクトがすべてのレプリカが準備状態にあることを報告する (これはオブジェクトに定義された readiness プローブに従います)	該当なし

以下は、テンプレートサンプルを一部抜粋したものです。この例では、**wait-for-ready** アノテーションが使用されています。他のサンプルは、OpenShift クイックスタートテンプレートにあります。

```
kind: Template
apiVersion: v1
metadata:
  name: my-template
objects:
- kind: BuildConfig
  apiVersion: v1
  metadata:
    name: ...
    annotations:
      # wait-for-ready used on BuildConfig ensures that template instantiation
      # will fail immediately if build fails
      template.alpha.openshift.io/wait-for-ready: "true"
  spec:
    ...
- kind: DeploymentConfig
  apiVersion: v1
  metadata:
    name: ...
    annotations:
      template.alpha.openshift.io/wait-for-ready: "true"
  spec:
    ...
- kind: Service
  apiVersion: v1
  metadata:
    name: ...
  spec:
    ...
```

10.7.8. その他の推奨事項

- アプリケーションがスムーズ に実行するのに十分なリソースが提供されるようにメモリー、CPU、および ストレージ のデフォルトサイズを設定します。
- **latest** タグが複数のメジャーバージョンで使用されている場合には、イメージからこのタグを参照しないようにします。新規イメージがそのタグにプッシュされると、実行中のアプリケーションが破損してしまう可能性があります。
- 適切なテンプレートの場合、テンプレートのデプロイ後に変更の必要なく、クリーンにビルド、デプロイが行われます。

10.7.9. 既存オブジェクトからのテンプレートの作成

テンプレートをゼロから作成するのではなく、プロジェクトから既存のオブジェクトをテンプレート形式でエクスポートして、パラメーターおよび他のカスタマイズを追加して、テンプレート形式を変更することができます。プロジェクトのオブジェクトをテンプレート形式でエクスポートするには、以下を実行します。

```
$ oc export all --as-template=<template_name> > <template_filename>
```

all ではなく、特定のリソースタイプや複数のリソースを置き換えることも可能です。他の例については、**oc export -h** を実行します。

以下は、**oc export all** に含まれるオブジェクトタイプです。

- BuildConfig
- Build
- DeploymentConfig
- ImageStream
- Pod
- ReplicationController
- Route
- Service

第11章 コンテナーへのリモートシェルを開く

11.1. 概要

oc rsh コマンドを使用すると、システム上にある各種のツールにローカルでアクセスし、管理することができます。セキュアシェル (SSH) は、アプリケーションへのセキュアな接続を提供する基礎となるテクノロジーであり、これは業界標準になっています。シェル環境を使ったアプリケーションへのアクセスは Security-Enhanced Linux (SELinux) ポリシーで保護され、制限されます。

11.2. セキュアなシェルセッションの開始

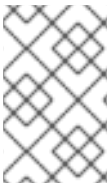
コンテナーへのリモートシェルセッションを開きます。

```
$ oc rsh <pod>
```

リモートシェルの使用時には、コンテナー内で実行しているかのようにコマンドを実行でき、モニタリングやデバッグ、およびコンテナー内で実行されているものに固有の CLI コマンドの使用などのローカルの操作を実行できます。

たとえば MySQL コンテナーの場合、**mysql** コマンドを起動し、プロンプトを使用して **SELECT** コマンド を入力することでデータベース内のレコード数をカウントできます。また、検証には **ps(1)** および **ls(1)** などのコマンドを使用することもできます。

BuildConfigs および **DeployConfigs** は内容の表示方法や、(コンテナーを内部に含む) Pod を必要に応じて作成し、取り外す方法を定義します。加えられる変更は永続化しません。コンテナー内で直接変更を加えても、そのコンテナーが破棄され再ビルドされると加えた変更は存在しなくなります。



注記

oc exec はコマンドをリモートで実行するために使用できます。しかしながら、**oc rsh** コマンドの使用がリモートシェルを永続的に開いた状態にするより簡単な方法になります。

11.3. セキュアなシェルセッションのヘルプ

使用方法やオプションについてのヘルプ、例を参照するには、以下を実行します。

```
$ oc rsh -h
```

第12章 サービスアカウント

12.1. 概要

ユーザーが OpenShift Container Platform CLI または web コンソールを使用する場合、API トークンは OpenShift API に対して認証を行いますが、一般ユーザーの認証情報が利用できない場合、通常各種コンポーネントが API 呼び出しを別個に実行します。ただし、一般ユーザーの認証情報を利用できない場合、以下のようにコンポーネントが API 呼び出しを行うのが通例になります。以下に例を示します。

- レプリケーションコントローラーが Pod を作成するか、または削除するために API 呼び出しを実行する。
- コンテナー内のアプリケーションが検出の目的で API 呼び出しを実行する。
- 外部アプリケーションがモニタリングおよび統合目的で API 呼び出しを実行する。

サービスアカウントは、一般ユーザーの認証情報を共有せずに API アクセスをより柔軟に制御する方法を提供します。

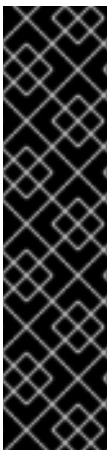
12.2. ユーザー名およびグループ

すべてのサービスアカウントには、一般ユーザーのようにロールを付与できるユーザー名が関連付けられています。ユーザー名はそのプロジェクトおよび名前から派生します。

```
system:serviceaccount:<project>:<name>
```

たとえば、**view (表示)** ロールを **top-secret** プロジェクトの **robot** サービスアカウントに追加するには、以下を実行します。

```
$ oc policy add-role-to-user view system:serviceaccount:top-secret:robot
```



重要

プロジェクトで特定のサービスアカウントにアクセスを付与する必要がある場合は、**-z** フラグを使用できます。サービスアカウントが属するプロジェクトから **-z** フラグを使用し、**<serviceaccount_name>** を指定します。これによりタイプミスの発生する可能性が減り、アクセスを指定したサービスアカウントのみに付与できるため、この方法を使用することを強くお勧めします。以下に例を示します。

```
$ oc policy add-role-to-user <role_name> -z <serviceaccount_name>
```

プロジェクトから実行しない場合は、以下の例に示すように **-n** オプションを使用してこれが適用されるプロジェクトの namespace を指定します。

すべてのサービスアカウントは以下の 2 つのグループのメンバーでもあります。

system:serviceaccount
: システムのすべてのサービスアカウントが含まれます。

system:serviceaccount:<project>
: 指定されたプロジェクトのすべてのサービスアカウントが含まれます。

たとえば、すべてのプロジェクトのすべてのサービスアカウントが **top-secret** プロジェクトのリソースを表示できるようにするには、以下を実行します。

```
$ oc policy add-role-to-group view system:serviceaccount -n top-secret
```

managers プロジェクトのすべてのサービスアカウントが **top-secret** プロジェクトのリソースを編集できるようにするには、以下を実行します。

```
$ oc policy add-role-to-group edit system:serviceaccount:managers -n top-secret
```

12.3. デフォルトのサービスアカウントおよびロール

3 つのサービスアカウントがすべてのプロジェクトで自動的に作成されます。

サービスアカウント	使用法
builder	ビルド Pod で使用されます。これには **system:image-builder** ロールが付与されます。 このロールは、内部 Docker レジストリーを使用してイメージをプロジェクトのイメージストリームにプッシュすることを可能にします。
deployer	デプロイメント Pod で使用され、**system:deployer** ロールが付与されます。 このロールは、プロジェクトでレプリケーションコントローラーや Pod を表示したり、変更したりすることを可能にします。
default	別のサービスアカウントが指定されていない限り、その他すべての Pod を実行するために使用されます。

プロジェクトのすべてのサービスアカウントには **system:image-puller** ロールが付与されます。 このロールは、内部 Docker レジストリーを使用してイメージをイメージストリームからプルすることを可能にします。

12.4. サービスアカウントの管理

サービスアカウントは、各プロジェクトに存在する API オブジェクトです。サービスアカウントを管理するには、**sa** または **serviceaccount** オブジェクトタイプと共に **oc** コマンドを使用するか、または web コンソールを使用することができます。

現在のプロジェクトの既存のサービスアカウントの一覧を取得するには、以下を実行します。

```
$ oc get sa
NAME      SECRETS   AGE
builder   2         2d
default   2         2d
deployer  2         2d
```

新規のサービスアカウントを作成するには、以下を実行します。

```
$ oc create sa robot
serviceaccount "robot" created
```

サービスアカウントの作成後すぐに、以下の 2 つのシークレットが自動的に追加されます。

- API トークン
- OpenShift Container レジストリーの認証情報

これらはサービスアカウントを記述すると表示できます。

```
$ oc describe sa robot
Name:  robot
Namespace: project1
Labels:  <none>
Annotations: <none>

Image pull secrets: robot-dockercfg-qzbhb

Mountable secrets:  robot-token-f4khf
                    robot-dockercfg-qzbhb

Tokens:             robot-token-f4khf
                    robot-token-z8h44
```

システムは、サービスアカウントが常に API トークンとレジストリーの認証情報を持っていることを保証します。

生成される API トークンとレジストリーの認証情報は期限切れになることはありませんが、シークレットを削除することで取り消すことができます。シークレットが削除されると、新規のシークレットが自動生成され、これに置き換わります。

12.5. サービスアカウント認証の有効化

サービスアカウントは、プライベート RSA キーで署名されるトークンを使用して API に対して認証されます。認証層では一致するパブリック RSA キーを使用して署名を検証します。

サービスアカウントトークンの生成を有効にするには、マスターで **/etc/origin/master/master-config.yml** ファイルの **serviceAccountConfig** スタンザを更新し、(署名 用に) **privateKeyFile** と **publicKeyFiles** 一覧の一致するパブリックキーファイルを指定します。

```
serviceAccountConfig:
  ...
  masterCA: ca.crt (1)
  privateKeyFile: serviceaccount.private.key (2)
  publicKeyFiles:
  - serviceaccount.public.key (3)
  - ...
```

1. API サーバーの提供する証明書を検証するために使用される CA ファイル。
2. プライベート RSA キーファイル (トークンの署名用)。
3. パブリック RSA キーファイル (トークンの検証用)。プライベートキーファイルが提供されている場合、パブリックキーコンポーネントが使用されます。複数のパブリックキーファイルを使用でき、トークンはパブリックキーのいずれかで検証できる場合に受け入れられます。これにより、署名するキーのローテーションが可能となり、以前の署名者が生成したトークンは依然として受け入れられます。

12.6. 管理サービスアカウント

サービスアカウントは、ビルド、デプロイメントおよびその他の Pod を実行するために各プロジェクトで必要になります。マスターの **/etc/origin/master/master-config.yml** ファイルの **managedNames** 設定は、すべてのプロジェクトに自動作成されるサービスアカウントを制御します。

```
serviceAccountConfig:
  ...
  managedNames: 1
  - builder 2
  - deployer 3
  - default 4
  - ...
```

1. すべてのプロジェクトで自動作成するサービスアカウントの一覧。
2. 各プロジェクトの **ビルダー** サービスアカウントはビルド Pod で必要になり、**system:image-builder** ロールが付与されます。このロールは、内部コンテナーレジストリーを使用してイメージをプロジェクトのイメージストリームにプッシュすることを可能にします。
3. 各プロジェクトの **deployer** サービスアカウントはデプロイメント Pod で必要になり、レプリケーションコントローラーおよびプロジェクトの Pod の表示および変更を可能にする **system:deployer** ロールが付与されます。
4. **デフォルト**のサービスアカウントは、別のサービスアカウントが指定されない限り、他のすべての Pod で使用されます。

プロジェクトのすべてのサービスアカウントには **system:image-puller** ロールが付与されます。このロールは、内部コンテナーレジストリーを使用してイメージをイメージストリームからプルすることを可能にします。

12.7. インフラストラクチャーサービスアカウント

一部のインフラストラクチャーコントローラーは、サービスアカウント認証情報を使用して実行されます。以下のサービスアカウントは、サーバーの起動時に OpenShift Container Platform インフラストラクチャープロジェクト (**openshift-infra**) に作成され、クラスター全体で以下のロールが付与されます。

サービスアカウント	説明
replication-controller	system:replication-controller ロールの割り当て
deployment-controller	system:deployment-controller ロールの割り当て
build-controller	system:build-controller ロールの割り当て。さらに、**build-controller** サービスアカウントは、特権付きの ビルド Pod を作成するために特権付きセキュリティーコンテキストに組み込まれます。

これらのサービスアカウントが作成されるプロジェクトを設定するには、マスターで **/etc/origin/master/master-config.yml** ファイルの **openshiftInfrastructureNamespace** フィールドを設定します。

```
policyConfig:
  ...
  openshiftInfrastructureNamespace: openshift-infra
```

12.8. サービスアカウントおよびシークレット

マスターで **/etc/origin/master/master-config.yml** ファイルの **limitSecretReferences** フィールドを **true** に設定し、Pod のシークレット参照をサービスアカウントでホワイトリストに入れることが必要になるようにします。この値を **false** に設定すると、Pod がプロジェクトのすべてのシークレットを参照できるようになります。

```
serviceAccountConfig:
  ...
  limitSecretReferences: false
```

12.9. 許可されたシークレットの管理

API 認証情報を提供するほかに、Pod のサービスアカウントは Pod が使用できるシークレットを決定します。

Pod は以下の 2 つの方法でシークレットを使用します。

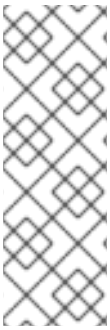
- イメージプルシークレットの使用: Pod のコンテナーのイメージをプルするために使用される認証情報を提供します。
- マウント可能なシークレットの使用: シークレットの内容をファイルとしてコンテナーに挿入します。

サービスアカウントの Pod がシークレットをイメージプルシークレットとして使用できるようにするには、以下を実行します。

```
$ oc secrets link --for=pull <serviceaccount-name> <secret-name>
```

サービスアカウントの Pod がシークレットをマウントできるようにするには、以下を実行します。

```
$ oc secrets link --for=mount <serviceaccount-name> <secret-name>
```



注記

シークレットを参照しているサービスアカウントにのみにシークレットを制限することはデフォルトで無効にされています。これは、**serviceAccountConfig.limitSecretReferences** がマスター設定ファイルで **false** (デフォルト設定) に設定されている場合はシークレットを **--for=mount** オプションを使ってサービスアカウントの Pod にマウントする必要がないことを意味します。ただし **serviceAccountConfig.limitSecretReferences** の値にかかわらず、**--for=pull** オプションを使用してイメージプルシークレットの使用を有効にする必要はあります。

以下の例では、シークレットを作成し、これをサービスアカウントに追加しています。

```
$ oc create secret generic secret-plans \
    --from-file=plan1.txt \
    --from-file=plan2.txt
```

```
secret/secret-plans

$ oc create secret docker-registry my-pull-secret \
    --docker-username=mastermind \
    --docker-password=12345 \
    --docker-email=mastermind@example.com
secret/my-pull-secret

$ oc secrets link robot secret-plans --for=mount

$ oc secrets link robot my-pull-secret --for=pull

$ oc describe serviceaccount robot
Name:               robot
Labels:             <none>
Image pull secrets: robot-dockercfg-624cx
                    my-pull-secret

Mountable secrets:  robot-token-uzkbh
                    robot-dockercfg-624cx
                    secret-plans

Tokens:             robot-token-8bhpp
                    robot-token-uzkbh
```

12.10. コンテナー内でのサービスアカウントの認証情報の使用

Pod が作成されると、Pod はサービスアカウントを指定し (またはデフォルトのサービスアカウントを使用し)、サービスアカウントの API 認証情報と参照されるシークレットを使用することができます。

Pod のサービスアカウントの API トークンが含まれるファイルは **/var/run/secrets/kubernetes.io/serviceaccount/token** に自動的にマウントされます。

このトークンは Pod のサービスアカウントとして API 呼び出しを実行するために使用できます。以下の例では、トークンによって識別されるユーザーについての情報を取得するために **users/~** API を呼び出しています。

```
$ TOKEN="$(cat /var/run/secrets/kubernetes.io/serviceaccount/token)"

$ curl --cacert /var/run/secrets/kubernetes.io/serviceaccount/ca.crt \
    "https://openshift.default.svc.cluster.local/oapi/v1/users/~" \
    -H "Authorization: Bearer $TOKEN"

kind: "User"
apiVersion: "user.openshift.io/v1"
metadata:
  name: "system:serviceaccount:top-secret:robot"
  selflink: "/oapi/v1/users/system:serviceaccount:top-secret:robot"
  creationTimestamp: null
identities: null
groups:
  - "system:serviceaccount"
  - "system:serviceaccount:top-secret"
```

12.11. サービスアカウントの認証情報の外部での使用

同じトークンを、API に対して認証する必要のある外部アプリケーションに配布することができます。

以下の構文を使用してサービスアカウントの API トークンを表示します。

```
$ oc describe secret <secret-name>
```

以下に例を示します。

```
$ oc describe secret robot-token-uzkbh -n top-secret
Name:  robot-token-uzkbh
Labels:  <none>
Annotations: kubernetes.io/service-account.name=robot,kubernetes.io/service-
account.uid=49f19e2e-16c6-11e5-afdc-3c970e4b7ffe

Type: kubernetes.io/service-account-token

Data

token: eyJhbGciOiJSUzI1NiIsInR5cCI6IkpXVCJ9...

$ oc login --token=eyJhbGciOiJSUzI1NiIsInR5cCI6IkpXVCJ9...
Logged into "https://server:8443" as "system:serviceaccount:top-secret:robot" using the token
provided.

You don't have any projects. You can try to create a new project, by running

    $ oc new-project <projectname>

$ oc whoami
system:serviceaccount:top-secret:robot
```

第13章 イメージの管理

13.1. 概要

イメージストリーム は、タグで識別される数多くの コンテナーイメージ で構成されます。これは Docker イメージリポジトリーのように関連イメージの単一仮想ビューを提供します。

イメージストリームの監視により、ビルドおよびデプロイメントは新規イメージの追加または変更時に通知を受信し、それぞれビルドまたはデプロイメントを実行してこれに対応します。

イメージのレジストリーが置かれる場所やレジストリー関連の認証要件、およびビルドやデプロイメントで必要とされる動作が何であるかによって、イメージと対話し、イメージストリームをセットアップする方法は異なり、数多くの方法でこれらを実行することができます。以下のセクションではこれらのトピックについて扱います。

13.2. イメージのタグ付け

OpenShift Container Platform イメージストリームとそのタグを使用する前に、コンテナーイメージのコンテキストにおけるイメージタグ全般について理解しておくと便利です。

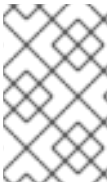
コンテナーイメージにはその内容を直感的に判別できるようにする名前 (**タグ**) を追加できます。タグの一般的な使用例として、イメージに含まれるもののバージョンを指定するために使用できます。**ruby** という名前のイメージがある場合、Ruby の 2.0 バージョン用に **2.0** という名前のタグを使用したり、リポジトリー全体における最新のビルドされたイメージを示す **latest** というタグを使用したりすることができます。

docker CLI を使用してイメージと直接対話する場合、**docker tag** コマンドを使用してタグを追加できます。基本的に、この操作は複数の部分で構成されるエイリアスをイメージに追加する操作です。 これには、以下が含まれます。

```
<registry_server>/<user_name>/<image_name>:<tag>
```

上記の **<user_name>** の部分は、イメージが内部レジストリー（OpenShift Container レジストリー）を使用して OpenShift Container Platform 環境に保存される場合には、プロジェクト または namespace も参照することがあります。

OpenShift Container Platform は **docker tag** コマンドに似た **oc tag** コマンドを提供しますが、これらはイメージ上で直接動作するのではなくイメージストリームで動作します。



注記

docker CLI を使用してイメージに直接タグ付けする方法についての詳細は、Red Hat Enterprise Linux 7 の『Getting Started with Containers』ドキュメントを参照してください。

13.2.1. タグのイメージストリームへの追加

OpenShift Container Platform のイメージストリームはタグで識別されるゼロまたは 1 つ以上のコンテナーイメージで構成されることに留意した上で、**oc tag** コマンドを使用してタグをイメージストリームに追加することができます。

```
$ oc tag <source> <destination>
```

たとえば、**ruby** イメージストリームの **static-2.0** タグを **ruby** イメージストリーム 2.0 タグの現行のイメージを常に参照するように設定するには、以下を実行します。

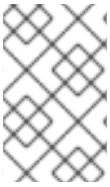
```
$ oc tag ruby:2.0 ruby:static-2.0
```

これにより、**ruby** イメージストリームに **static-2.0** という名前のイメージストリームタグが新たに作成されます。この新規タグは、**oc tag** の実行時に **ruby:2.0** イメージストリームタグが参照したイメージ ID を直接参照し、これが参照するイメージが変更されることがありません。

各種のタグを利用できます。デフォルト動作では、特定の時点の特定のイメージを参照する **永続** タグを使用します。 ソースが変更されても新規 (宛先) タグは変更されません。

トラッキング タグの場合は、宛先タグのメタデータがソースタグのインポート時に更新されます。宛先タグがソースタグの変更時に常に更新されるようにするには、**--alias=true** フラグを使用します。

```
$ oc tag --alias=true <source> <destination>
```



注記

永続的なエイリアス (**latest** または **stable** など) を作成するには **トラッキング** タグを使用します。このタグは単一イメージストリーム内で **のみ** 適切に機能します。複数のイメージストリーム間で使用されるエイリアスを作成しようとするとエラーが生じます。

さらに **--scheduled=true** フラグを追加して宛先タグが定期的に更新 (再インポートなど) されるようにできます。期間はシステムレベルで グローバルに設定 できます。詳細は、「タグおよびイメージメタデータのインポート」を参照してください。

--reference フラグはインポートされないイメージストリームを作成します。このタグはソースの場所を参照しますが、これを永続的に参照します。

Docker に対して統合レジストリーのタグ付けされたイメージを常にフェッチするよう指示するには、**--reference-policy=local** を使用します。レジストリーはプルスルー(pull-through)機能 を使用してイメージをクライアントに提供します。デフォルトで、イメージ Blob はレジストリーによってローカルにミラーリングされます。その結果、それらが次回必要となる場合により迅速にプルされます。またこのフラグは **--insecure-registry** を Docker デーモンに指定しなくても、イメージストリームに非セキュアなアノテーションがあるか、またはタグに非セキュアなインポートポリシーがある限り、非セキュアなレジストリーからのプルを許可します。

13.2.2. 推奨されるタグ付け規則

イメージは時間の経過と共に変化するもので、それらのタグはその変化を反映します。イメージタグはビルドされる最新イメージを常に参照します。

タグ名にあまりにも多くの情報が組み込まれる場合 (例: **v2.0.1-may-2016**)、タグはイメージの 1 つのリビジョンのみを参照し、更新されることがなくなります。デフォルトのイメージのプルーニングオプションを使用しても、このようなイメージは削除されません。非常に大規模なクラスターでは、イメージが修正されるたびに新規タグが作成される設定の場合、古くなって久しいイメージの余分のタグメタデータで etcd データストアが一杯になる可能性があります。

一方、タグの名前が **v2.0** である場合はイメージリビジョンの数が多くなることが予想されます。これにより タグ履歴 が長くなるため、イメージプルーナーが古くなり使われなくなったイメージを削除する可能性が高くなります。詳細は、「 イメージのプルーニング 」を参照してください。

タグの名前付け規則は各自で定めることができますが、ここでは **<image_name>:<image_tag>** 形式のいくつかの例を見てみましょう。

表13.1 イメージタグの名前付け規則

説明	例
リビジョン	**myimage:v2.0.1**
アーキテクチャー	**myimage:v2.0-x86_64**
ベースイメージ	**myimage:v1.2-centos7**
最新 (不安定な可能性がある)	**myimage:latest**
最新 (安定性がある)	**myimage:stable**

タグ名に日付を含める必要がある場合、古くなり使用されなくなったイメージおよび **istags** を定期的に検査し、これらを削除してください。そうしないと、古いイメージによるリソース使用量が増大する可能性があります。

13.2.3. タグのイメージストリームからの削除

タグをイメージストリームから完全に削除するには、以下を実行します。

```
$ oc delete istag/ruby:latest
```

または

```
$ oc tag -d ruby:latest
```

13.2.4. イメージストリームでのイメージの参照

以下の参照タイプを使用して、イメージをイメージストリームで参照できます。

- **ImageStreamTag** は、所定のイメージストリームおよびタグのイメージを参照し、取得するために使用されます。この名前は以下の規則に基づいて付けられます。

  ```
  <image_stream_name>:<tag>
  ```

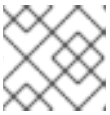
- **ImageStreamImage** は、所定のイメージストリームおよびイメージ名のイメージを参照し、取得するために使用されます。この名前は以下の規則に基づいて付けられます。

  ```
  <image_stream_name>@<id>
  ```

  <id> は、ダイジェストとも呼ばれる特定イメージのイミュータブルな ID です。

- **DockerImage** は、所定の外部レジストリーのイメージを参照し、取得するために使用されます。この名前は、以下のような標準の Docker **プル仕様** に基づいて付けられます。

  ```
  openshift/ruby-20-centos7:2.0
  ```



注記

タグが指定されていない場合、**latest** タグが使用されることが想定されます。

サードパーティーのレジストリーを参照することもできます。

```
registry.access.redhat.com/rhel7:latest
```

またはダイジェストでイメージを参照できます。

```
centos/ruby-22-
centos7@sha256:3a335d7d8a452970c5b4054ad7118ff134b3a6b50a2bb6d0c07c746e8986b28e
```

CentOS イメージストリームのサンプル などのイメージストリーム定義のサンプルを表示する場合、それらには **ImageStreamTag** の定義や **DockerImage** の参照が含まれる一方で、**ImageStreamImage** に関連するものは何も含まれていないことに気づかれることでしょう。

これは、イメージストリームでのイメージのインポートまたはイメージのタグ付けを行う場合は常に **ImageStreamImage** オブジェクトが OpenShift Container Platform に自動的に作成されるためです。イメージストリームを作成するために使用するイメージストリーム定義で **ImageStreamImage** オブジェクトを明示的に定義する必要はありません。

イメージのオブジェクト定義は、イメージストリーム名および ID を使用し、**ImageStreamImage** 定義を取得して確認することができます。

```
$ oc export isimage <image_stream_name>@<id>
```

注記

以下を実行して所定のイメージストリームの有効な **<id>** 値を確認することができます。

```
$ oc describe is <image_stream_name>
```

たとえば、**ruby** イメージストリームから **ruby@3a335d7** の名前および ID を使って **ImageStreamImage** を検索します。

ImageStreamImage で取得されるイメージオブジェクトの定義

```
$ oc export isimage ruby@3a335d7

apiVersion: v1
image:
  dockerImageLayers:
  - name: sha256:a3ed95caeb02ffe68cdd9fd84406680ae93d633cb16422d00e8a7c22955b46d4
    size: 0
  - name: sha256:ee1dd2cb6df21971f4af6de0f1d7782b81fb63156801cfde2bb47b4247c23c29
    size: 196634330
  - name: sha256:a3ed95caeb02ffe68cdd9fd84406680ae93d633cb16422d00e8a7c22955b46d4
    size: 0
  - name: sha256:a3ed95caeb02ffe68cdd9fd84406680ae93d633cb16422d00e8a7c22955b46d4
    size: 0
```

```
  - name: sha256:ca062656bff07f18bff46be00f40cfbb069687ec124ac0aa038fd676cfaea092
    size: 177723024
  - name: sha256:63d529c59c92843c395befd065de516ee9ed4995549f8218eac6ff088bfa6b6e
    size: 55679776
  dockerImageMetadata:
    Architecture: amd64
    Author: SoftwareCollections.org <sclorg@redhat.com>
    Config:
      Cmd:
      - /bin/sh
      - -c
      - $STI_SCRIPTS_PATH/usage
      Entrypoint:
      - container-entrypoint
      Env:
      - PATH=/opt/app-root/src/bin:/opt/app-
root/bin:/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin
      - STI_SCRIPTS_URL=image:///usr/libexec/s2i
      - STI_SCRIPTS_PATH=/usr/libexec/s2i
      - HOME=/opt/app-root/src
      - BASH_ENV=/opt/app-root/etc/scl_enable
      - ENV=/opt/app-root/etc/scl_enable
      - PROMPT_COMMAND=. /opt/app-root/etc/scl_enable
      - RUBY_VERSION=2.2
      ExposedPorts:
        8080/tcp: {}
      Image: d9c3abc5456a9461954ff0de8ae25e0e016aad35700594714d42b687564b1f51
      Labels:
        build-date: 2015-12-23
        io.k8s.description: Platform for building and running Ruby 2.2 applications
        io.k8s.display-name: Ruby 2.2
        io.openshift.builder-base-version: 8d95148
        io.openshift.builder-version: 8847438ba06307f86ac877465eadc835201241df
        io.openshift.s2i.scripts-url: image:///usr/libexec/s2i
        io.openshift.tags: builder,ruby,ruby22
        io.s2i.scripts-url: image:///usr/libexec/s2i
        license: GPLv2
        name: CentOS Base Image
        vendor: CentOS
      User: "1001"
      WorkingDir: /opt/app-root/src
    ContainerConfig: {}
    Created: 2016-01-26T21:07:27Z
    DockerVersion: 1.8.2-el7
    Id: 57b08d979c86f4500dc8cad639c9518744c8dd39447c055a3517dc9c18d6fccd
    Parent: d9c3abc5456a9461954ff0de8ae25e0e016aad35700594714d42b687564b1f51
    Size: 430037130
    apiVersion: "1.0"
    kind: DockerImage
  dockerImageMetadataVersion: "1.0"
  dockerImageReference: centos/ruby-22-
centos7@sha256:3a335d7d8a452970c5b4054ad7118ff134b3a6b50a2bb6d0c07c746e8986b28e
  metadata:
    creationTimestamp: 2016-01-29T13:17:45Z
    name: sha256:3a335d7d8a452970c5b4054ad7118ff134b3a6b50a2bb6d0c07c746e8986b28e
    resourceVersion: "352"
```

```
    uid: af2e7a0c-c68a-11e5-8a99-525400f25e34
kind: ImageStreamImage
metadata:
  creationTimestamp: null
  name: ruby@3a335d7
  namespace: openshift
  selflink: /oapi/v1/namespaces/openshift/imagestreamimages/ruby@3a335d7
```

13.3. KUBERNETES リソースでのイメージストリームの使用

OpenShift Container Platform のネイティブリソースであるイメージストリームは、ビルド または デプロイメント などの OpenShift Container Platform で利用可能な残りのネイティブリソースのすべてと共に追加の設定なしで機能します。現時点で、これらは ジョブ、レプリケーションコントローラー、レプリカセットまたは Kubernetes デプロイメントなどのネイティブ Kubernetes リソースと共に機能することもでき ます。

クラスター管理者は使用可能 なリソースを正確に設定 することができます。

この機能が有効な場合、リソースの **image** フィールドにイメージストリームの参照を配置することができます。この機能を使用する場合、リソースと同じプロジェクトにあるイメージストリームのみを参照することができます。イメージストリームの参照は、単一セグメントの値で構成される必要があります。たとえば **ruby:2 .** 4 の場合、ruby は 2.4 という名前のタグを持ち、参照するリソースと同じプロジェクトにあるイメージストリームの名前になります。

この機能を有効にする 2 つの方法があります。

1. 特定のリソースでイメージストリームの解決を有効にする。これにより、このリソースのみがイメージフィールドのイメージストリーム名を使用できます。

2. イメージストリームでイメージストリームの解決を有効にする。これにより、このイメージストリームを参照するすべてのリソースがイメージフィールドのイメージストリーム名を使用できます。

上記の操作のいずれも **oc set image-lookup** を使用して実行できます。たとえば、以下のコマンドはすべてのリソースが **mysql** という名前のイメージストリームを参照できるようにします。

```
$ oc set image-lookup mysql
```

これにより、**Imagestream.spec.lookupPolicy.local** フィールドが true に設定されます。

イメージルックアップが有効なイメージストリーム

```
apiVersion: v1
kind: ImageStream
metadata:
  annotations:
    openshift.io/display-name: mysql
  name: mysql
  namespace: myproject
spec:
  lookupPolicy:
    local: true
```

有効な場合には、この動作はイメージストリーム内のすべてのタグに対して有効化されます。

以下を使用してイメージストリームをクエリーし、このオプションが設定されているかどうかを確認できます。

```
$ oc set image-lookup
```

さらに、特定のリソースでイメージルックアップを有効にすることもできます。以下のコマンドは **mysql** という名前の Kubernetes デプロイメントがイメージストリームを使用できるようにします。

```
$ oc set image-lookup deploy/mysql
```

これにより、**alpha.image.policy.openshift.io/resolve-names** アノテーションがデプロイメントに設定されます。

イメージルックアップが有効にされたデプロイメント

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: mysql
  namespace: myproject
spec:
  replicas: 1
  template:
    metadata:
      annotations:
        alpha.image.policy.openshift.io/resolve-names: '*'
    spec:
      containers:
      - image: mysql:latest
        imagePullPolicy: Always
        name: mysql
```

イメージルックアップを無効にするには、**--enabled=false** を渡します。

```
$ oc set image-lookup deploy/mysql --enabled=false
```

13.4. イメージプルポリシー

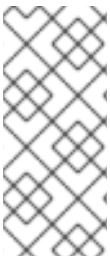
Pod のそれぞれのコンテナーにはコンテナーイメージがあります。イメージを作成し、これをレジストリーにプッシュすると、イメージを Pod で参照できます。

OpenShift Container Platform はコンテナーを作成すると、コンテナーの **imagePullPolicy** を作成して、コンテナーの起動前にイメージをプルする必要があるかどうかを決定します。**imagePullPolicy** には以下の 3 つの値を使用できます。

- **Always**: 常にイメージをプルします。
- **IfNotPresent**: イメージがノード上にない場合にのみイメージをプルします。
- **Never**: イメージをプルしません。

コンテナーの **imagePullPolicy** パラメーターが指定されていない場合、OpenShift Container Platform はイメージのタグに基づいてこれを設定します。

1. タグが **最新** の場合、OpenShift Container Platform は **imagePullPolicy** を **Always** にデフォルト設定します。
2. それ以外の場合に、OpenShift Container Platform は **imagePullPolicy** を **IfNotPresent** にデフォルト設定します。



注記

Never Image Pull Policy を使用する場合、 **AlwaysPullImages** 受付コントローラー を使用してプライベートイメージをプルするための認証情報を持つ Pod のみがそれらのイメージを使用できることを確認できます。この受付コントローラーが有効になっていない場合、イメージの認可検査なしにノード上の任意のユーザーからの Pod がイメージを使用できます。

13.5. 内部レジストリーへのアクセス

イメージのプッシュまたはプルを実行するために OpenShift Container Platform の内部レジストリーに直接アクセスできます。たとえば、これは イメージの手動プッシュによってイメージストリームを作成する場合や、単にイメージに対して **docker pull** を直接実行する場合に役立ちます。

内部レジストリーは OpenShift Container Platform API と同じ トークン を使用して認証します。内部レジストリーに対して **docker login** を実行するには、任意のユーザー名およびメールを使用できますが、パスワードは有効な OpenShift Container Platform トークンである必要があります。

内部レジストリーにログインするには、以下を実行します。

1. OpenShift Container Platform にログインします。

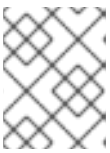
```
$ oc login
```

2. アクセストークンを取得します。

```
$ oc whoami -t
```

3. トークンを使用して内部レジストリーにログインします。**docker** をシステムにインストールしておく必要があります。

```
$ docker login -u <user_name> -e <email_address> \
    -p <token_value> <registry_server>:<port>
```



注記

使用するレジストリー IP またはホスト名およびポートが不明な場合は、クラスター管理者に問い合わせてください。

イメージをプルするには、要求される **imagestreams/layers** に対する **get** 権限が、この認証済みのユーザーに割り当てられている必要があります。また、イメージをプッシュするには、認証済みのユーザーに、要求される **imagestreams/layers** に対する **update** 権限が割り当てられている必要があります。

デフォルトで、プロジェクトのすべてのサービスアカウントは同じプロジェクトの任意のイメージをプルする権限を持ち、**builder** サービスアカウントには同じプロジェクトの任意のイメージをプッシュする権限を持ちます。

13.6. イメージプルシークレットの使用

Docker レジストリー のセキュリティーを保護し、承認されていないユーザーが特定イメージにアクセスできないようにすることができます。OpenShift Container Platform の内部レジストリーを使用し、同じプロジェクトにあるイメージストリームからプルしている場合は、Pod のサービスアカウントに適切なパーミッションがすでに設定されているために追加のアクションは不要です。

ただし、OpenShift Container Platform プロジェクト全体でイメージを参照する場合や、セキュリティー保護されたレジストリーからイメージを参照するなどの他のシナリオでは、追加の設定手順が必要になります。以下のセクションでは、それらのシナリオと必要な手順について詳しく説明します。

13.6.1. Pod が複数のプロジェクト間でのイメージを参照できるようにする設定

内部レジストリーを使用している場合で **project-a** の Pod が **project-b** のイメージを参照できるようにするには、**project-a** のサービスアカウントが **project-b** の **system:image-puller** ロールにバインドされている必要があります。

```
$ oc policy add-role-to-user \
    system:image-puller system:serviceaccount:project-a:default \
    --namespace=project-b
```

このロールを追加した後に、デフォルトのサービスアカウントを参照する **project-a** の Pod は **project-b** からイメージをプルできるようになります。

project-a のすべてのサービスアカウントにアクセスを許可するには、グループを使用します。

```
$ oc policy add-role-to-group \
    system:image-puller system:serviceaccounts:project-a \
    --namespace=project-b
```

13.6.2. Pod による他のセキュアなレジストリーからのイメージの参照を許可する

.dockercfg ファイル (または新規 Docker クライアントの場合は **$HOME/.docker/config.json**) は、ユーザーがセキュア/非セキュアなレジストリーに事前にログインしている場合にそのユーザーの情報を保存する Docker 認証情報ファイルです。

OpenShift Container Platform の内部レジストリーにないセキュリティー保護されたコンテナーイメージをプルするには、Docker 認証情報で **プルシークレット** を作成し、これをサービスアカウントに追加する必要があります。

セキュリティー保護されたレジストリーの **.dockercfg** ファイルがある場合、以下を実行してそのファイルからシークレットを作成できます。

```
$ oc create secret generic <pull_secret_name> \
    --from-file=.dockercfg=<path/to/.dockercfg> \
    --type=kubernetes.io/dockercfg
```

または、**$HOME/.docker/config.json** ファイルがある場合は以下を実行します。

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

セキュアなレジストリーについての Docker 認証情報ファイルがまだない場合には、以下のコマンドを実行してシークレットを作成することができます。

```
$ oc create secret docker-registry <pull_secret_name> \
    --docker-server=<registry_server> \
    --docker-username=<user_name> \
    --docker-password=<password> \
    --docker-email=<email>
```

Pod のイメージをプルするためにシークレットを使用するには、サービスアカウントにシークレットを追加する必要があります。この例では、サービスアカウントの名前は Pod が使用するサービスアカウントの名前に一致している必要があります。 **default** はデフォルトのサービスアカウントです。

```
$ oc secrets link default <pull_secret_name> --for=pull
```

ビルドイメージのプッシュおよびプルにシークレットを使用するには、シークレットは Pod 内でマウント可能でなければなりません。以下でこれを実行できます。

```
$ oc secrets link builder <pull_secret_name>
```

13.6.2.1. 委任された認証を使用したプライベートレジストリーからのプル

プライベートレジストリーは認証を別個のサービスに委任できます。この場合、イメージプルシークレットは認証およびレジストリーのエンドポイントの両方に対して定義されている必要があります。

注記

Red Hat Container Catalog のサードパーティーのイメージは Red Hat Connect Partner Registry (**registry.connect.redhat.com**) から提供されます。このレジストリーは認証を **sso.redhat.com** に委任するため、以下の手順が適用されます。

1. 委任された認証サーバーのシークレットを作成します。

```
$ oc create secret docker-registry \
    --docker-server=sso.redhat.com \
    --docker-username=developer@example.com \
    --docker-password=******** \
    --docker-email=unused \
    redhat-connect-sso

secret/redhat-connect-sso
```

2. プライベートレジストリーのシークレットを作成します。

```
$ oc create secret docker-registry \
    --docker-server=privateregistry.example.com \
    --docker-username=developer@example.com \
    --docker-password=******** \
    --docker-email=unused \
    private-registry

secret/private-registry
```

注記

Red Hat Connect Partner Registry (**registry.connect.redhat.com**) は自動生成される **dockercfg** シークレットタイプを受け入れません (BZ#1476330)。汎用のファイルベースのシークレットは **docker login** コマンドで生成されるファイルを使用して作成する必要があります。

```
$ docker login registry.connect.redhat.com --username developer@example.com

Password: **************
Login Succeeded

$ oc create secret generic redhat-connect --from-
file=.dockerconfigjson=.docker/config.json

$ oc secrets link default redhat-connect --for=pull
```

13.7. タグおよびイメージメタデータのインポート

イメージストリームは、外部 Docker イメージレジストリーのイメージリポジトリーからタグおよびイメージメタデータをインポートするように設定できます。これは複数の異なる方法で実行できます。

- **oc import-image** コマンドで **--from** オプションを使用してタグとイメージ情報を手動でインポートできます。

  ```
  $ oc import-image <image_stream_name>[:<tag>] --from=<docker_image_repo> --confirm
  ```

  以下に例を示します。

  ```
  $ oc import-image my-ruby --from=docker.io/openshift/ruby-20-centos7 --confirm
  The import completed successfully.

  Name:   my-ruby
  Created:  Less than a second ago
  Labels:   <none>
  Annotations:  openshift.io/image.dockerRepositoryCheck=2016-05-06T20:59:30Z
  Docker Pull Spec: 172.30.94.234:5000/demo-project/my-ruby

  Tag Spec     Created   PullSpec       Image
  latest docker.io/openshift/ruby-20-centos7 Less than a second ago docker.io/openshift/ruby-
  20-centos7@sha256:772c5bf9b2d1e8... <same>
  ```

  また、**latest** だけではなくイメージのすべてのタグをインポートするには **--all** フラグを追加することもできます。

- OpenShift Container Platform のほとんどのオブジェクトの場合と同様に、CLI を使用して JSON または YAML 定義を作成し、これをファイルに保存してからオブジェクトを作成できます。**spec.dockerImageRepository** フィールドをイメージの Docker プル仕様に設定します。

  ```
  apiVersion: "v1"
  kind: "ImageStream"
  metadata:
  ```

```
  name: "my-ruby"
spec:
  dockerImageRepository: "docker.io/openshift/ruby-20-centos7"
```

次にオブジェクトを作成します。

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

外部 Docker レジストリーのイメージを参照するイメージストリームを作成する場合、OpenShift Container Platform は短時間で外部レジストリーと通信し、イメージについての最新情報を取得します。

タグおよびイメージメタデータの同期後に、イメージストリームオブジェクトは以下のようになります。

```
apiVersion: v1
kind: ImageStream
metadata:
  name: my-ruby
  namespace: demo-project
  selflink: /oapi/v1/namespaces/demo-project/imagestreams/my-ruby
  uid: 5b9bd745-13d2-11e6-9a86-0ada84b8265d
  resourceVersion: '4699413'
  generation: 2
  creationTimestamp: '2016-05-06T21:34:48Z'
  annotations:
    openshift.io/image.dockerRepositoryCheck: '2016-05-06T21:34:48Z'
spec:
  dockerImageRepository: docker.io/openshift/ruby-20-centos7
  tags:
    -
      name: latest
      annotations: null
      from:
        kind: DockerImage
        name: 'docker.io/openshift/ruby-20-centos7:latest'
      generation: 2
      importPolicy: {  }
status:
  dockerImageRepository: '172.30.94.234:5000/demo-project/my-ruby'
  tags:
    -
      tag: latest
      items:
        -
          created: '2016-05-06T21:34:48Z'
          dockerImageReference: 'docker.io/openshift/ruby-20-
centos7@sha256:772c5bf9b2d1e8e80742ed75aab05820419dc4532fa6d7ad8a1efddda5493dc3'
          image: 'sha256:772c5bf9b2d1e8e80742ed75aab05820419dc4532fa6d7ad8a1efddda5493dc3'
          generation: 2
```

タグおよびイメージメタデータを同期するため、タグをスケジュールに応じて外部レジストリーのクエリーを実行できるよう設定できます。 これは、「タグのイメージストリームへの追加」で説明されているように **--scheduled=true** フラグを **oc tag** コマンドに設定して実行できます。

または、タグの定義で **importPolicy.scheduled** を **true** に設定することもできます。

```
apiVersion: v1
kind: ImageStream
metadata:
  name: ruby
spec:
  tags:
  - from:
      kind: DockerImage
      name: openshift/ruby-20-centos7
    name: latest
    importPolicy:
      scheduled: true
```

13.7.1. 非セキュアなレジストリーからのイメージのインポート

イメージストリームは、自己署名型の証明書を使って署名されたものを使用する場合や、HTTPS ではなく単純な HTTP を使用する場合など、非セキュアなイメージレジストリーからタグおよびイメージメタデータをインポートするように設定できます。

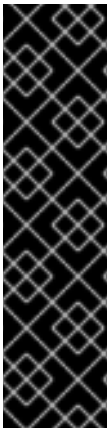
これを設定するには、**openshift.io/image.insecureRepository** アノテーションを追加し、これを **true** に設定します。この設定はレジストリーへの接続時の証明書の検証をバイパスします。

```
kind: ImageStream
apiVersion: v1
metadata:
  name: ruby
  annotations:
    openshift.io/image.insecureRepository: "true" (1)
  spec:
    dockerImageRepository: my.repo.com:5000/myimage
```

(1) **openshift.io/image.insecureRepository** アノテーション **true** に設定します。

重要

このオプションは統合レジストリーに対して、イメージの提供時にイメージストリームでタグ付けされた外部イメージについて非セキュアなトランスポートにフォールバックするよう指示しますが、これにはリスクが伴います。可能な場合には、**istag** にのみ非セキュアのマークを付けてこのリスクを回避します。



重要

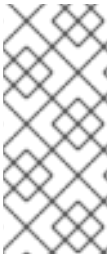
上記の定義はタグおよびイメージメタデータのインポートのみに適用されます。このイメージがクラスターで使用されるようにするには (**docker pull** を実行できるようにするには)、以下のいずれかが該当している必要があります。

1. 各ノードには Docker が **dockerImageRepository** のレジストリーの部分に一致する **--insecure-registry** フラグで設定されている。詳細は、「 Host Preparation 」を参照してください。
2. 各 **istag** 仕様では **referencePolicy.type** が **Local** に設定されている。詳細は、「参照ポリシー」を参照してください。

13.7.1.1. イメージストリームタグのポリシー

13.7.1.1.1. 非セキュアなタグのインポートポリシー

上記のアノテーションは、特定の **ImageStream** のすべてのイメージおよびタグに適用されます。より詳細な制御を実行するために、ポリシーを **istags** に設定できます。タグの定義の **importPolicy.insecure** を **true** に設定すると、このタグ下のイメージについてのみ非セキュアなトランスポートへのフォールバックが許可されます。



注記

特定の **istag** 下のイメージについてのセキュアでないトランスポートへのフォールバックは、イメージストリームにセキュアでないアノテーションが付けられるか、または **istag** にセキュアでないインポートポリシーが設定されている場合に有効になります。**importPolicy.insecure`** が **false** に設定されていると、イメージストリームのアノテーションは上書きできません。

13.7.1.1.2. 参照ポリシー

参照ポリシーにより、このイメージストリームタグを参照するリソースがどこからイメージをプルするかを指定できます。これはリモートイメージ (外部レジストリーからインポートされるもの) にのみ適用されます。**Local** と **Source** のオプションから選択できます。

Source ポリシーはクライアントに対し、イメージのソースレジストリーから直接プルするように指示します。統合レジストリーは、イメージがクラスターによって管理されていない限り使用されません。(これは外部イメージではありません。) これはデフォルトポリシーになります。

Local ポリシーはクライアントに対し、常に統合レジストリーからプルするように指示します。これは Docker デーモンの設定を変更せずに外部の非セキュアなレジストリーからプルする場合に役立ちます。

このポリシーはイメージストリームタグの使用にのみ適用されます。外部レジストリーの場所を使用してイメージを直接参照したり、プルしたりするコンポーネントまたは操作は内部レジストリーにリダイレクトされません。

プルスルー(pull-through)機能

このレジストリーの機能はリモートイメージをクライアントに提供します。この機能はデフォルトで有効にされており、ローカルの参照ポリシーが使用されるようにするには有効にされている必要があります。さらにすべての Blob は後のアクセスを速めるためにミラーリングされます。

イメージストリームタグの仕様でポリシーを **referencePolicy.type** として設定できます。

ローカル参照ポリシーが設定されたセキュアでないタグの例

```
kind: ImageStream
apiVersion: v1
metadata:
  name: ruby
  tags:
  - from:
      kind: DockerImage
      name: my.repo.com:5000/myimage
    name: mytag
    importPolicy:
      insecure: true (1)
    referencePolicy:
      type: Local (2)
```

(1) 該当レジストリーへの非セキュアな接続を使用するようタグ **mytag** を設定します。

(2) 外部イメージをプルするために統合レジストリーを使用するよう **mytag** を設定します。参照ポリシータイプが **Source** に設定されている場合、クライアントはイメージを **my.repo.com:5000/myimage** から直接フェッチします。

13.7.2. プライベートレジストリーからのイメージのインポート

イメージストリームは、プライベートレジストリーからタグおよびイメージメタデータをインポートするように設定できます。 これには認証が必要です。

これを設定するには、認証情報を保存するために使用されるシークレットを作成する必要があります。**oc create secret** コマンドを使用してシークレットを作成する方法については、「Pod が他のセキュアなレジストリーからイメージを参照できるようにする設定」を参照してください。

シークレットが設定されたら、次に新規イメージストリームを作成するか、または **oc import-image** コマンドを使用します。インポートプロセスで OpenShift Container Platform はシークレットを取得してリモートパーティーに提供します。

注記

セキュアでないレジストリーからインポートする場合には、シークレットに定義されたレジストリーの URL に **:80** ポートのサフィックスを追加するようにしてください。 追加していない場合にレジストリーからインポートしようとすると、このシークレットは使用されません。

13.7.3. 外部レジストリーの信頼される証明書の追加

インポート元となっているレジストリーが標準の認証局で署名されていない証明書を使用している場合、レジストリーの証明書または署名する認証局を信頼するようシステムを明示的に設定する必要があります。これは 、レジストリーインポートコントローラーを実行するホストシステム (通常はマスターノード) に CA 証明書またはレジストリー証明書を追加して実行できます。

証明書または CA 証明書は、ホストシステムの **/etc/pki/tls/certs** または **/etc/pki/ca-trust** にそれぞれ追加する必要があります。また証明書の変更を反映するには、**update-ca-trust** コマンドを Red Hat ディストリビューションで実行して、マスターサービスを再起動する必要があります。

13.7.4. 複数のプロジェクト間でのイメージのインポート

イメージストリームは、異なるプロジェクトから内部レジストリーのタグおよびイメージメタデータをインポートするように設定できます。推奨される方法としては、「タグのイメージストリームへの追加」で説明されている **oc tag** コマンドを使用できます。

```
$ oc tag <source_project>/<image_stream>:<tag> <new_image_stream>:<new_tag>
```

別の方法として、プル仕様を使用して他のプロジェクトからイメージを手動でインポートすることもできます。

警告

以下の方法は使用しないことを強く推奨します。 この使用は **oc tag** を使用するだけでは不十分な場合にのみに使用する必要があります。

1. 最初に、他のプロジェクトにアクセスするために必要なポリシーを追加します。

```
$ oc policy add-role-to-group \
    system:image-puller \
    system:serviceaccounts:<destination_project> \
    -n <source_project>
```

   これにより、**<destination_project>** が **<source_project>** からイメージをプルできます。

2. ポリシーが有効な場合、イメージを手動でインポートできます。

```
$ oc import-image <new_image_stream> --confirm \
    --from=<docker_registry>/<source_project>/<image_stream>
```

13.7.5. イメージの手動プッシュによるイメージストリームの作成

イメージストリームはイメージを内部レジストリーに手動でプッシュすると自動的に作成されます。これは OpenShift Container Platform 内部レジストリーを使用している場合にのみ可能です。

この手順を実行する前に、以下の条件を満たしている必要があります。

- プッシュ先となる宛先プロジェクトがすでに存在している必要がある。
- ユーザーはそのプロジェクトで **{get, update} "imagestream/layers"** を実行する権限がある必要があります。さらに、イメージストリームが存在していない場合、ユーザーはそのプロジェクトで **{create} "imagestream"** を実行する権限がなければなりません。プロジェクト管理者にはこれらを実行するパーミッションがあります。

注記

system:image-pusher ロールは新規イメージストリームの作成パーミッションを付与せず、既存イメージストリームにイメージをプッシュするパーミッションのみを付与するため、ユーザーに追加パーミッションが付与されない場合、存在していないイメージストリームにイメージをプッシュするためにこのパーミッションを使用することはできません。

イメージを手動でプッシュしてイメージストリームを作成するには、以下を実行します。

1. まず、内部レジストリーにログインします。
2. 次に、適切な内部レジストリーの場所を使用してイメージにタグを付けます。たとえば、**docker.io/centos:centos7** イメージをローカルにプルしている場合は以下を実行します。

```
$ docker tag docker.io/centos:centos7 172.30.48.125:5000/test/my-image
```

3. 最後に、イメージを内部レジストリーにプッシュします。以下に例を示します。

```
$ docker push 172.30.48.125:5000/test/my-image
The push refers to a repository [172.30.48.125:5000/test/my-image] (len: 1)
c8a648134623: Pushed
2bf4902415e3: Pushed
latest: digest:
sha256:be8bc4068b2f60cf274fc216e4caba6aa845fff5fa29139e6e7497bb57e48d67 size:
6273
```

4. イメージストリームが作成されていることを確認します。

```
$ oc get is
NAME       DOCKER REPO                       TAGS      UPDATED
my-image   172.30.48.125:5000/test/my-image   latest    3 seconds ago
```

13.8. イメージストリーム変更時の更新のトリガー

イメージストリームタグが新規イメージを参照するように更新される場合、OpenShift Container Platform は、古いイメージを使用していたリソースに新規イメージをロールアウトするためのアクションを自動的に実行します。イメージストリームタグを参照しているリソースのタイプに応じ、この設定はさまざまな方法で実行できます。

13.8.1. OpenShift リソース

OpenShift DeploymentConfigs および BuildConfigs は ImageStreamTags への変更によって自動的にトリガーされます。トリガーされたアクションは更新された ImageStreamTag で参照されるイメージの新規の値を使用して実行されます。この機能の使用方法についての詳細は、BuildConfig トリガーおよび DeploymentConfig トリガーについての説明を参照してください。

13.8.2. Kubernetes リソース

API 定義の一部としてトリガーを制御するためのフィールドセットを含む DeploymentConfigs および BuildConfigs とは異なり、Kubernetes リソースにはトリガー用のフィールドがありません。その代わりに、OpenShift Container Platform はアノテーションを使用してユーザーがトリガーを要求できるようにします。アノテーションは以下のように定義されます。

```
Key: image.openshift.io/triggers
Value: array of triggers, where each item has the schema:
[
 {
   "from" :{
     "kind": "ImageStreamTag", // required, the resource to trigger from, must be ImageStreamTag
     "name": "example:latest", // required, the name of an ImageStreamTag
     "namespace": "myapp",     // optional, defaults to the namespace of the object
   },
   // required, JSON path to change
   // Note that this field is limited today, and only accepts a very specific set
   // of inputs (a JSON path expression that precisely matches a container by ID or index).
   // For pods this would be "spec.containers[?(@.name='web')].image".
   "fieldPath": "spec.template.spec.containers[?(@.name='web')].image",
   // optional, set to true to temporarily disable this trigger.
   "paused": "false"
 },
 ...
]
```

OpenShift Container Platform が Pod テンプレート (CronJobs、Deployments、StatefulSets、DaemonSets、Jobs、ReplicaSets、ReplicationControllers、および Pods のみ) とこのアノテーションの両方が指定されたコアの Kubernetes リソースを検出すると、トリガーが参照する ImageStreamTag に関連付けられているイメージを使用してオブジェクトの更新を試行します。この更新は、指定の **fieldPath** に対して実行されます。

以下の例では、トリガーは **example:latest** イメージストリームタグの更新時に実行されます。実行時に、オブジェクトの Pod テンプレートにある **web** コンテナーへのイメージ参照が、新しいイメージの値に更新されます。Pod テンプレートがデプロイメント定義の一部である場合には、Pod テンプレートへの変更はデプロイメントを自動的にトリガーされて、新規イメージがロールアウトされます。

```
image.openshift.io/triggers=[{"from":
{"kind":"ImageStreamTag","name":"example:latest"},"fieldPath":"spec.template.spec.containers[?
(@.name='web')].image"}]
```

イメージトリガーをデプロイメントに追加する時に、**oc set triggers** コマンドも使用できます。たとえば、以下のコマンドは **example** という名前のデプロイメントにイメージ変更トリガーを追加し、**example:latest** イメージストリームタグが更新されるとデプロイメント内の **web** コンテナーがイメージの新規の値で更新されます。

```
$ oc set triggers deploy/example --from-image=example:latest -c web
```

デプロイメントが一時停止されない限り、この Pod テンプレートの更新により、デプロイメントはイメージの新規の値で自動的に実行されます。

13.9. イメージストリーム定義の記述

イメージストリーム全体に対するイメージストリームの定義を記述して、複数のイメージストリームを定義できます。これにより、**oc** コマンドを実行せずに異なるクラスターに定義を配信することができます。

イメージストリームの定義は、イメージストリームやインポートする固有のタグに関する情報を指定します。

イメージストリームオブジェクトの定義

```
apiVersion: v1
kind: ImageStream
metadata:
  name: ruby
  annotations:
    openshift.io/display-name: Ruby (1)
spec:
  tags:
    - name: '2.0' (2)
      annotations:
        openshift.io/display-name: Ruby 2.0 (3)
        description: >- (4)
          Build and run Ruby 2.0 applications on CentOS 7. For more information
          about using this builder image, including OpenShift considerations,
          see
          https://github.com/sclorg/s2i-ruby-container/tree/master/2.0/README.md.
        iconClass: icon-ruby (5)
        sampleRepo: 'https://github.com/sclorg/ruby-ex.git' (6)
        tags: 'builder,ruby' (7)
        supports: 'ruby' (8)
        version: '2.0' (9)
      from:
        kind: DockerImage (10)
        name: 'docker.io/openshift/ruby-20-centos7:latest' (11)
```

(1) イメージストリーム全体での簡単でユーザーフレンドリーな名前。

(2) タグはバージョンとして参照されます。タグはドロップダウンメニューに表示されます。

(3) イメージストリーム内のこのタグのユーザーフレンドリーな名前です。これは簡単で、バージョン情報が含まれている必要があります (該当する場合)。

(4) タグの説明。これにはユーザーがイメージの提供内容を把握できる程度の詳細情報が含まれます。これには追加の説明へのリンクを含めることができます。説明をいくつかの文に制限します。

(5) このタグの表示されるアイコン。可能な場合は既存のロゴアイコンから選択します。FontAwesome および Patternfly のアイコンも使用できます。または、イメージストリームを使用する OpenShift Container Platform クラスターに CSS カスタマイズ を追加できるので、CSS カスタマイズ経由でアイコンを提供します。存在するアイコンクラスを指定する必要があります。これを指定しないと、汎用アイコンへのフォールバックが禁止されます。

(6) このイメージストリームタグをビルダーイメージタグとして使用してビルドでき、サンプルアプリケーションを実行するために使用されるソースリポジトリーの URL です。

(7) イメージストリームタグが関連付けられるカテゴリーです。このタグがカタログに表示されるには、ビルダータグが必要です。これを提供されているカタログカテゴリーのいずれかに関連付けるタグを追加します。コンソールの定数ファイルの **CATALOG_CATEGORIES** で **id** および **categoryAliases** を参照してください。カテゴリーはクラスター全体に対して カスタマイズ することもできます。

(8) このイメージがサポートする言語。この値は builder イメージを指定されるソースリポジトリーに一致させるように **oc new-app** の起動時に使用されます。

9 このタグのバージョン情報。

10 このイメージストリームタグが参照するオブジェクトのタイプ。有効な値は **DockerImage**、**ImageStreamTag** および **ImageStreamImage** です。

11 このイメージストリームタグがインポートするオブジェクト。

ImageStream に定義できるフィールドについての詳細は、「Imagestream API」および「ImagestreamTag API」を参照してください。

第14章 クォータおよび制限範囲

14.1. 概要

クォータおよび制限範囲を使用して、クラスター管理者はプロジェクトで使用されるオブジェクトの数やコンピュートリソースの量を制限するための制約を設定することができます。これは、管理者がすべてのプロジェクトでリソースの効果的な管理および割り当てを実行し、いずれのプロジェクトでの使用量がクラスターサイズに対して適切な量を超えることのないようにするのに役立ちます。

開発者は Pod およびコンテナーのレベルでコンピュートリソースの要求および制限を設定することもできます。

以下のセクションは、クォータおよび制限範囲の設定を確認し、それらの制約対象や、独自の Pod およびコンテナーでコンピュートリソースを要求し、制限する方法について理解するのに役立ちます。

14.2. クォータ

ResourceQuota オブジェクトで定義されるリソースクォータは、プロジェクトごとにリソース消費量の総計を制限する制約を指定します。これは、タイプ別にプロジェクトで作成できるオブジェクトの数量を制限すると共に、そのプロジェクトのリソースが消費できるコンピュートリソースおよびストレージの合計量を制限することができます。

注記

クォータはクラスター管理者によって設定され、所定プロジェクトにスコープが設定されます。

14.2.1. クォータの表示

web コンソールでプロジェクトの **Quota** ページに移動し、プロジェクトのクォータで定義されるハード制限に関連する使用状況の統計を表示できます。

CLI を使用してクォータの詳細を表示することもできます。

1. 最初に、プロジェクトで定義されたクォータの一覧を取得します。たとえば、**demoproject** というプロジェクトの場合は以下のようになります。

```
$ oc get quota -n demoproject
NAME                 AGE
besteffort           11m
compute-resources    2m
core-object-counts   29m
```

2. 次に、関心のあるクォータについて記述します。たとえば、**core-object-counts** クォータの場合、以下を実行します。

```
$ oc describe quota core-object-counts -n demoproject
Name:   core-object-counts
Namespace:  demoproject
Resource  Used Hard
--------  ---- ----
configmaps  3 10
persistentvolumeclaims 0 4
```

```
replicationcontrollers 3 20
secrets   9 10
services  2 10
```

詳細のクォータ定義は、オブジェクトで **oc export** を実行して表示できます。以下は、クォータ定義のサンプルを示しています。

core-object-counts.yaml

```
apiVersion: v1
kind: ResourceQuota
metadata:
  name: core-object-counts
spec:
  hard:
    configmaps: "10" 1
    persistentvolumeclaims: "4" 2
    replicationcontrollers: "20" 3
    secrets: "10" 4
    services: "10" 5
```

1. プロジェクトに存在できる **ConfigMap** オブジェクトの合計数です。
2. プロジェクトに存在できる Persistent Volume Claim (永続ボリューム要求、PVC) の合計数です。
3. プロジェクトに存在できるレプリケーションコントローラーの合計数です。
4. プロジェクトに存在できるシークレットの合計数です。
5. プロジェクトに存在できるサービスの合計数です。

openshift-object-counts.yaml

```
apiVersion: v1
kind: ResourceQuota
metadata:
  name: openshift-object-counts
spec:
  hard:
    openshift.io/imagestreams: "10" 1
```

1. プロジェクトに存在できるイメージストリームの合計数です。

compute-resources.yaml

```
apiVersion: v1
kind: ResourceQuota
metadata:
  name: compute-resources
spec:
  hard:
    pods: "4" 1
```

```
    requests.cpu: "1" 2
    requests.memory: 1Gi 3
    limits.cpu: "2" 4
    limits.memory: 2Gi 5
```

1 プロジェクトに存在できる非終了状態の Pod の合計数です。

2 非終了状態のすべての Pod において、CPU 要求の合計は 1 コアを超えることができません。

3 非終了状態のすべての Pod において、メモリー要求の合計は 1 Gi を超えることができません。

4 非終了状態のすべての Pod において、CPU 制限の合計は 2 コアを超えることができません。

5 非終了状態のすべての Pod において、メモリー制限の合計は 2 Gi を超えることができません。

besteffort.yaml

```
apiVersion: v1
kind: ResourceQuota
metadata:
  name: besteffort
spec:
  hard:
    pods: "1" 1
  scopes:
  - BestEffort 2
```

1 プロジェクトに存在できる QoS (Quality of Service) が **BestEffort** の非終了状態の Pod の合計数です

2 クォータを、メモリーまたは CPU のいずれかの QoS (Quality of Service) が **BestEffort** の一致する Pod のみに制限します。

compute-resources-long-running.yaml

```
apiVersion: v1
kind: ResourceQuota
metadata:
  name: compute-resources-long-running
spec:
  hard:
    pods: "4" 1
    limits.cpu: "4" 2
    limits.memory: "2Gi" 3
  scopes:
  - NotTerminating 4
```

1 非終了状態の Pod の合計数です。

2 非終了状態のすべての Pod において、CPU 制限の合計はこの値を超えることができません。

3 非終了状態のすべての Pod において、メモリー制限の合計はこの値を超えることができません。

4 クォータを**spec.activeDeadlineSeconds** が **nil** に設定されている一致する Pod のみに制限します。ビルド Pod は、**RestartNever** ポリシーが適用されない場合に **NotTerminating** になります。

compute-resources-time-bound.yaml

```
apiVersion: v1
kind: ResourceQuota
metadata:
  name: compute-resources-time-bound
spec:
  hard:
    pods: "2" 1
    limits.cpu: "1" 2
    limits.memory: "1Gi" 3
  scopes:
  - Terminating 4
```

1 非終了状態の Pod の合計数です。

2 非終了状態のすべての Pod において、CPU 制限の合計はこの値を超えることができません。

3 非終了状態のすべての Pod において、メモリー制限の合計はこの値を超えることができません。

4 クォータを**spec.activeDeadlineSeconds >=0** に設定されている一致する Pod のみに制限します。たとえば、このクォータはビルド Pod またはデプロイヤー Pod に影響を与えますが、web サーバーまたはデータベースなどの長時間実行されない Pod には影響を与えません。

storage-consumption.yaml

```
apiVersion: v1
kind: ResourceQuota
metadata:
  name: storage-consumption
spec:
  hard:
    persistentvolumeclaims: "10" 1
    requests.storage: "50Gi" 2
    gold.storageclass.storage.k8s.io/requests.storage: "10Gi" 3
    silver.storageclass.storage.k8s.io/requests.storage: "20Gi" 4
    silver.storageclass.storage.k8s.io/persistentvolumeclaims: "5" 5
    bronze.storageclass.storage.k8s.io/requests.storage: "0" 6
    bronze.storageclass.storage.k8s.io/persistentvolumeclaims: "0" 7
```

1 プロジェクト内の永続ボリューム要求 (PVC) の合計数です。

2 プロジェクトのすべての永続ボリューム要求 (PVC) において、要求されるストレージの合計はこの値を超えることができません。

3 プロジェクトのすべての永続ボリューム要求 (PVC) において、gold ストレージクラスで要求されるストレージの合計はこの値を超えることができません。

4 プロジェクトのすべての永続ボリューム要求 (PVC) において、silver ストレージクラスで要求されるストレージの合計はこの値を超えることができません。

るストレージの合計はこの値を超えることができません。

5 プロジェクトのすべての永続ボリューム要求 (PVC) において、silver ストレージクラスの要求の合計数はこの値を超えることができません。

6 プロジェクトのすべての永続ボリューム要求 (PVC) において、bronze ストレージクラスで要求されるストレージの合計はこの値を超えることができません。これが **0** に設定される場合、bronze ストレージクラスはストレージを要求できないことを意味します。

7 プロジェクトのすべての永続ボリューム要求 (PVC) において、bronze ストレージクラスで要求されるストレージの合計はこの値を超えることができません。これが **0** に設定される場合は、bronze ストレージクラスでは要求を作成できないことを意味します。

14.2.2. クォータで管理されるリソース

以下では、クォータで管理できる一連のコンピュートリソースとオブジェクトタイプについて説明します。

注記

status.phase in (Failed, Succeeded) が true の場合、Pod は終了状態にあります。

表14.1 クォータで管理されるコンピュートリソース

リソース名	説明
cpu	非終了状態のすべての Pod での CPU 要求の合計はこの値を超えることができません。**cpu** および **requests.cpu** は同じ値であり、相互に置き換え可能なものとして使用できます。
memory	非終了状態のすべての Pod でのメモリー要求の合計はこの値を超えることができません。**memory** および **requests.memory** は同じ値であり、相互に置き換え可能なものとして使用できます。
requests.cpu	非終了状態のすべての Pod での CPU 要求の合計はこの値を超えることができません。**cpu** および **requests.cpu** は同じ値であり、相互に置き換え可能なものとして使用できます。
requests.memory	非終了状態のすべての Pod でのメモリー要求の合計はこの値を超えることができません。**memory** および **requests.memory** は同じ値であり、相互に置き換え可能なものとして使用できます。
limits.cpu	非終了状態のすべての Pod での CPU 制限の合計はこの値を超えることができません。
limits.memory	非終了状態のすべての Pod でのメモリー制限の合計はこの値を超えることができません。

表14.2 クォータで管理されるストレージリソース

リソース名	説明
requests.storage	任意の状態のすべての永続ボリューム要求 (PVC) でのストレージ要求の合計は、この値を超えることができません。
persistentvolumeclaims	プロジェクトに存在できる永続ボリューム要求 (PVC) の合計数です。
<storage-class-name>.storageclass.storage.k8s.io/requests.storage	一致するストレージクラスを持つ、任意の状態のすべての永続ボリューム要求 (PVC) でのストレージ要求の合計はこの値を超えることができません。
<storage-class-name>.storageclass.storage.k8s.io/persistentvolumeclaims	プロジェクトに存在できる、一致するストレージクラスを持つ Persistent Volume Claim (永続ボリューム要求、PVC) の合計数です。

表14.3 クォータで管理されるオブジェクト数

リソース名	説明
pods	プロジェクトに存在できる非終了状態の Pod の合計数です。
replicationcontrollers	プロジェクトに存在できるレプリケーションコントローラーの合計数です。
resourcequotas	プロジェクトに存在できるリソースクォータの合計数です。
services	プロジェクトに存在できるサービスの合計数です。
secrets	プロジェクトに存在できるシークレットの合計数です。
configmaps	プロジェクトに存在できる **ConfigMap** オブジェクトの合計数です。
persistentvolumeclaims	プロジェクトに存在できる永続ボリューム要求 (PVC) の合計数です。
openshift.io/imagestreams	プロジェクトに存在できるイメージストリームの合計数です。

14.2.3. クォータのスコープ

各クォータには **スコープ** のセットが関連付けられます。クォータは、列挙されたスコープの交差部分に一致する場合にのみリソースの使用状況を測定します。

スコープをクォータに追加すると、クォータが適用されるリソースのセットを制限できます。許可されるセット以外のリソースを設定すると、検証エラーが発生します。

スコープ	説明
Terminating	**spec.activeDeadlineSeconds >= 0** の Pod に一致します。
NotTerminating	**spec.activeDeadlineSeconds** が **nil** の Pod に一致します。
BestEffort	**cpu** または **memory** のいずれかの QoS (Quality of Service) が Best Effort の Pod に一致します。コンピュートリソースのコミットについての詳細は、「QoS(Quality of Service)クラス」を参照してください。
NotBestEffort	**cpu** および **memory** の QoS (Quality of Service) が Best Effort でない Pod に一致します。

BestEffort スコープは、以下のリソースを制限するようにクォータを制限します。

- **pods**

Terminating、NotTerminating、および NotBestEffort スコープは、以下のリソースを追跡するようにクォータを制限します。

- **pods**
- **memory**
- **requests.memory**
- **limits.memory**
- **cpu**
- **requests.cpu**
- **limits.cpu**

14.2.4. クォータの実施

プロジェクトのリソースクォータが最初に作成されると、プロジェクトは、更新された使用状況の統計が計算されるまでクォータ制約の違反を引き起こす可能性のある新規リソースの作成機能を制限します。

クォータが作成され、使用状況の統計が更新されると、プロジェクトは新規コンテンツの作成を許可します。リソースを作成または変更する場合、クォータの使用量はリソースの作成または変更要求があるとすぐに増分します。

リソースを削除する場合、クォータの使用量は、プロジェクトのクォータ統計の次回の完全な再計算時に減分されます。プロジェクトの変更がクォータの使用制限を超える場合、サーバーはアクションを拒否します。クォータ制約を違反していること、およびシステムで現在確認される使用量の統計値を示す適切なエラーメッセージが返されます。

14.2.5. 要求 vs 制限

コンピュートリソースの割り当て時に、各コンテナーは CPU およびメモリーの要求値と制限値を指定できます。クォータはこれらの値のいずれも制限できます。

クォータに **requests.cpu** または **requests.memory** の値が指定されている場合、すべての着信コンテナーがそれらのリソースを明示的に要求することが求められます。クォータに **limits.cpu** または **limits.memory** の値が指定されている場合、すべての着信コンテナーがそれらのリソースの明示的な制限を指定することが求められます。

Pod およびコンテナーに要求および制限を設定する方法についての詳細は、「コンピュートリソース」を参照してください。

14.3. 制限範囲

LimitRange オブジェクトで定義される制限範囲は、Pod、コンテナー、イメージ、イメージストリーム、および Persistent Volume Claim (永続ボリューム要求、PVC) のレベルでプロジェクトのコンピュートリソース制約を列挙し、Pod、コンテナー、イメージ、イメージストリームまたは Persistent Volume Claim (永続ボリューム要求、PVC) で消費できるリソースの量を指定します。

すべてのリソース作成および変更要求は、プロジェクトの各 **LimitRange** オブジェクトに対して評価されます。リソースが列挙される制約に違反する場合、そのリソースは拒否されます。リソースが明示的な値を指定しない場合で、制約がデフォルト値をサポートする場合は、デフォルト値がリソースに適用されます。

注記

制限範囲はクラスター管理者によって設定され、所定プロジェクトにスコープが設定されます。

14.3.1. 制限範囲の表示

web コンソールでプロジェクトの **Quota** ページに移動し、プロジェクトで定義される制限範囲を表示できます。

CLI を使用して制限範囲の詳細を表示することもできます。

1. まず、プロジェクトで定義される制限範囲の一覧を取得します。たとえば、**demoproject** というプロジェクトの場合は以下のようになります。

```
$ oc get limits -n demoproject
NAME              AGE
resource-limits   6d
```

2. 次に、関連のある制限範囲の説明を表示します。 たとえば、**resource-limits** 制限範囲の場合は以下のようになります。

```
$ oc describe limits resource-limits -n demoproject
Name:                   resource-limits
Namespace:              demoproject
Type                    Resource              Min     Max     Default Request Default Limit   Max
Limit/Request Ratio
----                    --------              ---     ---     --------------- -------------   -----------------------
Pod                     cpu                   200m    2       -               -               -
Pod                     memory                6Mi     1Gi     -               -               -
Container               cpu                   100m    2       200m            300m            10
Container               memory                4Mi     1Gi     100Mi           200Mi           -
openshift.io/Image      storage               -       1Gi     -               -               -
openshift.io/ImageStream    openshift.io/image      -       12      -               -               -
openshift.io/ImageStream    openshift.io/image-tags -       10      -               -               -
```

詳細の制限範囲の定義は、オブジェクトで **oc export** を実行して表示できます。以下は、制限範囲の定義例を示しています。

コア Limit Range オブジェクトの定義

```
apiVersion: "v1"
kind: "LimitRange"
metadata:
  name: "core-resource-limits" (1)
spec:
  limits:
    - type: "Pod"
      max:
        cpu: "2" (2)
        memory: "1Gi" (3)
      min:
        cpu: "200m" (4)
        memory: "6Mi" (5)
    - type: "Container"
      max:
        cpu: "2" (6)
        memory: "1Gi" (7)
      min:
        cpu: "100m" (8)
        memory: "4Mi" (9)
      default:
        cpu: "300m" (10)
        memory: "200Mi" (11)
      defaultRequest:
        cpu: "200m" (12)
        memory: "100Mi" (13)
      maxLimitRequestRatio:
        cpu: "10" (14)
```

(1) 制限範囲オブジェクトの名前です。

(2) すべてのコンテナーにおいて Pod がノードで要求できる CPU の最大量です。

(3) すべてのコンテナーにおいて Pod がノードで要求できるメモリーの最大量です。

(4) すべてのコンテナーにおいて Pod がノードで要求できる CPU の最小量です。

(5) すべてのコンテナーにおいて Pod がノードで要求できるメモリーの最小量です。

(6) Pod の単一コンテナーが要求できる CPU の最大量です。

(7) Pod の単一コンテナーが要求できるメモリーの最大量です。

(8) Pod の単一コンテナーが要求できる CPU の最小量です。

(9) Pod の単一コンテナーが要求できるメモリーの最小量です。

(10) 指定がない場合、コンテナーによる使用を制限する CPU のデフォルト量です。

11 コンテナーによる使用を制限するメモリーのデフォルト量です (指定がない場合)。

12 コンテナーが使用を要求する CPU のデフォルト量です (指定がない場合)。

13 コンテナーが使用を要求するメモリーのデフォルト量です (指定がない場合)。

14 制限の要求に対する比率でコンテナーで実行できる CPU バーストの最大量です。

CPU およびメモリーの測定方法についての詳細は、「コンピュートリソース」を参照してください。

14.3.2. コンテナーの制限

サポートされるリソース:

- CPU
- メモリー

サポートされる制約:

コンテナーごとに設定されます。 指定される場合、以下を満たしている必要があります。

表14.4 コンテナー

制約	動作
Min	**Min[resource]**: **container.resources.requests[resource]** (必須) または **container/resources.limits[resource]** (オプション) 以下 設定で **min** CPU を定義している場合、要求値はその CPU 値よりも大きくなければなりません。制限値を指定する必要はありません。
Max	**container.resources.limits[resource]** (必須): **Max[resource]** 以下 設定で **max** CPU を定義している場合、要求値を定義する必要はありませんが、CPU 制約の最大値の条件を満たす制限値を設定する必要があります。
MaxLimitRequestRatio	**MaxLimitRequestRatio[resource]**: (**container.resources.limits[resource]** / **container.resources.requests[resource]**) 以下 設定で **maxLimitRequestRatio** 値を定義している場合に、新規コンテナーには要求値および制限値の両方が必要になります。さらに OpenShift Container Platform は、制限を要求で除算して制限の要求に対する比率を算出します。この値は、1 より大きい正の整数でなければなりません。 たとえば、コンテナーの**limit** 値が **cpu: 500** で、**request** 値が **cpu: 100** である場合、**cpu** の要求に対する制限の比は **5** になります。この比率は **maxLimitRequestRatio** より小さいか等しくなければなりません。

サポートされるデフォルト:

Default[resource]
指定がない場合は **container.resources.limit[resource]** を所定の値にデフォルト設定します。

Default Requests[resource]

指定がない場合は、**container.resources.requests[resource]** を所定の値にデフォルト設定します。

14.3.3. Pod の制限

サポートされるリソース:

- CPU
- メモリー

サポートされる制約:

Pod のすべてのコンテナーにおいて、以下を満たしている必要があります。

表14.5 Pod

制約	実施される動作
Min	**Min[resource]**: **container.resources.requests[resource]** (必須) または **container.resources.limits[resource]** (オプション) 以下
Max	**container.resources.limits[resource]** (必須): **Max[resource]** 以下
MaxLimitRequestRatio	**MaxLimitRequestRatio[resource]**: (**container.resources.limits[resource]** / **container.resources.requests[resource]**) 以下

14.4. コンピュートリソース

ノードで実行される各コンテナーはコンピュートリソースを消費します。 コンピュートリソースは要求し、割り当て、消費できる数量を測定できるリソースです。

Pod 設定ファイルの作成時に、クラスターでの Pod のスケジュールを効果的に実行し、適切なパフォーマンスを確保できるように各コンテナーが必要とする CPU およびメモリー (RAM) の量をオプションで指定できます。

CPU は millicore という単位で測定されます。クラスター内の各ノードはオペレーティングシステムを検査して、ノード上の CPU コアの量を判別し、その値を 1000 で乗算して合計容量を表します。たとえば、ノードに 2 コアある場合に、ノードの CPU 容量は 2000m として表されます。単一コアの 1/10 を使用する場合には、100m として表されます。

メモリーはバイト単位で測定されます。さらに、これには SI サフィックス (E、P、T、G、M、 K) または、相当する 2 のべき乗の値 (Ei、Pi、Ti、Gi、Mi、Ki) を指定できます。

```
apiVersion: v1
kind: Pod
spec:
  containers:
  - image: openshift/hello-openshift
    name: hello-openshift
    resources:
```

```
    requests:
      cpu: 100m 1
      memory: 200Mi 2
    limits:
      cpu: 200m 3
      memory: 400Mi 4
```

1 コンテナーは 100m CPU を要求します。

2 コンテナーは 200Mi メモリーを要求します。

3 コンテナーは 200m CPU の制限を設定します。

4 コンテナーは 400Mi メモリーの制限を設定します。

14.4.1. CPU 要求

Pod の各コンテナーはノードで要求する CPU の量を指定できます。スケジューラーは CPU 要求を使用してコンテナーに適したノードを検索します。

CPU 要求はコンテナーが消費できる CPU の最小量を表しますが、CPU の競合がない場合、ノード上の利用可能なすべての CPU を使用できます。ノードに CPU の競合がある場合、CPU 要求はシステム上のすべてのコンテナーに対し、コンテナーで使用可能な CPU 時間についての相対的な重みを指定します。

ノード上でこの動作を実施するために CPU 要求がカーネル CFS 共有にマップされます。

14.4.2. コンピュートリソースの表示

Pod のコンピュートリソースを表示するには、以下を実行します。

```
$ oc describe pod ruby-hello-world-tfjxt
Name:       ruby-hello-world-tfjxt
Namespace:      default
Image(s):     ruby-hello-world
Node:       /
Labels:       run=ruby-hello-world
Status:       Pending
Reason:
Message:
IP:
Replication Controllers:  ruby-hello-world (1/1 replicas created)
Containers:
  ruby-hello-world:
    Container ID:
    Image ID:
    Image:    ruby-hello-world
    QoS Tier:
      cpu:  Burstable
      memory: Burstable
    Limits:
      cpu:  200m
      memory: 400Mi
    Requests:
```

```
    cpu:    100m
    memory:   200Mi
  State:    Waiting
  Ready:    False
  Restart Count:  0
  Environment Variables:
```

14.4.3. CPU 制限

Pod の各コンテナーはノードで使用を制限する CPU 量を指定できます。CPU 制限はコンテナーがノードの競合の有無とは関係なく使用できる CPU の最大量を制御します。コンテナーが指定の制限を以上を使用しようとする場合には、システムによりコンテナーの使用量が調節されます。これにより、コンテナーがノードにスケジュールされる Pod 数とは関係なく一貫したサービスレベルを維持することができます。

14.4.4. メモリー要求

デフォルトで、コンテナーはノード上の可能な限り多くのメモリーを使用できます。クラスター内での Pod の配置を改善するには、コンテナーの実行に必要なメモリーの量を指定します。スケジューラーは Pod をノードにバインドする前にノードの利用可能なメモリー容量を考慮に入れます。コンテナーは、要求を指定する場合も依然として可能な限り多くのメモリーを消費することができます。

14.4.5. メモリー制限

メモリー制限を指定する場合、コンテナーが使用できるメモリーの量を制限できます。たとえば 200Mi の制限を指定する場合、コンテナーの使用はノード上のそのメモリー量に制限されます。コンテナーが指定されるメモリー制限を超える場合、コンテナーは終了します。 その後はコンテナーの再起動ポリシーによって再起動する可能性もあります。

14.4.6. QoS (Quality of Service) 層

作成後、コンピュートリソースは **quality of service** (QoS) で分類されます。これは、リソースごとに指定される要求および制限値に基づいて 3 つの層に分類されます。

QoS (Quality of Service)	説明
BestEffort	要求および制限が指定されない場合に指定されます。
Burstable	要求がオプションで指定される制限よりも小さい値に指定される場合に指定されます。
Guaranteed	制限がオプションで指定される要求に等しい値に指定される場合に指定されます。

コンテナーに各コンピュートリソースに異なる QoS を生じさせる要求および制限セットが含まれる場合、これは **Burstable** として分類されます。

QoS は、リソースが圧縮可能であるかどうかによって各種のリソースにそれぞれ異なる影響を与えます。CPU は圧縮可能なリソースですが、メモリーは圧縮できないリソースです。

CPU リソースの場合:

- **BestEffort CPU** コンテナーはノードで利用可能な CPU を消費できますが、最も低い優先順位で実行されます。
- **Burstable CPU** コンテナーは要求される CPU の最小量を取得することが保証されますが、追加の CPU 時間を取得できる場合もあればできない場合もあります。追加の CPU リソースはノード上のすべてのコンテナーで要求される量に基づいて分配されます。
- **Guaranteed CPU** コンテナーは、追加の CPU サイクルが利用可能な場合でも要求される量のみを取得することが保証されます。これにより、ノード上の他のアクティビティーとは関係なく一定のパフォーマンスレベルを確保できます。

メモリーリソースの場合:

- **BestEffort メモリー** コンテナーはノード上で利用可能なメモリーを消費できますが、スケジューラーがコンテナーを、その必要を満たすのに十分なメモリーを持つノードに配置する保証はありません。さらにノードで OOM (Out of Memory) イベントが発生する場合、**BestEffort** コンテナーが強制終了される可能性が最も高くなります。
- **Burstable メモリー** コンテナーは要求されるメモリー量を取得できるようノードにスケジュールされますが、それ以上の量を消費する可能性があります。ノード上で OOM イベントが発生する場合、**Burstable** コンテナーはメモリー回復の試行時に **BestEffort** コンテナーの次に強制終了されます。
- **Guaranteed メモリー** コンテナーは、要求されるメモリー量のみを取得します。OOM イベントが発生する場合は、システム上に他の **BestEffort** または **Burstable** コンテナーがない場合にのみ強制終了されます。

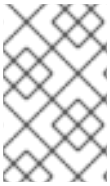
14.4.7. CLI でのコンピュートリソースの指定

CLI でコンピュートリソースを指定するには、以下を実行します。

```
$ oc run ruby-hello-world --image=ruby-hello-world --limits=cpu=200m,memory=400Mi --
requests=cpu=100m,memory=200Mi
```

14.4.8. 不透明な整数リソース

不透明な整数リソースは、クラスターのオペレーターがシステムで認識されない新規のノードレベルのリソースを提供することを可能にします。ユーザーは CPU やメモリーと同様に Pod 仕様にあるこれらのリソースを消費できます。スケジューラーは、利用可能な量を上回るリソースが複数の Pod に同時に割り当てられないようにリソースアカウンティングを実行します。



注記

不透明な整数リソースは現時点でアルファ機能であり、リソースアカウンティングのみが実装されています。これらのリソースについてのリソースクォータや制限範囲のサポートはなく、これらが QoS に影響を与えることはありません。

不透明な整数リソースが **不透明 (opaque)** と言われるのは、OpenShift Container Platform がリソースが何を認識しない状態でも、そのリソースが十分にある場合に Pod をノードにスケジュールするためです。それらが **整数リソース** と言われるのは、それらが整数で表される量で利用可能であるか、または **公開** されるためです。API サーバーはこれらのリソースの量を整数に制限します。**有効な** 量の例は、**3**、**3000m**、および **3Ki** などです。

通常はクラスター管理者がリソースを作成し、それらを利用可能にします。不透明な整数リソースの作成についての詳細は、『管理者ガイド』の「 不透明な 整数リソース」を参照してください。

Pod の不透明な整数リソースを消費するには、不透明なリソースの名前を **spec.containers[].resources.requests** フィールドにキーとして含めるように Pod を編集します。

例: 以下の Pod は 2 つの CPU および 1 つの **foo** (不透明なリソース) を要求しています。

```
apiVersion: v1
kind: Pod
metadata:
  name: my-pod
spec:
  containers:
  - name: my-container
    image: myimage
    resources:
      requests:
        cpu: 2
        pod.alpha.kubernetes.io/opaque-int-resource-foo: 1
```

Pod は、(CPU、メモリー、およびすべての不透明なリソースを含む) リソース要求のすべてが満たされる場合にのみスケジュールされます。Pod は、リソース要求がいずれのノードでも満たされない場合には **PENDING** 状態のままになります。

```
Conditions:
  Type    Status
  PodScheduled  False
...
Events:
  FirstSeen  LastSeen Count From    SubObjectPath Type   Reason     Message
  ---------  -------- ----- ----    ------------- -------- ------     -------
  14s      0s  6 default-scheduler  Warning   FailedScheduling  No nodes are available that match all of the following predicates:: Insufficient pod.alpha.kubernetes.io/opaque-int-resource-foo (1).
```

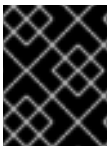
14.5. プロジェクトごとのリソース制限

クラスター管理者 はリソース制限をプロジェクトごとに設定することができ ます。開発者にはこれらの制限を作成し、編集し、削除することができませんが、アクセス可能なプロジェクトのリソース制限を 表示すること ができます。

第15章 POD の PRESET (プリセット) を使用した情報の POD への挿入

15.1. 概要

Pod の Preset は、ユーザーが指定する情報を Pod の作成時に Pod に挿入するオブジェクトです。



重要

OpenShift Container Platform 3.7 の時点で、Pod の Preset はサポートされなくなりました。

挿入可能な Pod の Preset オブジェクトを使用します。

- シークレットオブジェクト
- **ConfigMap** オブジェクト
- ストレージボリューム
- コンテナーボリュームのマウント
- 環境変数

開発者は、すべての情報を Pod に追加するために Pod ラベルが PodPreset のラベルセレクターに一致することのみを確認する必要があります。Pod の ラベル は、一致する ラベルセレクター を持つ 1 つ以上の Pod Preset オブジェクトに Pod を関連付けます。

Pod の Preset を使用すると、開発者は Pod が消費するサービスの詳細を把握せずに Pod をプロビジョニングできます。管理者は、開発者が Pod をデプロイできないようにすることなく、サービスの設定項目を開発者に表示されないようにすることができます。たとえば、管理者は環境変数を使ってシークレットおよびデータベースポート経由でデータベースの名前、ユーザー名、パスワードを提供する Pod の Preset を作成できます。Pod 開発者は、すべての情報を Pod に含めるために使用するラベルのみを知っている必要があります。開発者は Pod の Preset を作成し、すべての同じタスクを実行することもできます。たとえば、開発者は環境変数を複数の Pod に自動的に挿入する Preset を作成できます。

Pod の Preset が Pod に適用されると、OpenShift Container Platform は Pod 仕様を変更し、挿入可能なデータを追加し、Pod の Preset で変更されたことを示すアノテーションを Pod 仕様に付けます。アノテーションの形式は以下のとおりです。

```
podpreset.admission.kubernetes.io/<pod-preset name>: `resource version`
```

クラスターで Pod の Preset を使用するには、以下を実行します。

- 管理者は、**/etc/origin/master/master -config.yaml で Pod の Preset 受付コントローラープラグインを有効** にする必要があります。
- Pod の Preset の作成者は Pod の Preset で API タイプの **settings.k8s.io/v1alpha1/podpreset** を有効にし、挿入可能な情報を Pod の Preset に追加する必要があります。

Pod の作成時にエラーが生じる場合は、Pod の Preset から挿入されたリソースなしに Pod が作成されている場合です。

Pod 仕様の **podpreset.admission.kubernetes.io/exclude: "true"** パラメーターを使用して、Pod の Preset 変更によって特定の Pod が変更されないようにすることができます。以下の Pod 仕様の例 を参照してください。

注記

Pod の Preset 機能は、サービスカタログ がインストールされている場合にのみ利用できます。

Pod の Preset オブジェクトの例

```
kind: PodPreset
apiVersion: settings.k8s.io/v1alpha1 (1)
metadata:
  name: allow-database (2)
spec:
  selector:
    matchLabels:
      role: frontend (3)
  env:
    - name: DB_PORT (4)
      value: "6379" (5)
  envFrom:
    - configMapRef: (6)
      name: etcd-env-config
    - secretKeyRef: (7)
      name: test-secret
  volumeMounts: (8)
    - mountPath: /cache
      name: cache-volume
  volumes: (9)
    - name: cache-volume
      emptyDir: {}
```

(1) **settings.k8s.io/v1alpha1** API を指定します。

(2) Pod の Preset の名前。この名前は Pod アノテーションで使用されます。

(3) Pod 仕様のラベルに一致するラベルセレクターです。

(4) (5) コンテナーに渡す環境変数を作成します。

(6) **ConfigMap** を Pod 仕様に追加します。

(7) シークレットオブジェクトを Pod 仕様に追加します。

(8) 外部ストレージボリュームをコンテナー内にマウントするかどうかを指定します。

(9) コンテナーが利用できるストレージボリュームを定義します。

Pod 仕様の例

```
apiVersion: v1
```

```
kind: Pod
metadata:
  name: website
  labels:
    app: website
    role: frontend (1)
spec:
  containers:
    - name: website
      image: ecorp/website
      ports:
        - containerPort: 80
```

(1) Pod の Preset のラベルセレクターに一致するラベルです。

Pod の Preset 適用後の Pod 仕様の例

```
apiVersion: v1
kind: Pod
metadata:
  name: website
  labels:
    app: website
    role: frontend
  annotations:
    podpreset.admission.kubernetes.io/allow-database: "resource version" (1)
spec:
  containers:
    - name: website
      image: ecorp/website
      volumeMounts: (2)
        - mountPath: /cache
          name: cache-volume
      ports:
        - containerPort: 80
      env: (3)
        - name: DB_PORT
          value: "6379"
      envFrom: (4)
        - configMapRef:
          name: etcd-env-config
        - secretKeyRef:
          name: test-secret
  volumes: (5)
    - name: cache-volume
      emptyDir: {}
```

(1) Pod 仕様が変更を禁止するように設定されていない場合に、Pod の Preset が挿入されたことを示すアノテーションが追加されます。

(2) ボリュームマウントが Pod に追加されます。

(3) 環境変数が Pod に追加されます。

4 Pod に追加される **ConfigMap** およびシークレットオブジェクト。

5 ボリュームマウントが Pod に追加されます。

Pod を Pod の Preset から除外する Pod 仕様の例

```
apiVersion: v1
kind: Pod
metadata:
  name: no-podpreset
  labels:
    app: website
    role: frontend
  annotations:
    podpreset.admission.kubernetes.io/exclude: "true" 1
spec:
  containers:
    - name: hello-pod
      image: docker.io/ocpqe/hello-pod
```

1 Pod の Preset 機能がこの Pod を挿入できないようにするにはこのパラメーターを追加します。

15.2. POD の PRESET の作成

以下の例は、Pod の Preset を作成し、使用する方法を示しています。

受付コントローラーの追加

管理者は **/etc/origin/master/master-config.yaml** ファイルをチェックして、Pod の Preset 受付コントローラープラグインが存在することを確認できます。受付コントローラーが存在しない場合は、以下を使用してプラグインを追加します。

```
admissionConfig:
  pluginConfig:
    PodPreset:
      configuration:
        kind: DefaultAdmissionConfig
        apiVersion: v1
        disable: false
```

次に OpenShift Container Platform サービスを再起動します。

```
# systemctl restart atomic-openshift-master-api atomic-openshift-master-controllers
```

Pod の Preset の作成

管理者または開発者は、**settings.k8s.io/v1alpha1** API、挿入する情報、および Pod に一致するラベルセレクターを使用して Pod の Preset を作成します。

```
kind: PodPreset
apiVersion: settings.k8s.io/v1alpha1
metadata:
  name: allow-database
```

```
spec:
  selector:
    matchLabels:
      role: frontend
  env:
    - name: DB_PORT
      value: "6379"
  volumeMounts:
    - mountPath: /cache
      name: cache-volume
  volumes:
    - name: cache-volume
      emptyDir: {}
```

Pod の作成

開発者は Pod の Preset のラベルセレクターに一致するラベルを使って Pod を作成します。

1. Pod の Preset のラベルセレクターに一致するラベルで標準的な Pod 仕様を作成します。

```
apiVersion: v1
kind: Pod
metadata:
  name: website
  labels:
    app: website
    role: frontend
spec:
  containers:
    - name: website
      image: ecorp/website
      ports:
        - containerPort: 80
```

2. Pod を作成します。

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

3. 作成後に Pod 仕様をチェックします。

```
$ oc get pod website -o yaml

apiVersion: v1
kind: Pod
metadata:
  name: website
  labels:
    app: website
    role: frontend
  annotations:
    podpreset.admission.kubernetes.io/allow-database: "resource version" (1)
spec:
  containers:
    - name: website
      image: ecorp/website
      volumeMounts: (2)
```

```
      - mountPath: /cache
        name: cache-volume
    ports:
      - containerPort: 80
    env: 3
      - name: DB_PORT
        value: "6379"
  volumes:
    - name: cache-volume
      emptyDir: {}
```

1 2 3 アノテーションが含まれており、コンテナーのストレージおよび環境変数が挿入されています。

15.3. 複数の POD の PRESET の使用

複数の Pod 挿入ポリシーを挿入するために複数の Pod の Preset を使用することができます。

- Pod の Preset 受付コントローラープラグインが有効に なっていることを確認します。
- 環境変数、マウントポイントおよび/またはストレージボリュームを使用して、以下のような Pod の Preset を作成します。

```
kind: PodPreset
apiVersion: settings.k8s.io/v1alpha1
metadata:
  name: allow-database
spec:
  selector:
    matchLabels:
      role: frontend 1
  env:
    - name: DB_PORT
      value: "6379"
  volumeMounts:
    - mountPath: /cache
      name: cache-volume
  volumes:
    - name: cache-volume
      emptyDir: {}
```

1 Pod ラベルに一致するラベルセレクターです。

- 以下のように 2 つ目の Pod の Preset を作成します。

```
kind: PodPreset
apiVersion: settings.k8s.io/v1alpha1
metadata:
  name: proxy
spec:
  selector:
    matchLabels:
```

```
    role: frontend 1
  volumeMounts:
    - mountPath: /etc/proxy/configs
      name: proxy-volume
  volumes:
    - name: proxy-volume
      emptyDir: {}
```

1 Pod ラベルに一致するラベルセレクターです。

- 標準的な Pod 仕様を作成します。

```
apiVersion: v1
kind: Pod
metadata:
  name: website
  labels:
    app: website
    role: frontend 1
spec:
  containers:
    - name: website
      image: ecorp/website
      ports:
        - containerPort: 80
```

1 Pod の Preset ラベルセレクターのいずれにも一致するラベルです。

- Pod を作成します。

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

- 作成後に Pod 仕様をチェックします。

```
apiVersion: v1
kind: Pod
metadata:
  name: website
  labels:
    app: website
    role: frontend
  annotations:
    podpreset.admission.kubernetes.io/allow-database: "resource version" 1
    podpreset.admission.kubernetes.io/proxy: "resource version" 2
spec:
  containers:
    - name: website
      image: ecorp/website
      volumeMounts:
        - mountPath: /cache
          name: cache-volume
        - mountPath: /etc/proxy/configs
          name: proxy-volume
```

```
    ports:
      - containerPort: 80
    env:
      - name: DB_PORT
        value: "6379"
volumes:
  - name: cache-volume
    emptyDir: {}
  - name: proxy-volume
    emptyDir: {}
```

1 2 複数の Pod の Preset が挿入されたことを示すアノテーションです。

15.4. POD の PRESET の削除

以下のコマンドを使用して Pod の Preset を削除できます。

```
$ oc delete podpreset <name>
```

以下に例を示します。

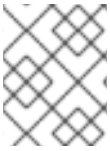
```
$ oc delete podpreset allow-database

podpreset "allow-database" deleted
```

第16章 クラスターへのトラフィックの送信

16.1. クラスターへのトラフィックの送信

OpenShift Container Platform は、クラスター内で実行されるサービスを使ってクラスター外からの通信を実行するための複数の方法を提供します。



注記

このセクションの手順では、クラスターの管理者が事前に行っておく必要のある前提条件があります。

管理者は、一定範囲の外部 IP アドレスから固有の外部 IP アドレスをサービスに割り当てることにより外部トラフィックが到達できるサービスのエンドポイントを公開することができます。管理者は CIDR 表記を使用してアドレスの範囲を指定でき、これによりユーザーはクラスターに対して外部 IP アドレスの要求を行うことができます。

各 IP アドレスは、各サービスがそれぞれ固有のエンドポイントを持つように 1 つのサービスにのみ割り当てる必要があります。起こり得るポートのクラッシュについては 「first-come, first-served (先着順)」で処理されます。

以下は、推奨事項を推奨される順で示しています。

- HTTP/HTTPS を使用する場合はルーターを使用します。
- HTTPS 以外の TLS で暗号化されたプロトコルを使用する場合 (TLS と SNI ヘッダーの使用など) はルーターを使用します。
- それ以外の場合は、ロードバランサー、外部 IP、または NodePort を使用します。

方法	目的
ルーターの使用	HTTP/HTTPS トラフィックおよび HTTPS 以外の TLS で暗号化されたプロトコル (TLS と SNI ヘッダーの使用など) へのアクセスを許可します。
ロードバランサーサービスを使用したパブリック IP の自動割り当て	プールから割り当てられた IP アドレスを使った非標準ポートへのトラフィックを許可します。
外部 IP のサービスへの手動割り当て	特定の IP アドレスを使った非標準ポートへのトラフィックを許可します。
NodePort の設定	クラスターのすべてのノードでサービスを公開します。

16.2. ルーターを使用したトラフィックのクラスターへの送信

16.2.1. 概要

ルーターの使用は、OpenShift Container Platform クラスターへの外部アクセスを許可する最も一般的な方法です。

ルーター は外部要求を許可し、設定された ルート に基づいてそれらをプロキシー送信するよう設定されます。これは Web アプリケーションに対応する HTTP/HTTPS(SNI)/TLS(SNI) に制限されます。

16.2.2. 管理者の前提条件

この手順を開始する前に、管理者は以下の条件を満たしていることを確認する必要があります。

- 要求がクラスターに到達するように外部ポートをクラスターネットワーク環境にセットアップします。たとえば名前については、クラスター内の特定ノードまたは他の IP アドレスを参照するように DNS で設定できます。DNS ワイルドカード 機能はクラスター内の IP アドレスに対して名前のサブセットを設定するために使用できます。これを使用するユーザーは、管理者に問い合わせることなくクラスター内でルートをセットアップできます。
- 各ノードのローカルのファイアウォールが、IP アドレスの到達要求を許可していることを確認します。
- OpenShift Container Platform クラスターを、適切なユーザーアクセスを許可 するアイデンティティープロバイダーを使用 するように設定します。
- クラスター管理者ロールを持つユーザーが 1 名以上いることを確認します。このロールをユーザーに追加するには、以下のコマンドを実行します。

```
oc adm policy add-cluster-role-to-user cluster-admin username
```

- OpenShift Container Platform クラスターを、1 つ以上のマスターと 1 つ以上のノード、およびクラスターへのネットワークアクセスのあるクラスター外のシステムと共に用意します。この手順では、外部システムがクラスターと同じサブセットにあることを前提とします。別のサブセットの外部システムに必要な追加のネットワーク設定については、このトピックでは扱いません。

16.2.2.1. パブリック IP 範囲の定義

サービスへのアクセスを許可するための最初の手順として、マスター設定ファイルで外部 IP アドレス範囲を定義します。

1. クラスター管理者ロールを持つユーザーとして OpenShift Container Platform にログインします。

```
$ oc login
Authentication required (openshift)
Username: admin
Password:
Login successful.

You have access to the following projects and can switch between them with 'oc project
<projectname>':
  * default
Using project "default".
```

2. 以下のように **/etc/origin/master/master-config.yaml** ファイルで **externalIPNetworkCIDRs** パラメーターを設定します。

```
networkConfig:
  externalIPNetworkCIDRs:
  - <ip_address>/<cidr>
```

以下に例を示します。

```
networkConfig:
  externalIPNetworkCIDRs:
  - 192.168.120.0/24
```

3. 変更を有効にするために OpenShift Container Platform マスターサービスを再起動します。

```
# systemctl restart atomic-openshift-master-api atomic-openshift-master-controllers
```

注意

IP アドレスプールはクラスター内の 1 つ以上のノードで終了している必要があります。

16.2.3. プロジェクトおよびサービスの作成

公開するプロジェクトおよびサービスが存在しない場合、最初にプロジェクトを作成し、次にサービスを作成します。

プロジェクトおよびサービスがすでに存在する場合は、**サービスを公開し、ルートを作成する** という次の手順に進みます。

1. OpenShift Container Platform にログインします。
2. サービスの新規プロジェクトを作成します。

```
$ oc new-project <project_name>
```

以下に例を示します。

```
$ oc new-project external-ip
```

3. **oc new-app** コマンドを使用してサービスを作成します。
以下に例を示します。

```
$ oc new-app \
    -e MYSQL_USER=admin \
    -e MYSQL_PASSWORD=redhat \
    -e MYSQL_DATABASE=mysqldb \
    registry.access.redhat.com/openshift3/mysql-55-rhel7
```

4. 以下のコマンドを実行して新規サービスが作成されていることを確認します。

```
oc get svc
NAME             CLUSTER-IP      EXTERNAL-IP   PORT(S)    AGE
mysql-55-rhel7   172.30.131.89   <none>        3306/TCP   13m
```

デフォルトで、新規サービスには外部 IP アドレスがありません。

16.2.4. サービスを公開し、ルートを作成する

oc expose コマンドを使用して、サービスをルートと して公開する必要があります。

サービスを公開するには、以下を実行します。

1. OpenShift Container Platform にログインします。
2. 公開するサービスが置かれているプロジェクトにログインします。

```
$ oc project project1
```

3. 以下のコマンドを実行してルートを公開します。

```
oc expose service <service-name>
```

以下に例を示します。

```
oc expose service mysql-55-rhel7
route "mysql-55-rhel7" exposed
```

4. マスターで cURL などのツールを使用し、サービスのクラスター IP アドレスを使用してサービスに到達できることを確認します。

```
curl <pod-ip>:<port>
```

以下に例を示します。

```
curl 172.30.131.89:3306
```

このセクションの例では、クライアントアプリケーションを必要とする MySQL サービスを使用しています。**Got packets out of order** のメッセージと共に文字ストリングを取得する場合は、このサービスに接続されていることになります。

MySQL クライアントがある場合は、標準 CLI コマンドでログインします。

```
$ mysql -h 172.30.131.89 -u admin -p
Enter password:
Welcome to the MariaDB monitor.  Commands end with ; or \g.

MySQL [(none)]>
```

16.2.5. ルーターの設定

管理者と連携してルーターを設定します。 外部要求を許可し、設定されたルートに基づいてそれらをプロキシー送信するようにルーターを設定します。

管理者は ワイルドカード DNS エントリーを作成してからルーターをセットアップできます。その後は管理者に問い合わせることなく edge ルーターをセルフサービスで提供できます。

ルーターには、ユーザーがホスト名をセルフプロビジョニングできるかどうか、またはホスト名に特定のパターンを使用する必要があるかどうかを管理者が指定できるようにするコントロールがあります。

一連のルートが各種プロジェクトで作成される場合、ルートのセット全体が一連のルーターで利用可能になります。各ルートはルートのセットからルートを許可 (または選択) します。デフォルトで、すべてのルーターはすべてのルートを許可します。

すべてのプロジェクトのすべてのラベルを表示するパーミッションを持つルーターは ラベル に基づいて許可するルートを選択できます。これは ルーターのシャード化 と呼ばれています。これは一連のルーター間で着信トラフィックの負荷を分散する際や、特定のルーターへのトラフィックを分離する際に役立ちます。たとえば、Company A のトラフィックをあるルーターに設定し、Company B のトラフィックを別のルーターに指定する場合などに役立ちます。

ルーターは特定のノードで実行されるため、ルーターまたはノードが失敗すると、Ingress トラフィックが停止します。この影響は、各種の異なるノードで冗長なルーターを作成し、高可用性 を使用してノードの失敗時にルーター IP アドレスを切り換えることなどによって軽減することができます。

16.2.6. VIP を使用した IP フェイルオーバーの設定

オプションとして、管理者は IP フェイルオーバーを設定できます。

IP フェイルオーバーは、ノードセットの仮想 IP (VIP) アドレスのプールを管理します。セットのすべての VIP はセットから選択されるノードによって提供されます。VIP は単一ノードが利用可能である限り提供されます。ノード上で VIP を明示的に配布する方法がないため、VIP のないノードがある可能性も、多数の VIP を持つノードがある可能性もあります。そのため、VIPのないノードと、複数のVIPのあるノードが存在する場合があります。ノードが 1 つのみ存在する場合は、すべての VIP がそのノードに配置されます。

VIP はクラスター外からルーティングできる必要があります。

IP フェイルオーバーを設定するには、以下を実行します。

1. マスターで **ipfailover** サービスアカウントに十分なセキュリティー権限があることを確認します。

```
oc adm policy add-scc-to-user privileged -z ipfailover
```

2. 以下のコマンドを実行して IP フェイルオーバーを作成します。

```
oc adm ipfailover --virtual-ips=<exposed-ip-address> --watch-port=<exposed-port> --
replicas=<number-of-pods> --create
```

以下に例を示します。

```
oc adm ipfailover --virtual-ips="172.30.233.169" --watch-port=32315 --replicas=4 --create
--> Creating IP failover ipfailover ...
    serviceaccount "ipfailover" created
    deploymentconfig "ipfailover" created
--> Success
```

16.3. ロードバランサーを使用したトラフィックのクラスターへの送信

16.3.1. 概要

特定の外部 IP アドレスを必要としない場合、ロードバランサーサービスを OpenShift Container Platform クラスターへの外部アクセスを許可するよう設定することができます。

ロードバランサーサービスは設定済みのプールから固有の IP を割り当てます。ロードバランサーには単一の edge ルーター IP があります (これは仮想 IP (VIP) の場合もありますが、初期の負荷分散では単一マシンになります。

このプロセスには以下を実行することが関係します。

- 管理者が前提条件を実行する
- 開発者がプロジェクトおよびサービスを作成する (公開されるサービスが存在しない場合)
- 開発者がサービスを公開し、ルートを作成する
- 開発者がロードバランサーサービスを作成する
- ネットワーク管理者がサービスへのネットワークを設定する

16.3.2. 管理者の前提条件

この手順を開始する前に、管理者は以下の条件を満たしていることを確認する必要があります。

- 要求がクラスターに到達するように外部ポートをクラスターネットワーク環境にセットアップします。たとえば名前については、クラスター内の特定ノードまたは他の IP アドレスを参照するように DNS で設定できます。DNS ワイルドカード 機能はクラスター内の IP アドレスに対して名前のサブセットを設定するために使用できます。これを使用するユーザーは、管理者に問い合わせることなくクラスター内でルートをセットアップできます。
- 各ノードのローカルのファイアウォールが、IP アドレスの到達要求を許可していることを確認します。
- OpenShift Container Platform クラスターを、適切なユーザーアクセスを許可 するアイデンティティープロバイダーを使用 するように設定します。
- クラスター管理者ロールを持つユーザーが 1 名以上いることを確認します。このロールをユーザーに追加するには、以下のコマンドを実行します。

```
oc adm policy add-cluster-role-to-user cluster-admin username
```

- OpenShift Container Platform クラスターを、1 つ以上のマスターと 1 つ以上のノード、およびクラスターへのネットワークアクセスのあるクラスター外のシステムと共に用意します。この手順では、外部システムがクラスターと同じサブセットにあることを前提とします。別のサブセットの外部システムに必要な追加のネットワーク設定については、このトピックでは扱いません。

16.3.2.1. パブリック IP 範囲の定義

サービスへのアクセスを許可するための最初の手順として、マスター設定ファイルで外部 IP アドレス範囲を定義します。

1. クラスター管理者ロールを持つユーザーとして OpenShift Container Platform にログインします。

```
$ oc login
Authentication required (openshift)
Username: admin
Password:
Login successful.
```

```
You have access to the following projects and can switch between them with 'oc project
<projectname>':
  * default
Using project "default".
```

2. 以下のように **/etc/origin/master/master-config.yaml** ファイルで **externalIPNetworkCIDRs** パラメーターを設定します。

```
networkConfig:
  externalIPNetworkCIDRs:
  - <ip_address>/<cidr>
```

以下に例を示します。

```
networkConfig:
  externalIPNetworkCIDRs:
  - 192.168.120.0/24
```

3. 変更を有効にするために OpenShift Container Platform マスターサービスを再起動します。

```
# systemctl restart atomic-openshift-master-api atomic-openshift-master-controllers
```

注意

IP アドレスプールはクラスター内の 1 つ以上のノードで終了している必要があります。

16.3.3. プロジェクトおよびサービスの作成

公開するプロジェクトおよびサービスが存在しない場合、最初にプロジェクトを作成し、次にサービスを作成します。

プロジェクトおよびサービスがすでに存在する場合は、**サービスを公開し、ルートを作成する** という次の手順に進みます。

1. OpenShift Container Platform にログインします。
2. サービスの新規プロジェクトを作成します。

```
$ oc new-project <project_name>
```

以下に例を示します。

```
$ oc new-project external-ip
```

3. **oc new-app** コマンドを使用してサービスを作成します。
以下に例を示します。

```
$ oc new-app \
    -e MYSQL_USER=admin \
    -e MYSQL_PASSWORD=redhat \
    -e MYSQL_DATABASE=mysqldb \
    registry.access.redhat.com/openshift3/mysql-55-rhel7
```

4. 以下のコマンドを実行して新規サービスが作成されていることを確認します。

```
oc get svc
NAME             CLUSTER-IP      EXTERNAL-IP   PORT(S)    AGE
mysql-55-rhel7   172.30.131.89   <none>        3306/TCP   13m
```

デフォルトで、新規サービスには外部 IP アドレスがありません。

16.3.4. サービスを公開し、ルートを作成する

oc expose コマンドを使用して、サービスをルートと して公開する必要があります。

サービスを公開するには、以下を実行します。

1. OpenShift Container Platform にログインします。
2. 公開するサービスが置かれているプロジェクトにログインします。

```
$ oc project project1
```

3. 以下のコマンドを実行してルートを公開します。

```
oc expose service <service-name>
```

以下に例を示します。

```
oc expose service mysql-55-rhel7
route "mysql-55-rhel7" exposed
```

4. マスターで cURL などのツールを使用し、サービスのクラスター IP アドレスを使用してサービスに到達できることを確認します。

```
curl <pod-ip>:<port>
```

以下に例を示します。

```
curl 172.30.131.89:3306
```

このセクションの例では、クライアントアプリケーションを必要とする MySQL サービスを使用しています。**Got packets out of order** のメッセージと共に文字ストリングを取得する場合は、このサービスに接続されていることになります。

MySQL クライアントがある場合は、標準 CLI コマンドでログインします。

```
$ mysql -h 172.30.131.89 -u admin -p
Enter password:
Welcome to the MariaDB monitor.  Commands end with ; or \g.

MySQL [(none)]>
```

次に以下のタスクを実行します。

- ロードバランサーサービスの作成

- ネットワークの設定
- IP フェイルオーバーの設定

16.3.5. ロードバランサーサービスの作成

ロードバランサーサービスを作成するには、以下を実行します。

1. OpenShift Container Platform にログインします。
2. 公開するサービスが置かれているプロジェクトを読み込みます。プロジェクトまたはサービスが存在しない場合は、「プロジェクトおよびサービスの作成」を参照してください。

```
$ oc project project1
```

3. マスターノードでテキストファイルを開き、以下のテキストを貼り付け、必要に応じてファイルを編集します。

例16.1 ロードバランサー設定ファイルのサンプル

```
apiVersion: v1
kind: Service
metadata:
  name: egress-2 1
spec:
  ports:
  - name: db
    port: 3306 2
  loadBalancerIP:
  type: LoadBalancer 3
  selector:
    name: mysql 4
```

1. ロードバランサーサービスの説明となる名前を入力します。
2. 公開するサービスがリッスンしている同じポートを入力します。
3. タイプに **loadbalancer** を入力します。
4. サービスの名前を入力します。

4. ファイルを保存し、終了します。
5. 以下のコマンドを実行してサービスを作成します。

```
oc create -f <file-name>
```

以下は例になります。

```
oc create -f mysql-lb.yaml
```

6. 以下のコマンドを実行して新規サービスを表示します。

```
oc get svc
NAME           CLUSTER-IP      EXTERNAL-IP                PORT(S)             AGE
egress-2       172.30.236.167  172.29.121.74,172.29.121.74  3306/TCP            6s
```

サービスには自動的に割り当てられた外部 IP アドレスがあることに注意してください。

7. マスターで cURL などのツールを使用し、パブリック IP アドレスを使用してサービスに到達できることを確認します。

```
$ curl <public-ip>:<port>
```

++ 例：

```
$ curl 172.29.121.74:3306
```

このセクションの例では、クライアントアプリケーションを必要とする MySQL サービスを使用しています。**Got packets out of order** のメッセージと共に文字ストリングを取得する場合は、このサービスに接続していることになります。

MySQL クライアントがある場合は、標準 CLI コマンドでログインします。

```
$ mysql -h 172.30.131.89 -u admin -p
Enter password:
Welcome to the MariaDB monitor.  Commands end with ; or \g.

MySQL [(none)]>
```

16.3.6. ネットワークの設定

以下の手順は、他のノードから公開されたサービスにアクセスするために必要なネットワークを設定するための一般的なガイドラインです。ネットワーク環境は異なるため、お使いに環境に必要な特定の設定についてはネットワーク管理者にお問い合わせください。

以下の手順は、すべてのシステムが同じサブネットにあることを前提としています。

ノード上:

1. ネットワークを稼働させるためにネットワークを再起動します。

```
$ service network restart
Restarting network (via systemctl):  [  OK  ]
```

ネットワークが稼働していない場合、以下のコマンドを実行すると **Network is unreachable** などのエラーメッセージが表示されます。

2. マスター上の公開されたサービスの IP アドレスとマスターホストの IP アドレス間のルートを追加します。ネットワークルートのネットマスクを使用する場合には、使用するネットマスクおよび **netmask** オプションを使用します。

```
$ route add -net 172.29.121.74 netmask 255.255.0.0 gw 10.16.41.22 dev eth0
```

3. cURL などのツールを使用して、パブリック IP アドレスを使用してサービスに到達できることを確認します。

```
$ curl <public-ip>:<port>
```

以下に例を示します。

```
curl 172.29.121.74:3306
```

Got packets out of order のメッセージと共に文字ストリングを取得する場合は、サービスがノードからアクセス可能であることになります。

クラスター内にないシステム上:

1. ネットワークを稼働させるためにネットワークを再起動します。

```
$ service network restart
Restarting network (via systemctl):  [  OK  ]
```

ネットワークが稼働していない場合、以下のコマンドを実行すると **Network is unreachable** などのエラーメッセージが表示されます。

2. マスター上の公開されたサービスの IP アドレスとマスターホストの IP アドレス間のルートを追加します。ネットワークルートのネットマスクを使用する場合には、使用するネットマスクおよび **netmask** オプションを使用します。

```
$ route add -net 172.29.121.74 netmask 255.255.0.0 gw 10.16.41.22 dev eth0
```

3. パブリック IP アドレスを使用してサービスに到達できることを確認します。

```
$ curl <public-ip>:<port>
```

以下に例を示します。

```
curl 172.29.121.74:3306
```

Got packets out of order のメッセージと共に文字ストリングを取得する場合、サービスがクラスター外からアクセス可能であることになります。

16.3.7. VIP を使用した IP フェイルオーバーの設定

オプションとして、管理者は IP フェイルオーバーを設定できます。

IP フェイルオーバーは、ノードセットの仮想 IP (VIP) アドレスのプールを管理します。セットのすべての VIP はセットから選択されるノードによって提供されます。VIP は単一ノードが利用可能である限り提供されます。ノード上で VIP を明示的に配布する方法がないため、VIP のないノードがある可能性も、多数の VIP を持つノードがある可能性もあります。そのため、VIPのないノードと、複数のVIPのあるノードが存在する場合があります。ノードが 1 つのみ存在する場合は、すべての VIP がそのノードに配置されます。

VIP はクラスター外からルーティングできる必要があります。

IP フェイルオーバーを設定するには、以下を実行します。

1. マスターで **ipfailover** サービスアカウントに十分なセキュリティー権限があることを確認します。

```
oc adm policy add-scc-to-user privileged -z ipfailover
```

2. 以下のコマンドを実行して IP フェイルオーバーを作成します。

```
oc adm ipfailover --virtual-ips=<exposed-ip-address> --watch-port=<exposed-port> --
replicas=<number-of-pods> --create
```

以下に例を示します。

```
oc adm ipfailover --virtual-ips="172.30.233.169" --watch-port=32315 --replicas=4 --create
--> Creating IP failover ipfailover ...
    serviceaccount "ipfailover" created
    deploymentconfig "ipfailover" created
--> Success
```

16.4. サービスの外部 IP を使用したトラフィックのクラスターへの送信

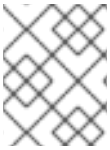
16.4.1. 概要

サービスを公開する 1 つの方法として、外部 IP アドレスをクラスター外からアクセス可能にするサービスに直接割り当てることができます。

「パブリック IP アドレス範囲の定義」で説明されているように、使用する IP アドレスの範囲を作成していることを確認します。

サービスに外部 IP を設定することにより、OpenShift Container Platform は、その IP アドレスをターゲットとするクラスターノードに到達するトラフィックが内部 Pod のいずれかに送信されることを許可する IP テーブルルールをセットアップします。これは内部サービス IP アドレスと似ていますが、外部 IP は OpenShift Container Platform に対し、このサービスが所定の IP で外部に公開される必要があることを示します。管理者は、この IP アドレスをクラスター内のノードのいずれかのホスト (ノード) インターフェースに割り当てる必要があります。または、このアドレスは仮想 IP (VIP) として使用することができます。

OpenShift Container Platform ではこれらの IP を管理しないため、管理者はトラフィックがこの IP を持つノードに到達することを確認する必要があります。



注記

以下は非 HA ソリューションであり、IP フェイルオーバーを設定しません。IP フェイルオーバーはサービスの高可用性を確保するために必要です。

このプロセスには以下を実行することが関係します。

- 管理者が前提条件を実行する
- 開発者がプロジェクトおよびサービスを作成する (公開されるサービスが存在しない場合)
- 開発者がサービスを公開し、ルートを作成する
- 開発者が IP アドレスをサービスに割り当てる
- ネットワーク管理者がサービスへのネットワークを設定する

16.4.2. 管理者の前提条件

この手順を開始する前に、管理者は以下の条件を満たしていることを確認する必要があります。

- 要求がクラスターに到達するように外部ポートをクラスターネットワーク環境にセットアップします。たとえば名前については、クラスター内の特定ノードまたは他の IP アドレスを参照するように DNS で設定できます。DNS ワイルドカード 機能はクラスター内の IP アドレスに対して名前のサブセットを設定するために使用できます。これを使用するユーザーは、管理者に問い合わせることなくクラスター内でルートをセットアップできます。
- 各ノードのローカルのファイアウォールが、IP アドレスの到達要求を許可していることを確認します。
- OpenShift Container Platform クラスターを、適切なユーザーアクセスを許可 するアイデンティティープロバイダーを使用 するように設定します。
- クラスター管理者ロールを持つユーザーが 1 名以上いることを確認します。このロールをユーザーに追加するには、以下のコマンドを実行します。

```
oc adm policy add-cluster-role-to-user cluster-admin username
```

- OpenShift Container Platform クラスターを、1 つ以上のマスターと 1 つ以上のノード、およびクラスターへのネットワークアクセスのあるクラスター外のシステムと共に用意します。この手順では、外部システムがクラスターと同じサブセットにあることを前提とします。別のサブセットの外部システムに必要な追加のネットワーク設定については、このトピックでは扱いません。

16.4.2.1. パブリック IP 範囲の定義

サービスへのアクセスを許可するための最初の手順として、マスター設定ファイルで外部 IP アドレス範囲を定義します。

1. クラスター管理者ロールを持つユーザーとして OpenShift Container Platform にログインします。

```
$ oc login
Authentication required (openshift)
Username: admin
Password:
Login successful.

You have access to the following projects and can switch between them with 'oc project
<projectname>':
  * default
Using project "default".
```

2. 以下のように **/etc/origin/master/master-config.yaml** ファイルで **externalIPNetworkCIDRs** パラメーターを設定します。

```
networkConfig:
  externalIPNetworkCIDRs:
  - <ip_address>/<cidr>
```

以下に例を示します。

```
networkConfig:
  externalIPNetworkCIDRs:
  - 192.168.120.0/24
```

3. 変更を有効にするために OpenShift Container Platform マスターサービスを再起動します。

```
# systemctl restart atomic-openshift-master-api atomic-openshift-master-controllers
```

注意

IP アドレスプールはクラスター内の 1 つ以上のノードで終了している必要があります。

16.4.3. プロジェクトおよびサービスの作成

公開するプロジェクトおよびサービスが存在しない場合、最初にプロジェクトを作成し、次にサービスを作成します。

プロジェクトおよびサービスがすでに存在する場合は、**サービスを公開し、ルートを作成する** という次の手順に進みます。

1. OpenShift Container Platform にログインします。
2. サービスの新規プロジェクトを作成します。

```
$ oc new-project <project_name>
```

以下に例を示します。

```
$ oc new-project external-ip
```

3. **oc new-app** コマンドを使用してサービスを作成します。
以下に例を示します。

```
$ oc new-app \
    -e MYSQL_USER=admin \
    -e MYSQL_PASSWORD=redhat \
    -e MYSQL_DATABASE=mysqldb \
    registry.access.redhat.com/openshift3/mysql-55-rhel7
```

4. 以下のコマンドを実行して新規サービスが作成されていることを確認します。

```
oc get svc
NAME            CLUSTER-IP      EXTERNAL-IP   PORT(S)    AGE
mysql-55-rhel7    172.30.131.89   <none>        3306/TCP   13m
```

デフォルトで、新規サービスには外部 IP アドレスがありません。

16.4.4. サービスを公開し、ルートを作成する

oc expose コマンドを使用して、サービスをルートと して公開する必要があります。

サービスを公開するには、以下を実行します。

1. OpenShift Container Platform にログインします。
2. 公開するサービスが置かれているプロジェクトにログインします。

```
$ oc project project1
```

3. 以下のコマンドを実行してルートを公開します。

```
oc expose service <service-name>
```

以下に例を示します。

```
oc expose service mysql-55-rhel7
route "mysql-55-rhel7" exposed
```

4. マスターで cURL などのツールを使用し、サービスのクラスター IP アドレスを使用してサービスに到達できることを確認します。

```
curl <pod-ip>:<port>
```

以下に例を示します。

```
curl 172.30.131.89:3306
```

このセクションの例では、クライアントアプリケーションを必要とする MySQL サービスを使用しています。**Got packets out of order** のメッセージと共に文字ストリングを取得する場合は、このサービスに接続されていることになります。

MySQL クライアントがある場合は、標準 CLI コマンドでログインします。

```
$ mysql -h 172.30.131.89 -u admin -p
Enter password:
Welcome to the MariaDB monitor.  Commands end with ; or \g.

MySQL [(none)]>
```

次に以下のタスクを実行します。

- IP アドレスのサービスへの割り当て
- ネットワークの設定
- IP フェイルオーバーの設定

16.4.5. IP アドレスのサービスへの割り当て

外部 IP アドレスをサービスに割り当てるには、以下を実行します。

1. OpenShift Container Platform にログインします。
2. 公開するサービスが置かれているプロジェクトを読み込みます。プロジェクトまたはサービスが存在しない場合は、前提条件にある「プロジェクトおよびサービスの作成」を参照してください。

3. 以下のコマンドを実行して、アクセスするサービスに外部 IP アドレスを割り当てます。外部 IP アドレス範囲の IP アドレスを使用します。

```
oc patch svc <name> -p '{"spec":{"externalIPs":["<ip_address>"]}}'
```

<name> はサービスの名前であり、**-p** はサービス JSON ファイルに適用されるパッチを示しています。括弧内の式は特定の IP アドレスを指定されたサービスに割り当てます。

以下に例を示します。

```
oc patch svc mysql-55-rhel7 -p '{"spec":{"externalIPs":["192.174.120.10"]}}'

"mysql-55-rhel7" patched
```

4. 以下のコマンドを実行してサービスにパブリック IP があることを確認します。

```
oc get svc
NAME            CLUSTER-IP      EXTERNAL-IP     PORT(S)    AGE
mysql-55-rhel7     172.30.131.89   192.174.120.10  3306/TCP   13m
```

5. マスターで cURL などのツールを使用し、パブリック IP アドレスを使用してサービスに到達できることを確認します。

```
$ curl <public-ip>:<port>
```

以下に例を示します。

```
curl 192.168.120.10:3306
```

Got packets out of order のメッセージと共に文字ストリングを取得する場合は、このサービスに接続されていることになります。

MySQL クライアントがある場合は、標準 CLI コマンドでログインします。

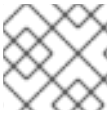
```
$ mysql -h 192.168.120.10 -u admin -p
Enter password:
Welcome to the MariaDB monitor. Commands end with ; or \g.

MySQL [(none)]>
```

16.4.6. ネットワークの設定

外部 IP アドレスが割り当てられた後は、その IP へのルートを作成する必要があります。

以下の手順は、他のノードから公開されたサービスにアクセスするために必要なネットワークを設定するための一般的なガイドラインです。ネットワーク環境は異なるため、お使いに環境に必要な特定の設定についてはネットワーク管理者にお問い合わせください。



注記

以下の手順は、すべてのシステムが同じサブネットにあることを前提としています。

マスター上:

1. ネットワークを稼働させるためにネットワークを再起動します。

```
$ service network restart
Restarting network (via systemctl):  [  OK  ]
```

ネットワークが稼働していない場合、以下のコマンドを実行すると **Network is unreachable** などのエラーメッセージが表示されます。

2. 公開するサービスの外部 IP アドレスおよび **ifconfig** コマンド出力からのホスト IP に関連付けられたデバイス名を使って以下のコマンドを実行します。

```
$ ip address add <external-ip> dev <device>
```

以下に例を示します。

```
$ ip address add 192.168.120.10 dev eth0
```

必要な場合は、以下のコマンドを実行してマスターが置かれているホストサーバーの IP アドレスを取得します。

```
$ ifconfig
```

UP,BROADCAST,RUNNING,MULTICAST のように一覧表示されているデバイスを検索します。

```
eth0: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 1500
      inet 10.16.41.22  netmask 255.255.248.0  broadcast 10.16.47.255
      ...
```

3. マスターが存在するホストの IP アドレスと、マスターホストのゲートウェイ IP アドレスの間のルートを追加します。ネットワークルートのネットマスクを使用する場合には、使用するネットマスクおよび **netmask** オプションを使用します。

```
$ route add -host <host_ip_address> netmask <netmask> gw <gateway_ip_address> dev
<device>
```

以下に例を示します。

```
$ route add -host 10.16.41.22 netmask 255.255.248.0 gw 10.16.41.254 dev eth0
```

netstat -nr コマンドはゲートウェイ IP アドレスを提供します。

```
$ netstat -nr
Kernel IP routing table
Destination     Gateway         Genmask         Flags   MSS Window  irtt Iface
0.0.0.0         10.16.41.254    0.0.0.0         UG        0 0          0 eth0
```

4. 公開されるサービスの IP アドレスとマスターホストの IP アドレス間のルートを追加します。

```
$ route add -net 192.174.120.0/24 gw 10.16.41.22 eth0
```

ノード上:

1. ネットワークを稼働させるためにネットワークを再起動します。

```
$ service network restart
Restarting network (via systemctl):  [  OK  ]
```

ネットワークが稼働していない場合、以下のコマンドを実行すると **Network is unreachable** などのエラーメッセージが表示されます。

2. ノードが配置されているホストの IP アドレスと、ノードホストのゲートウェイ IP との間のルートを追加します。ネットワークルートのネットマスクを使用する場合には、使用するネットマスクおよび **netmask** オプションを使用します。

```
$ route add -net 10.16.40.0 netmask 255.255.248.0 gw 10.16.47.254 eth0
```

ifconfig コマンドはホスト IP を表示します。

```
ifconfig
eth0: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 1500
        inet 10.16.41.71  netmask 255.255.248.0  broadcast 10.19.41.255
```

netstat -nrコマンドはゲートウェイ IP を表示します。

```
netstat -nr
Kernel IP routing table
Destination     Gateway         Genmask         Flags   MSS Window  irtt Iface
0.0.0.0         10.16.41.254    0.0.0.0         UG        0 0          0 eth0
```

3. 公開されるサービスの IP アドレスとマスターノードが置かれているホストサーバーの IP アドレス間のルートを追加します。

```
$ route add -net 192.174.120.0 netmask 255.255.255.0 gw 10.16.41.22 dev eth0
```

4. cURL などのツールを使用して、パブリック IP アドレスを使用してサービスに到達できることを確認します。

```
$ curl <public-ip>:<port>
```

以下に例を示します。

```
curl 192.168.120.10:3306
```

Got packets out of order のメッセージと共に文字ストリングを取得する場合は、サービスがノードからアクセス可能であることになります。

クラスター内にないシステム上:

1. ネットワークを稼働させるためにネットワークを再起動します。

```
$ service network restart
Restarting network (via systemctl):  [  OK  ]
```

ネットワークが稼働していない場合、以下のコマンドを実行すると **Network is unreachable** などのエラーメッセージが表示されます。

2. リモートホストの IP アドレスと、リモートホストのゲートウェイ IP の間のルートを追加します。ネットワークルートのネットマスクを使用する場合には、使用するネットマスクおよび **netmask** オプションを使用します。

```
$ route add -net 10.16.64.0 netmask 255.255.248.0 gw 10.16.71.254 eno1
```

3. マスター上の公開されたサービスの IP アドレスとマスターホストの IP アドレス間のルートを追加します。

```
$ route add -net 192.174.120.0 netmask 255.255.248.0 gw 10.16.41.22
```

4. cURL などのツールを使用して、パブリック IP アドレスを使用してサービスに到達できることを確認します。

```
$ curl <public-ip>:<port>
```

以下に例を示します。

```
curl 192.168.120.10:3306
```

Got packets out of order のメッセージと共に文字ストリングを取得する場合、サービスがクラスター外からアクセス可能であることになります。

16.4.7. VIP を使用した IP フェイルオーバーの設定

オプションとして、管理者は IP フェイルオーバーを設定できます。

IP フェイルオーバーは、ノードセットの仮想 IP (VIP) アドレスのプールを管理します。セットのすべての VIP はセットから選択されるノードによって提供されます。VIP は単一ノードが利用可能である限り提供されます。ノード上で VIP を明示的に配布する方法がないため、VIP のないノードがある可能性も、多数の VIP を持つノードがある可能性もあります。そのため、VIPのないノードと、複数のVIPのあるノードが存在する場合があります。ノードが 1 つのみ存在する場合は、すべての VIP がそのノードに配置されます。

VIP はクラスター外からルーティングできる必要があります。

IP フェイルオーバーを設定するには、以下を実行します。

1. マスターで **ipfailover** サービスアカウントに十分なセキュリティー権限があることを確認します。

```
oc adm policy add-scc-to-user privileged -z ipfailover
```

2. 以下のコマンドを実行して IP フェイルオーバーを作成します。

```
oc adm ipfailover --virtual-ips=<exposed-ip-address> --watch-port=<exposed-port> --
replicas=<number-of-pods> --create
```

以下に例を示します。

```
oc adm ipfailover --virtual-ips="172.30.233.169" --watch-port=32315 --replicas=4 --create
--> Creating IP failover ipfailover ...
    serviceaccount "ipfailover" created
```

```
    deploymentconfig "ipfailover" created
--> Success
```

16.5. NODEPORT を使用したトラフィックのクラスターへの送信

16.5.1. 概要

NodePort を使用してクラスター内のすべてのノードでサービス nodePort を公開 します。

NodePort を使用するには追加のポートリソースが必要です。

ノードポートはノード IP アドレスの静的ポートでサービスを公開します。

NodePort はデフォルトで 30000-32767 の範囲に置かれます。 つまり、NodePort はサービスの意図されるポートに一致しないことが予想されます (たとえば 8080 は 31020 として公開される可能性があります)。

管理者は外部 IP がノードにルーティングされており、すべてのノードのローカルのファイアウォールルールによって開いたポートへのアクセスが許可されることを確認する必要があります。

NodePort および外部 IP は独立しており、両方を同時に使用できます。

16.5.2. 管理者の前提条件

この手順を開始する前に、管理者は以下の条件を満たしていることを確認する必要があります。

- 要求がクラスターに到達するように外部ポートをクラスターネットワーク環境にセットアップします。たとえば名前については、クラスター内の特定ノードまたは他の IP アドレスを参照するように DNS で設定できます。DNS ワイルドカード 機能はクラスター内の IP アドレスに対して名前のサブセットを設定するために使用できます。これを使用するユーザーは、管理者に問い合わせることなくクラスター内でルートをセットアップできます。
- 各ノードのローカルのファイアウォールが、IP アドレスの到達要求を許可していることを確認します。
- OpenShift Container Platform クラスターを、適切なユーザーアクセスを許可 するアイデンティティープロバイダーを使用 するように設定します。
- クラスター管理者ロールを持つユーザーが 1 名以上いることを確認します。このロールをユーザーに追加するには、以下のコマンドを実行します。

  ```
  oc adm policy add-cluster-role-to-user cluster-admin username
  ```

- OpenShift Container Platform クラスターを、1 つ以上のマスターと 1 つ以上のノード、およびクラスターへのネットワークアクセスのあるクラスター外のシステムと共に用意します。この手順では、外部システムがクラスターと同じサブセットにあることを前提とします。別のサブセットの外部システムに必要な追加のネットワーク設定については、このトピックでは扱いません。

16.5.3. サービスの設定

サービスの作成または変更時に nodePort のポート番号を指定します。ポートを手動で指定しない場合は、システムが代わりにこれを割り当てます。

1. マスターノードにログインします。
2. 使用予定のプロジェクトが存在しない場合には、サービス用に新規プロジェクトを作成します。

```
$ oc new-project <project_name>
```

以下に例を示します。

```
$ oc new-project external-ip
```

3. サービス定義を編集して **spec.type:NodePort** を指定し、オプションで 30000-32767 範囲のポートを指定します。

```
apiVersion: v1
kind: Service
metadata:
  name: mysql
  labels:
    name: mysql
spec:
  type: NodePort
  ports:
    - port: 3036
      nodePort: 30036
      name: http
  selector:
    name: mysql
```

4. 以下のコマンドを実行して サービスを作成し ます。

```
$ oc new-app <file-name>
```

以下に例を示します。

```
oc new-app mysql.yaml
```

5. 以下のコマンドを実行して新規サービスが作成されていることを確認します。

```
oc get svc

NAME            CLUSTER_IP       EXTERNAL_IP   PORT(S)                 AGE
mysql           172.30.89.219    <nodes>       3036:30036/TCP          2m
```

外部 IP が **<nodes>** として一覧表示され、ノードのポートが一覧表示されることに注意してください。

<NodeIP>:<NodePort> アドレスを使用してサービスにアクセスできるはずです。

第17章 ルート

17.1. 概要

OpenShift Container Platform ルート は、外部クライアントが名前で到達できるように **www.example.com** などのホスト名で サービス を公開します。

ホスト名の DNS 解決はルーティングとは別に処理されます。 管理者は常に OpenShift Container Platform ルーターに対して正常に解決されるクラウドドメインを設定している場合がありますが、関連性のないホスト名を使用する場合には、ルーターに対して解決されるようにその DNS レコードを別途変更する必要がある場合があります。

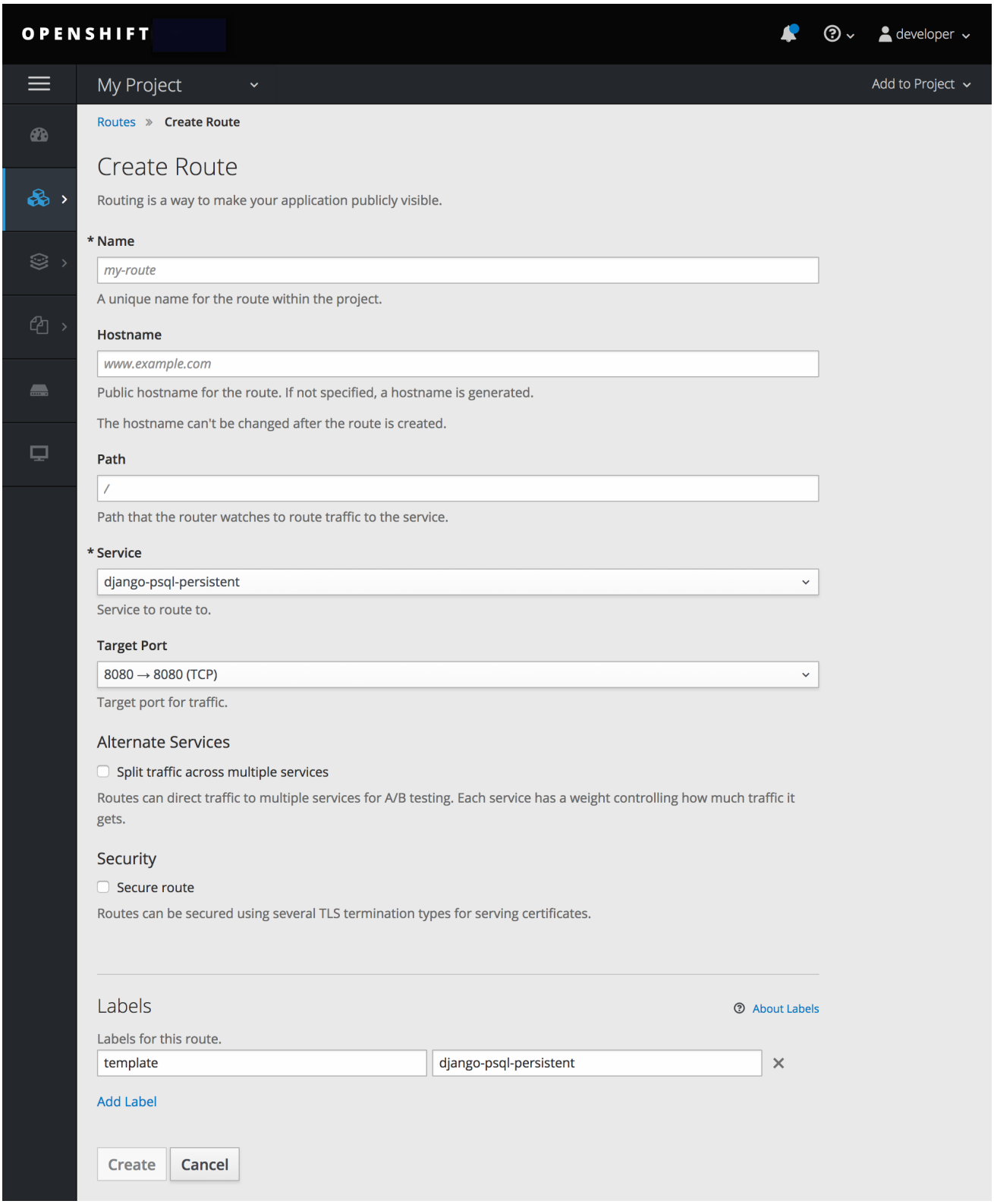
17.2. ルートの作成

Web コンソールまたは CLI を使用して、セキュリティー保護されていないルートとセキュリティー保護されているルートを作成できます。

Web コンソールを使用してナビゲーションの **Applications** セクションの下にある **Routes** ページに移動します。

Create Route をクリックしてプロジェクト内でルートを定義し、作成します。

図17.1 Web コンソールを使用したルートの作成



以下の例では、CLI を使用して非セキュアなルートを作成します。

```
$ oc expose svc/frontend --hostname=www.example.com
```

新規ルートは、**--name** オプションを使用して名前を指定しない限りサービスから名前を継承します。

上記で作成された非セキュアなルートの YAML 定義

```
apiVersion: v1
```

```
kind: Route
metadata:
  name: frontend
spec:
  host: www.example.com
  path: "/test" 1
  to:
    kind: Service
    name: frontend
```

1 パスベースのルーティング については、URL に対して比較対象となるパスコンポーネントを指定します。

CLI を使用したルートの設定についての情報は、「ルート タイプ」を参照 してください。

非セキュアなルートはデフォルト設定であるため、これが最も簡単なセットアップになります。ただし、セキュリティー 保護されたルート は、接続がプライベートのままになるようにセキュリティーを提供します。キーと証明書 (別々に生成し、署名する必要のある PEM 形式のファイル) で暗号化されたセキュリティー保護された HTTPS ルートを作成するには、**create route** コマンドを使用し、オプションで証明書およびキーを指定できます。

注記

TLS は、HTTPS および他の暗号化されたプロトコルにおける SSL の代わりとして使用されます。

```
$ oc create route edge --service=frontend \
    --cert=${MASTER_CONFIG_DIR}/ca.crt \
    --key=${MASTER_CONFIG_DIR}/ca.key \
    --ca-cert=${MASTER_CONFIG_DIR}/ca.crt \
    --hostname=www.example.com
```

上記で作成されたセキュリティー保護されたルートの YAML 定義

```
apiVersion: v1
kind: Route
metadata:
  name: frontend
spec:
  host: www.example.com
  to:
    kind: Service
    name: frontend
  tls:
    termination: edge
    key: |-
      -----BEGIN PRIVATE KEY-----
      [...]
      -----END PRIVATE KEY-----
    certificate: |-
      -----BEGIN CERTIFICATE-----
      [...]
      -----END CERTIFICATE-----
```

```
caCertificate: |-
  -----BEGIN CERTIFICATE-----
  [...]
  -----END CERTIFICATE-----
```

現時点で、パスワードで保護されたキーファイルはサポートされていません。HAProxy は開始時にパスワードを求めるプロンプトを出しますが、このプロセスを自動化する方法はありません。キーファイルからパスフレーズを削除するために、以下を実行できます。

```
# openssl rsa -in <passwordProtectedKey.key> -out <new.key>
```

キーと証明書を指定せずにセキュリティー保護されたルートを作成できます。この場合、ルーターのデフォルト証明書 が TLS 終端に使用されます。

注記

OpenShift Container Platform の TLS 終端は、カスタム証明書を提供する SNI に依存します。ポート 443 で受信される SNI 以外のトラフィックは TLS 終端で処理され、要求されるホスト名に一致しない可能性のあるデフォルト証明書により検証エラーが生じる可能性があります。

すべてのタイプの TLS 終端 および パスベースのルーティング についての詳細は、「アーキテクチャー」セクション を参照してください。

17.3. ルートエンドポイントによる COOKIE 名の制御の許可

OpenShift Container Platform は、すべてのトラフィックを同じエンドポイントにヒットさせることによりステートフルなアプリケーションのトラフィックを可能にするスティッキーセッションを提供します。ただし、エンドポイント Pod が再起動、スケーリング、または設定の変更などによって終了する場合、このステートフル性はなくなります。

OpenShift Container Platform は Cookie を使用してセッションの永続化を設定できます。ルーターはユーザー要求を処理するエンドポイントを選択し、そのセッションの Cookie を作成します。Cookie は要求の応答として戻され、ユーザーは Cookie をセッションの次の要求と共に送り返します。Cookie はルーターに対し、セッションを処理しているエンドポイントを示し、クライアント要求が Cookie を使用して同じ Pod にルーティングされるようにします。

ルート用に自動生成されるデフォルト名を上書きするために Cookie 名を設定できます。Cookie を削除すると、次の要求でエンドポイントの再選択が強制的に実行される可能性があります。そのためサーバーがオーバーロードしている場合には、クライアントからの要求を取り除き、それらの再分配を試行します。

1. 必要な Cookie 名でルートにアノテーションを付けます。

   ```
   $ oc annotate route <route_name> router.openshift.io/cookie_name="<your_cookie_name>"
   ```

   たとえば、**my_cookie** を新規の Cookie 名として指定するには、以下を実行します。

   ```
   $ oc annotate route my_route router.openshift.io/cookie_name="my_cookie"
   ```

2. cookie を保存し、ルートにアクセスします。

   ```
   $ curl $my_route -k -c /tmp/my_cookie
   ```

第18章 外部サービスの統合

18.1. 概要

数多くの OpenShift Container Platform アプリケーションは外部データベースや外部 SaaS エンドポイントなどの外部リソースを使用します。これらの外部リソースはネイティブの OpenShift Container Platform サービスとしてモデリングされ、アプリケーションが他の内部サービスの場合と同様にそれらを使用できるようにします。

egress トラフィック はファイアウォールルールまたは Egress ルーターで制御できます。これにより、アプリケーションサービスの静的 IP アドレスの使用が許可されます。

18.2. 外部データベースのサービスの定義

外部サービスの最も一般的なタイプとして外部データベースを挙げることができます。外部データベースをサポートするには、アプリケーションで以下が必要になります。

1. 通信するエンドポイント。
2. 以下を含む認証情報および位置情報 (coordinate)。
   - ユーザー名
   - パスフレーズ
   - データベース名

外部データベースと統合するためのソリューションには、以下が含まれます。

- **Service** オブジェクト: SaaS プロバイダーを OpenShift Container Platform サービスとして表示します。
- 1 つ以上のサービスの **Endpoint**。
- 認証情報を含む適切な Pod の環境変数。

以下の手順は、外部 MySQL データベースとの統合シナリオについて説明しています。

18.2.1. 手順 1: サービスの定義

サービスは、IP アドレスとエンドポイントを指定するか、または完全修飾ドメイン名 (FQDN) を指定し定義することができます。

18.2.1.1. IP アドレスの使用

1. 外部データベースを表す OpenShift Container Platform サービス を作成します。これは内部サービスを作成する場合と同様ですが、サービスの **Selector** フィールドが異なります。内部 OpenShift Container Platform サービスは **Selector** フィールドで ラベル を使用して Pod をサービスに関連付けます。**EndpointsController** システムコンポーネントは、セレクターに一致する Pod でセレクターを指定するサービスのエンドポイントを同期します。サービスプロキシー および OpenShift Container Platform ルーター はサービス のエンドポイント間でサービスに対する要求の負荷分散を実行します。

   外部リソースを表すサービスには関連付けられる Pod が不要です。代わりに、**Selector** フィールドを未設定のままにします。これは外部サービスであることを表します。 これにより

EndpointsController にこのサービスを無視させ、エンドポイントを手動で指定することができます。

```
kind: "Service"
apiVersion: "v1"
metadata:
  name: "external-mysql-service"
spec:
  ports:
    -
      name: "mysql"
      protocol: "TCP"
      port: 3306
      targetPort: 3306 (1)
      nodePort: 0
selector: {} (2)
```

(1) オプション: サービスによる接続の転送先となるバッキング Pod のポートです。

(2) **selector** フィールドは空白のままにします。

2. 次に、サービスの必要なエンドポイントを作成します。これによりサービスプロキシーとルーターに対し、サービスにダイレクトされたトラフィックを送信する場所が指定されます。

```
kind: "Endpoints"
apiVersion: "v1"
metadata:
  name: "external-mysql-service" (1)
subsets: (2)
  -
    addresses:
      -
        ip: "10.0.0.0" (3)
    ports:
      -
        port: 3306 (4)
        name: "mysql"
```

(1) 直前の手順で定義された **Service** インスタンスの名前です。

(2) サービスへのトラフィックは、複数の指定がある場合に指定された **Endpoints** 間で負荷分散されます。

(3) エンドポイント IP にはループバック (127.0.0.0/8)、リンクローカル (169.254.0.0/16)、またはリンクローカルマルチキャスト (224.0.0.0/24) を使用できません。

(4) **port** および **name** の定義は直前の手順で定義されたサービスの **port** および **name** の値に一致している必要があります。

18.2.1.2. 外部ドメイン名の使用

外部ドメイン名を使用すると、外部サービスの IP アドレスの変更について把握しておく必要がないために外部サービスのリンケージを管理するのが容易になります。

ExternalName サービスにはセレクターまたは定義されたポートまたはエンドポイントがないため、**ExternalName** サービスを使用してトラフィックを外部サービスにダイレクトすることができます。

```
kind: "Service"
apiVersion: "v1"
metadata:
  name: "external-mysql-service"
spec:
  type: ExternalName
  externalName: example.domain.name
selector: {} 1
```

1 **selector** フィールドは空白のままにします。

外部ドメイン名サービスを使用すると、システムに対して **externalName** フィールドの DNS 名 (直前の例では **example.domain.name**) がサービスをサポートするリソースの場所であることを示します。DNS 要求が Kubernetes DNS サーバーに対してなされる場合、CNAME レコードで **externalName** を返し、クライアントに対して返された名前を検索して IP アドレスを取得するように指示します。

18.2.2. 手順 2: サービスの消費

サービスおよびエンドポイントが定義されたので、適切なコンテナーの環境変数を設定し、適切な Pod が認証情報にアクセスしてサービスを使用できるようにします。

```
kind: "DeploymentConfig"
apiVersion: "v1"
metadata:
  name: "my-app-deployment"
spec: 1
  strategy:
    type: "Rolling"
    rollingParams:
      updatePeriodSeconds: 1 2
      intervalSeconds: 1 3
      timeoutSeconds: 120
  replicas: 2
  selector:
    name: "frontend"
  template:
    metadata:
      labels:
        name: "frontend"
    spec:
      containers:
        -
          name: "helloworld"
          image: "origin-ruby-sample"
          ports:
            -
              containerPort: 3306
              protocol: "TCP"
          env:
            -
```

```
        name: "MYSQL_USER"
        value: "${MYSQL_USER}" 4
      -
        name: "MYSQL_PASSWORD"
        value: "${MYSQL_PASSWORD}" 5
      -
        name: "MYSQL_DATABASE"
        value: "${MYSQL_DATABASE}" 6
```

1. **DeploymentConfig** の他のフィールドは省略されます。
2. 各 Pod が次に更新されるまで待機する時間。
3. 更新後に実行されるデプロイメントステータスのポーリング間の待機時間です。
4. サービスで使用するユーザー名です。
5. サービスで使用するパスフレーズです。
6. データベース名です。

外部データベースの環境変数

アプリケーションで外部サービスを使用することは内部サービスを使用することに似ています。アプリケーションには、直前の手順で説明されている認証情報と共に、サービスの環境変数と追加の環境変数が割り当てられます。たとえば、MySQL コンテナーは以下の環境変数を受信します。

- **EXTERNAL_MYSQL_SERVICE_SERVICE_HOST=<ip_address>**
- **EXTERNAL_MYSQL_SERVICE_SERVICE_PORT=<port_number>**
- **MYSQL_USERNAME=<mysql_username>**
- **MYSQL_PASSWORD=<mysql_password>**
- **MYSQL_DATABASE_NAME=<mysql_database>**

アプリケーションは環境からサービスの位置情報 (coordinate) および認証情報を読み取り、サービス経由でデータベースとの接続を確立します。

18.3. 外部 SAAS プロバイダー

外部サービスの一般的なタイプは外部 SaaS エンドポイントです。外部 SaaS プロバイダーをサポートするために、アプリケーションには以下が必要になります。

1. 通信に使用するエンドポイント
2. 以下を含む認証情報のセット
    a. API キー
    b. ユーザー名
    c. パスフレーズ

以下の手順は、外部 SaaS プロバイダーとの統合シナリオについて説明しています。

18.3.1. IP アドレスおよびエンドポイントの使用

1. 外部サービスを表す OpenShift Container Platform サービス を作成します。これは内部サービスを作成することと同様ですが、サービスの **Selector** フィールドが異なります。
内部 OpenShift Container Platform サービスは **Selector** フィールドで ラベル を使用して Pod をサービスに関連付けます。**EndpointsController** というシステムコンポーネントは、セレクターに一致する Pod でセレクターを指定するサービスのエンドポイントを同期します。サービスプロキシー および OpenShift Container Platform ルーター はサービス のエンドポイント間でサービスに対する要求の負荷分散を実行します。

   外部リソースを表すサービスは関連付けられる Pod が不要です。代わりに、**Selector** フィールドを未設定のままにします。これにより**EndpointsController** にこのサービスを無視させ、エンドポイントを手動で指定することができます。

```
kind: "Service"
apiVersion: "v1"
metadata:
  name: "example-external-service"
spec:
  ports:
    -
      name: "mysql"
      protocol: "TCP"
      port: 3306
      targetPort: 3306 (1)
      nodePort: 0
selector: {} (2)
```

   (1) オプション: サービスによる接続の転送先となるバッキング Pod のポートです。

   (2) **selector** フィールドは空白のままにします。

2. 次に、サービスプロキシーおよびルーターにダイレクトされたトラフィックの送信先についての情報が含まれるサービスのエンドポイントを作成します。

```
kind: "Endpoints"
apiVersion: "v1"
metadata:
  name: "example-external-service" (1)
subsets: (2)
- addresses:
  - ip: "10.10.1.1"
  ports:
  - name: "mysql"
    port: 3306
```

   (1) **Service** インスタンスの名前です。

   (2) サービスへのトラフィックはここで指定される **subsets** 間で負荷分散されます。

3. サービスおよびエンドポイントが定義されたので、適切なコンテナーの環境変数を設定し、Pod にサービスを使用するための認証情報を付与します。

```
kind: "DeploymentConfig"
apiVersion: "v1"
metadata:
  name: "my-app-deployment"
spec: (1)
  strategy:
    type: "Rolling"
    rollingParams:
      timeoutSeconds: 120
  replicas: 1
  selector:
    name: "frontend"
  template:
    metadata:
      labels:
        name: "frontend"
    spec:
      containers:
        -
          name: "helloworld"
          image: "openshift/openshift/origin-ruby-sample"
          ports:
            -
              containerPort: 3306
              protocol: "TCP"
          env:
            -
              name: "SAAS_API_KEY" (2)
              value: "<SaaS service API key>"
            -
              name: "SAAS_USERNAME" (3)
              value: "<SaaS service user>"
            -
              name: "SAAS_PASSPHRASE" (4)
              value: "<SaaS service passphrase>"
```

(1) **DeploymentConfig** の他のフィールドは省略されます。

(2) **SAAS_API_KEY**: サービスで使用する API キーです。

(3) **SAAS_USERNAME**: サービスで使用するユーザー名です。

(4) **SAAS_PASSPHRASE**: サービスで使用するパスフレーズです。

これらの変数は環境変数としてコンテナーに追加されます。環境変数を使用することによりサービス間の通信が許可されます。 これには API キーやユーザー名およびパスワード認証または証明書が必要になる場合とそうでない場合があります。

外部 SaaS プロバイダーの環境変数

内部サービスを作成する場合と同様に、アプリケーションには、直前の手順で説明されている認証情報と共に、サービスの環境変数と追加の環境変数が割り当てられます。直前の例では、コンテナーは以下の環境変数を受信します。

- **EXAMPLE_EXTERNAL_SERVICE_SERVICE_HOST=<ip_address>**

- **EXAMPLE_EXTERNAL_SERVICE_SERVICE_PORT=<port_number>**
- **SAAS_API_KEY=<saas_api_key>**
- **SAAS_USERNAME=<saas_username>**
- **SAAS_PASSPHRASE=<saas_passphrase>**

アプリケーションは環境からサービスの位置情報 (coordinate) および認証情報を読み取り、サービス経由でデータベースとの接続を確立します。

18.3.2. 外部ドメイン名の使用

ExternalName サービスにはセレクターや、定義されたポートまたはエンドポイントがありません。**ExternalName** サービスを使用して、クラスター内にない外部サービスに、トラフィックを割り当てることができます。

```
kind: "Service"
apiVersion: "v1"
metadata:
  name: "external-mysql-service"
spec:
  type: ExternalName
  externalName: example.domain.name
selector: {} 1
```

1 **selector** フィールドは空白のままにします。

ExternalName サービスを使用してサービスを **externalName** フィールドの値 (直前の例では **example.domain.name**) にマップします。 これは CNAME レコードを挿入し、サービス名を外部 DNS アドレスに直接マップするので、エンドポイントのレコードは必要ありません。

第19章 デバイスマネージャーの使用

19.1. デバイスマネージャーの機能

重要

デバイスマネージャーはテクノロジープレビュー機能です。テクノロジープレビュー機能は、Red Hat の実稼働環境でのサービスレベルアグリーメント (SLA) ではサポートされていないため、Red Hat では実稼働環境での使用を推奨していません。これらの機能は、近々発表予定の製品機能をリリースに先駆けてご提供することにより、開発プロセスの中でお客様に機能性のテストとフィードバックをしていただくことを目的としています。

Red Hat のテクノロジープレビュー機能のサポートについての詳細は、https://access.redhat.com/support/offerings/techpreview/を参照してください。

デバイスマネージャーは、特殊なノードのハードウェアリソースをデバイスプラグインとして知られる Kubelet プラグインを使って公開するメカニズムを提供する Kubelet 機能です。

すべてのベンダーがデバイスプラグインを実装し、アップストリームのコード変更なしにそれぞれの特殊なハードウェアを公開できます。

デバイスマネージャーはデバイスを **拡張リソース** として公開します。ユーザー Pod は、他の **拡張リソース** を要求するために使用されるのと同じ **制限/要求** メカニズムを使用してデバイスマネージャーで公開されるデバイスを消費できます。

19.1.1. 登録

使用開始時に、デバイスプラグイン は **/var/lib/kubelet/device-plugins/kubelet.sock** の **Register** を起動してデバイスマネージャーに自己登録し、デバイスマネージャーの要求を提供するために **/var/lib/kubelet/device-plugins/<plugin>.sock** で gRPC サービスを起動します。

19.1.2. デバイスの検出および正常性のモニタリング

デバイスマネージャーは、新規登録要求の処理時にデバイスプラグインサービスで **ListAndWatch** リモートプロシージャーコール (RPC) を起動します。応答としてデバイスマネージャーは gRPC ストリームでプラグインからの **デバイス** オブジェクトの一覧を取得します。デバイスマネージャーはプラグインからの新規の更新の有無についてストリームを監視します。プラグイン側では、プラグインはストリームを開いた状態にし、デバイスの状態に変更があった場合には常に新規デバイスの一覧が同じストリーム接続でデバイスマネージャーに送信されます。

19.1.3. デバイスの割り当て

新規 Pod の受付要求の処理時に、Kubelet はデバイスの割り当てのために要求された **Extended Resource** をデバイスマネージャーに送信します。デバイスマネージャーはそのデータベースにチェックインして対応するプラグインが存在するかどうかを確認します。プラグインが存在し、ローカルキャッシュと共に割り当て可能な空きデバイスがある場合、**Allocate** RPC がその特定デバイスのプラグインで起動します。

さらにデバイスプラグインは、ドライバーのインストール、デバイスの初期化、およびデバイスのリセットなどの他のいくつかのデバイス固有の操作も実行できます。これらの機能は実装ごとに異なります。

19.2. デバイスマネージャーの有効化

デバイスマネージャーを有効にし、デバイスプラグインを実装してアップストリームのコード変更なしに特殊なハードウェアを公開できるようにします。

1. ターゲットノードでのデバイスマネージャーのサポートを有効にします。

```
# cat /etc/origin/node/node-config.yaml
...
kubeletArguments:
...
  feature-gates:
  - DevicePlugins=true

# systemctl restart atomic-openshift-node
```

2. デバイスマネージャーが実際に有効にされるように、**/var/lib/kubelet/device-plugins/kubelet.sock** がノードで作成されていることを確認します。これは、デバイスマネージャーの gRPC サーバーが新規プラグインの登録がないかどうかリッスンする UNIX ドメインソケットです。このソケットファイルは、デバイスマネージャーが有効にされている場合にのみ Kubelet の起動時に作成されます。

第20章 デバイスプラグインの使用

20.1. デバイスプラグインの機能

重要

デバイスプラグインはテクノロジープレビューであり、実稼働環境のワークロードには適していません。テクノロジープレビュー機能は、Red Hat の実稼働環境でのサービスレベルアグリーメント (SLA) ではサポートされていないため、Red Hat では実稼働環境での使用を推奨していません。これらの機能は、近々発表予定の製品機能をリリースに先駆けてご提供することにより、開発プロセスの中でお客様に機能性のテストとフィードバックをしていただくことを目的としています。

Red Hat のテクノロジープレビュー機能のサポートについての詳細は、https://access.redhat.com/support/offerings/techpreview/を参照してください。

デバイスプラグインを使用すると、カスタムコードを作成せずに特定のデバイスタイプ (GPU、InfiniBand、またはベンダー固有の初期化およびセットアップを必要とする他の同様のコンピューティングリソース) を OpenShift Container Platform Pod で使用できます。デバイスプラグインは、クラスター全体でハードウェアデバイスを消費するための一貫性のある移植可能なソリューションを提供します。デバイスプラグインはこれらのデバイスのサポートを拡張メカニズムでサポートします。 これにより、これらのデバイスはコンテナーで利用可能となり、デバイスのヘルスチェックやセキュリティーが保護された状態でのデバイスの共有が可能になります。

デバイスプラグインは、特定のハードウェアリソースの管理を行う、ノード上で実行される gRPC サービスです (**atomic-openshift-node.service** の外部にあります)。デバイスプラグインは以下のリモートプロシージャーコール (RPC) をサポートしている必要があります。

```
service DevicePlugin {
    // ListAndWatch returns a stream of List of Devices
    // Whenever a Device state change or a Device disappears, ListAndWatch
    // returns the new list
    rpc ListAndWatch(Empty) returns (stream ListAndWatchResponse) {}

    // Allocate is called during container creation so that the Device
    // Plugin can run device specific operations and instruct Kubelet
    // of the steps to make the Device available in the container
    rpc Allocate(AllocateRequest) returns (AllocateResponse) {}
}
```

20.1.1. 外部デバイスプラグイン

- COS ベースのオペレーティングシステム用の Nvidia GPU デバイスプラグイン
- Nvidia の公式 GPU デバイスプラグイン
- Solarflare デバイスプラグイン
- KubeVirt デバイスプラグイン: vfio および kvm

注記

デバイスプラグイン参照の実装を容易にするため
に、**vendor/k8s.io/kubernetes/pkg/kubelet/cm/deviceplugin/device_plugin_stub.go**
という Device Manager コードのスタブデバイスプラグインを使用できます。

20.2. デバイスプラグインのデプロイ方法

- DeamonSet は、デバイスプラグインのデプロイメントに推奨される方法です。
- 起動時にデバイスプラグインは、デバイスマネージャー から RPC を送信するためにノードの **/var/lib/kubelet/device-plugin/** での UNIX ドメインソケットの作成を試行します。
- デバイスプラグインは、ソケットの作成のほかにもハードウェアリソース、ホストファイルシステムへのアクセスを管理する必要があるため、特権付きセキュリティーコンテキストで実行される必要があります。
- デプロイメント手順の詳細については、それぞれのデバイスプラグインの実装で確認できます。

第21章 シークレット

21.1. シークレットの使用

このトピックでは、シークレットの重要なプロパティーについて説明し、開発者がこれらを使用する方法の概要を説明します。

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) シークレットのキー名および値の構造を示しています。

(2) **data** フィールドのキーに使用可能な形式については、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** マップのキーに関連付けられた値は単純なテキスト文字列で構成されます。

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

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。

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:bm5ubm5ubm5ubm5ubm5ubm5ubm5ubmdnZ2dnZ2dnZ2dnZ2dnZ2dnZ2cgYXV0aC
BrZXIzCg== 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

21.1.1. 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。

21.1.2. 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)。

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>
```

-

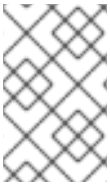
21.1.3. シークレットの種類

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。
- **kubernetes.io/dockercfg**。必須の Docker 認証情報に **.dockercfg** file を使用します。
- **kubernetes.io/dockerconfigjson**。必須の Docker 認証情報に **.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。

検証が必要ない場合には **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** ペアも使用できます。



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はその種類のキー/値の要件に従うことが意図されていることを示します。

異なるシークレットタイプの例については、**シークレットの使用**のコードサンプルを参照してください。

21.1.4. シークレットの更新

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

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

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

注記

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

21.2. ボリュームおよび環境変数のシークレット

シークレットデータを含む YAML ファイルのサンプルを参照してください。

シークレットの作成後に以下を実行できます。

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
```

21.3. イメージプルのシークレット

詳細は、「イメージプルシークレットの使用」を参照してください。

21.4. ソースクローンのシークレット

ビルド時にソースクローンのシークレットを使用する方法についての詳細は、「ビルド入力」を参照してください。

21.5. 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。

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

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

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

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

21.6. 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。

21.6.1. シークレットデータキー

シークレットキーは DNS サブドメインになければなりません。

21.7. 例

例21.1 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

例21.2 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
```

例21.3 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
```

例21.4 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: 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
```

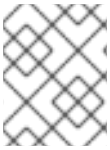
21.8. トラブルシューティング

サービス証明書の生成が失敗する場合 (サービスの **service.alpha.openshift.io/serving-cert-generation-error** のアノテーション):

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

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.alpha.openshift.io/serving-cert-generation-error**、 **service.alpha.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.alpha.openshift.io/serving-cert-generation-error-
$ oc annotate service <service_name> service.alpha.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の後に **-** を付けます。

第22章 CONFIGMAP

22.1. 概要

数多くのアプリケーションには、設定ファイル、コマンドライン引数、および環境変数の組み合わせを使用した設定が必要です。これらの設定アーティファクトは、コンテナー化されたアプリケーションを移植可能な状態に保つためにイメージコンテンツから切り離す必要があります。

ConfigMap オブジェクトは、コンテナーを OpenShift Container Platform に依存しない状態にする一方でコンテナーに設定データを挿入するメカニズムを提供します。**ConfigMap** は、個々のプロパティーなどの粒度の細かい情報や設定ファイル全体または JSON Blob などの粒度の荒い情報を保存するために使用できます。

ConfigMap API オブジェクトは、 Pod で使用したり、コントローラーなどのシステムコントローラーの設定データを保存するために使用できる設定データのキーと値のペアを保持します。**ConfigMap** はシークレットに似ていますが、機密情報を含まない文字列の使用をより効果的にサポートするように設計されています。

以下は例になります。

ConfigMap オブジェクト定義

```
kind: ConfigMap
apiVersion: v1
metadata:
  creationTimestamp: 2016-02-18T19:14:38Z
  name: example-config
  namespace: default
data: 1
  example.property.1: hello
  example.property.2: world
  example.property.file: |-
    property.1=value-1
    property.2=value-2
    property.3=value-3
```

1 設定データが含まれます。

設定データはさまざまな方法で Pod 内で使用できます。**ConfigMap** は以下を実行するために使用できます。

1. 環境変数の値の設定
2. コンテナーのコマンドライン引数の設定
3. ボリュームの設定ファイルの設定

ユーザーとシステムコンポーネントの両方が設定データを **ConfigMap** に保存できます。

22.2. CONFIGMAP の作成

以下のコマンドを使用すると、**ConfigMap** をディレクトリーや特定ファイルまたはリテラル値から簡単に作成できます。

-

```
$ oc create configmap <configmap_name> [options]
```

以下のセクションでは、**ConfigMap** を作成するための各種の方法について説明します。

22.2.1. ディレクトリーからの作成

ConfigMap の設定に必要なデータを含むファイルのあるディレクトリーについて見てみましょう。

```
$ ls example-files
game.properties
ui.properties

$ cat example-files/game.properties
enemies=aliens
lives=3
enemies.cheat=true
enemies.cheat.level=noGoodRotten
secret.code.passphrase=UUDDLRLRBABAS
secret.code.allowed=true
secret.code.lives=30

$ cat example-files/ui.properties
color.good=purple
color.bad=yellow
allow.textmode=true
how.nice.to.look=fairlyNice
```

以下のコマンドを使用して、このディレクトリーの各ファイルの内容を保持する **ConfigMap** を作成できます。

```
$ oc create configmap game-config \
    --from-file=example-files/
```

--from-file オプションがディレクトリーを参照する場合、そのディレクトリーに直接含まれる各ファイルが **ConfigMap** でキーを設定するために使用されます。 このキーの名前はファイル名であり、キーの値はファイルの内容になります。

たとえば、上記のコマンドは以下の **ConfigMap** を作成します。

```
$ oc describe configmaps game-config
Name:           game-config
Namespace:      default
Labels:         <none>
Annotations:    <none>

Data

game.properties:        121 bytes
ui.properties:          83 bytes
```

マップにある 2 つのキーが、コマンドで指定されたディレクトリーのファイル名に基づいて作成されていることに気づかれることでしょう。それらのキーの内容のサイズは大きくなる可能性があるため、**oc describe** の出力はキーとキーのサイズのみを表示します。

キーの値を確認する必要がある場合は、オブジェクトに対して **oc get** をオプション **-o** を指定して実行できます。

```
$ oc get configmaps game-config -o yaml

apiVersion: v1
data:
  game.properties: |-
    enemies=aliens
    lives=3
    enemies.cheat=true
    enemies.cheat.level=noGoodRotten
    secret.code.passphrase=UUDDLRLRBABAS
    secret.code.allowed=true
    secret.code.lives=30
  ui.properties: |
    color.good=purple
    color.bad=yellow
    allow.textmode=true
    how.nice.to.look=fairlyNice
kind: ConfigMap
metadata:
  creationTimestamp: 2016-02-18T18:34:05Z
  name: game-config
  namespace: default
  resourceVersion: "407"-
  selflink: /api/v1/namespaces/default/configmaps/game-config
  uid: 30944725-d66e-11e5-8cd0-68f728db1985
```

22.2.2. ファイルからの作成

特定のファイルを指定して **--from-file** オプションを渡し、それを CLI に複数回渡すことができます。以下を実行すると、ディレクトリーからの作成の例と同等の結果を出すことができます。

1. 特定のファイルを指定して **ConfigMap** を作成します。

```
$ oc create configmap game-config-2 \
    --from-file=example-files/game.properties \
    --from-file=example-files/ui.properties
```

2. 結果を確認します。

```
$ oc get configmaps game-config-2 -o yaml

apiVersion: v1
data:
  game.properties: |-
    enemies=aliens
    lives=3
    enemies.cheat=true
    enemies.cheat.level=noGoodRotten
    secret.code.passphrase=UUDDLRLRBABAS
    secret.code.allowed=true
    secret.code.lives=30
  ui.properties: |
```

```
    color.good=purple
    color.bad=yellow
    allow.textmode=true
    how.nice.to.look=fairlyNice
kind: ConfigMap
metadata:
  creationTimestamp: 2016-02-18T18:52:05Z
  name: game-config-2
  namespace: default
  resourceVersion: "516"
  selflink: /api/v1/namespaces/default/configmaps/game-config-2
  uid: b4952dc3-d670-11e5-8cd0-68f728db1985
```

さらに **key=value** の式を渡して、個々のファイルに使用するキーを **--from-file** オプションで設定することができます。以下は例になります。

1. キーと値のペアを指定して **ConfigMap** を作成します。

```
$ oc create configmap game-config-3 \
    --from-file=game-special-key=example-files/game.properties
```

2. 結果を確認します。

```
$ oc get configmaps game-config-3 -o yaml

apiVersion: v1
data:
  game-special-key: |-
    enemies=aliens
    lives=3
    enemies.cheat=true
    enemies.cheat.level=noGoodRotten
    secret.code.passphrase=UUDDLRLRBABAS
    secret.code.allowed=true
    secret.code.lives=30
kind: ConfigMap
metadata:
  creationTimestamp: 2016-02-18T18:54:22Z
  name: game-config-3
  namespace: default
  resourceVersion: "530"
  selflink: /api/v1/namespaces/default/configmaps/game-config-3
  uid: 05f8da22-d671-11e5-8cd0-68f728db1985
```

22.2.3. リテラル値からの作成

ConfigMap にリテラル値を指定することもできます。**--from-literal** オプションは、リテラル値をコマンドラインに直接指定できる **key=value** 構文を取ります。

1. リテラル値を指定して **ConfigMap** を作成します。

```
$ oc create configmap special-config \
    --from-literal=special.how=very \
    --from-literal=special.type=charm
```

2. 結果を確認します。

```
$ oc get configmaps special-config -o yaml

apiVersion: v1
data:
  special.how: very
  special.type: charm
kind: ConfigMap
metadata:
  creationTimestamp: 2016-02-18T19:14:38Z
  name: special-config
  namespace: default
  resourceVersion: "651"
  selflink: /api/v1/namespaces/default/configmaps/special-config
  uid: dadce046-d673-11e5-8cd0-68f728db1985
```

22.3. ユースケース: POD での CONFIGMAP の使用

以下のセクションでは、Pod で **ConfigMap** オブジェクトを使用する際のいくつかのユースケースについて説明します。

22.3.1. 環境変数での使用

ConfigMaps は個別の環境変数を設定するために使用したり、有効な環境変数名を生成するすべてのキーで環境変数を設定したりできます。例として、以下の **ConfigMaps** について見てみましょう。

2 つの環境変数を含む ConfigMap

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: special-config (1)
  namespace: default
data:
  special.how: very (2)
  special.type: charm (3)
```

(1) ConfigMap の名前です。

(2) (3) 挿入する環境変数

1 つの環境変数を含む ConfigMap

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: env-config (1)
  namespace: default
data:
  log_level: INFO (2)
```

1. ConfigMap の名前です。
2. 注入する環境変数

configMapKeyRef セクションを使用して、Pod の **ConfigMap** のキーを使用できます。

特定の環境変数を挿入するように設定されている Pod 仕様のサンプル

```
apiVersion: v1
kind: Pod
metadata:
  name: dapi-test-pod
spec:
  containers:
    - name: test-container
      image: gcr.io/google_containers/busybox
      command: [ "/bin/sh", "-c", "env" ]
      env: 1
        - name: SPECIAL_LEVEL_KEY
          valueFrom:
            configMapKeyRef:
              name: special-config 2
              key: special.how 3
        - name: SPECIAL_TYPE_KEY
          valueFrom:
            configMapKeyRef:
              name: special-config 4
              key: special.type 5
              optional: true 6
      envFrom: 7
        - configMapRef:
            name: env-config 8
  restartPolicy: Never
```

1. **ConfigMap** から指定された環境変数をプルするためのスタンザです。
2. 4 特定の環境変数のプルに使用する **ConfigMap** の名前です。
3. 5 **ConfigMap** からプルする環境変数です。
6. 環境変数をオプションにします。オプションとして、Pod は指定された **ConfigMap** およびキーが存在しない場合でも起動します。
7. **ConfigMap** からすべての環境変数をプルするためのスタンザです。
8. すべての環境変数のプルに使用する **ConfigMap** の名前です。

この Pod が実行されると、その出力には以下の行が含まれます。

```
SPECIAL_LEVEL_KEY=very
log_level=INFO
```

22.3.2. コマンドライン引数の設定

ConfigMap は、コンテナーのコマンドまたは引数の値を設定するために使用することもできます。これは、Kubernetes 置換構文 **$(VAR_NAME)** を使用して実行できます。以下の **ConfigMaps** について見てみましょう。

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: special-config
  namespace: default
data:
  special.how: very
  special.type: charm
```

値をコマンドラインに挿入するには、「環境変数での使用」のユースケースで説明されているように環境変数として使用する必要のあるキーを使用する必要があります。次に、**$(VAR_NAME)** 構文を使用してコンテナーのコマンドでそれらを参照することができます。

特定の環境変数を挿入するように設定されている Pod 仕様のサンプル

```
apiVersion: v1
kind: Pod
metadata:
  name: dapi-test-pod
spec:
  containers:
    - name: test-container
      image: gcr.io/google_containers/busybox
      command: [ "/bin/sh", "-c", "echo $(SPECIAL_LEVEL_KEY) $(SPECIAL_TYPE_KEY)" ]
      env:
        - name: SPECIAL_LEVEL_KEY
          valueFrom:
            configMapKeyRef:
              name: special-config
              key: special.how
        - name: SPECIAL_TYPE_KEY
          valueFrom:
            configMapKeyRef:
              name: special-config
              key: special.type
  restartPolicy: Never
```

この Pod が実行されると、**test-container** コンテナーからの出力は以下のようになります。

```
very charm
```

22.3.3. ボリュームでの使用

ConfigMap はボリュームで使用することもできます。以下の **ConfigMap** の例に戻りましょう。

```
apiVersion: v1
kind: ConfigMap
metadata:
```

```
  name: special-config
  namespace: default
data:
  special.how: very
  special.type: charm
```

ボリュームでこの **ConfigMap** を使用する方法として 2 つの異なるオプションがあります。最も基本的な方法は、キーがファイル名であり、ファイルの内容がキーの値になっているファイルでボリュームを設定する方法です。

```
apiVersion: v1
kind: Pod
metadata:
  name: dapi-test-pod
spec:
  containers:
    - name: test-container
      image: gcr.io/google_containers/busybox
      command: [ "/bin/sh", "cat", "/etc/config/special.how" ]
      volumeMounts:
      - name: config-volume
        mountPath: /etc/config
  volumes:
    - name: config-volume
      configMap:
        name: special-config
  restartPolicy: Never
```

この Pod が実行されると出力は以下のようになります。

```
very
```

ConfigMap キーが展開されるボリューム内のパスを制御することもできます。

```
apiVersion: v1
kind: Pod
metadata:
  name: dapi-test-pod
spec:
  containers:
    - name: test-container
      image: gcr.io/google_containers/busybox
      command: [ "/bin/sh", "cat", "/etc/config/path/to/special-key" ]
      volumeMounts:
      - name: config-volume
        mountPath: /etc/config
  volumes:
    - name: config-volume
      configMap:
        name: special-config
        items:
        - key: special.how
          path: path/to/special-key
  restartPolicy: Never
```

この Pod が実行されると出力は以下のようになります。

```
very
```

22.4. REDIS の設定例

実際の使用例として、**ConfigMap** を使用して Redis を設定することができます。推奨の設定で Redis を挿入して Redis をキャッシュとして使用するには、Redis 設定ファイルに以下を含めるようにしてください。

```
maxmemory 2mb
maxmemory-policy allkeys-lru
```

設定ファイルが **example-files/redis/redis-config** にある場合、これを使って **ConfigMap** を作成します。

1. 設定ファイルを指定して **ConfigMap** を作成します。

```
$ oc create configmap example-redis-config \
    --from-file=example-files/redis/redis-config
```

2. 結果を確認します。

```
$ oc get configmap example-redis-config -o yaml

apiVersion: v1
data:
  redis-config: |
    maxmemory 2mb
    maxmemory-policy allkeys-lru
kind: ConfigMap
metadata:
  creationTimestamp: 2016-04-06T05:53:07Z
  name: example-redis-config
  namespace: default
  resourceVersion: "2985"
  selflink: /api/v1/namespaces/default/configmaps/example-redis-config
  uid: d65739c1-fbbb-11e5-8a72-68f728db1985
```

ここで、この **ConfigMap** を使用する Pod を作成します。

1. 以下のような Pod 定義を作成し、これを **redis-pod.yaml** などのファイルに保存します。

```
apiVersion: v1
kind: Pod
metadata:
  name: redis
spec:
  containers:
  - name: redis
    image: kubernetes/redis:v1
    env:
    - name: MASTER
      value: "true"
```

```
    ports:
    - containerPort: 6379
    resources:
      limits:
        cpu: "0.1"
    volumeMounts:
    - mountPath: /redis-master-data
      name: data
    - mountPath: /redis-master
      name: config
  volumes:
    - name: data
      emptyDir: {}
    - name: config
      configMap:
        name: example-redis-config
        items:
        - key: redis-config
          path: redis.conf
```

2. Pod を作成します。

```
$ oc create -f redis-pod.yaml
```

新規に作成された Pod には、**example-redis-config ConfigMap** の **redis-config** キーを **redis.conf** というファイルに置く **ConfigMap** ボリュームがあります。このボリュームは Redis コンテナーの **/redis-master** ディレクトリーにマウントされ、設定ファイルを **/redis-master/redis.conf** に配置します。 ここでイメージがマスターの Redis 設定ファイルを検索します。

この Pod に対して **oc exec** を実行し、**redis-cli** ツールを実行する場合、設定が正常に適用されたことを確認できます。

```
$ oc exec -it redis redis-cli
127.0.0.1:6379> CONFIG GET maxmemory
1) "maxmemory"
2) "2097152"
127.0.0.1:6379> CONFIG GET maxmemory-policy
1) "maxmemory-policy"
2) "allkeys-lru"
```

22.5. 制約

ConfigMap は、それらが Pod で使用される前に作成される必要があります。コントローラーは設定データが欠落している場合にもそれを容認するように作成できます。 個別のケースについては、**ConfigMap** で設定された個々のコンポーネントを確認してください。

ConfigMap オブジェクトはプロジェクトにあります。それらは同じプロジェクトの Pod によってのみ参照されます。

Kubelet は、API サーバーから取得する Pod の **ConfigMap** の使用のみをサポートします。これには、CLI を使用して作成された Pod、またはレプリケーションコントローラーから間接的に作成された Pod が含まれます。これには、OpenShift Container Platform ノードの **--manifest-url** フラグ、その **--config** フラグ、またはその REST API を使用して作成された Pod は含まれません (これらは Pod を作成する一般的な方法ではありません)。

第23章 DOWNWARD API

23.1. 概要

Downward API は、OpenShift Container Platform に結合せずにコンテナーが API オブジェクトについての情報を使用できるメカニズムです。この情報には、Pod の名前、namespace およびリソース値が含まれます。コンテナーは、環境変数またはボリュームプラグインを使用して Downward API から情報を使用できます。

23.2. フィールドの選択

Pod 内のフィールドは、**FieldRef** API タイプを使用して選択されます。**FieldRef** には 2 つのフィールドが含まれます。

フィールド	説明
fieldPath	Pod に関連して選択するフィールドのパスです。
apiVersion	**fieldPath** セレクターの解釈に使用する API バージョンです。

現時点で v1 API の有効なセレクターには以下が含まれます。

セレクター	説明
metadata.name	Pod の名前です。これは環境変数およびボリュームでサポートされています。
metadata.namespace	Pod の namespace です。 これは環境変数およびボリュームでサポートされています。
metadata.labels	Pod のラベルです。これはボリュームでのみサポートされ、環境変数ではサポートされていません。
metadata.annotations	Pod のアノテーションです。これはボリュームでのみサポートされ、環境変数ではサポートされていません。
status.podIP	Pod の IP です。これは環境変数でのみサポートされ、ボリュームではサポートされていません。

apiVersion フィールドです。 指定されていない場合は、対象の Pod テンプレートの API バージョンにデフォルト設定されます。

23.3. DOWNWARD API を使用したコンテナー値の使用

23.3.1. 環境変数の使用

Downward API を使用するための 1 つのメカニズムとして、コンテナーの環境変数を使用することができます。**EnvVar** タイプの **valueFrom** フィールド (タイプは **EnvVarSource**)は、変数の値が **value** フィールドで指定されるリテラル値ではなく、**FieldRef** ソースからの値になるように指定するために使用されます。今後は追加のソースがサポートされる可能性があります。 現時点では、ソースの **fieldRef** フィールドは Downward API からフィールドを選択するために使用されます。

この方法で使用できるのは Pod の定数属性のみです。環境変数を使用してサポートされるフィールドには、以下が含まれます。

- Pod の名前
- Pod の namespace

1. **pod.yaml** ファイルを作成します。

```
apiVersion: v1
kind: Pod
metadata:
  name: dapi-env-test-pod
spec:
  containers:
    - name: env-test-container
      image: gcr.io/google_containers/busybox
      command: [ "/bin/sh", "-c", "env" ]
      env:
        - name: MY_POD_NAME
          valueFrom:
            fieldRef:
              fieldPath: metadata.name
        - name: MY_POD_NAMESPACE
          valueFrom:
            fieldRef:
              fieldPath: metadata.namespace
  restartPolicy: Never
```

2. **pod.yaml** ファイルから Pod を作成します。

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

3. コンテナーのログで **MY_POD_NAME** および **MY_POD_NAMESPACE** の値を確認します。

```
$ oc logs -p dapi-env-test-pod
```

23.3.2. ボリュームプラグインの使用

Downward API を使用するもう 1 つのメカニズムとしてボリュームプラグインを使用することができます。Downward API ボリュームプラグインは、ファイルに展開される設定済みのフィールドを使ってボリュームを作成します。**VolumeSource** API オブジェクトの **metadata** フィールドはこのボリュームを設定するために使用されます。プラグインは以下のフィールドをサポートします。

- Pod の名前
- Pod の namespace

- Pod のアノテーション
- Pod のラベル

例23.1 Downward API ボリュームプラグイン設定

```
spec:
  volumes:
    - name: podinfo
      downwardAPI:: (1)
        items: (2)
          -name: "labels" (3)
           fieldRef:
             fieldPath: metadata.labels (4)
```

(1) ボリュームソースの **metadata** フィールドは Downward API ボリュームを設定します。

(2) **items** フィールドはボリュームに展開するフィールドの一覧を保持します。

(3) フィールドを展開するファイルの名前です。

(4) 展開するフィールドのセレクターです。

以下に例を示します。

1. **volume-pod.yaml** ファイルを作成します。

```
kind: Pod
apiVersion: v1
metadata:
  labels:
    zone: us-east-coast
    cluster: downward-api-test-cluster1
    rack: rack-123
  name: dapi-volume-test-pod
  annotations:
    annotation1: "345"
    annotation2: "456"
spec:
  containers:
    - name: volume-test-container
      image: gcr.io/google_containers/busybox
      command: ["sh", "-c", "cat /tmp/etc/pod_labels /tmp/etc/pod_annotations"]
      volumeMounts:
        - name: podinfo
          mountPath: /tmp/etc
          readOnly: false
  volumes:
  - name: podinfo
    downwardAPI:
      defaultMode: 420
      items:
      - fieldRef:
```

```
        fieldPath: metadata.name
      path: pod_name
    - fieldRef:
        fieldPath: metadata.namespace
      path: pod_namespace
    - fieldRef:
        fieldPath: metadata.labels
      path: pod_labels
    - fieldRef:
        fieldPath: metadata.annotations
      path: pod_annotations
  restartPolicy: Never
```

2. **volume-pod.yaml** ファイルから Pod を作成します。

```
$ oc create -f volume-pod.yaml
```

3. コンテナーのログを確認し、設定されたフィールドの有無を確認します。

```
$ oc logs -p dapi-volume-test-pod
cluster=downward-api-test-cluster1
rack=rack-123
zone=us-east-coast
annotation1=345
annotation2=456
kubernetes.io/config.source=api
```

23.4. DOWNWARD API を使用したコンテナーリソースの使用

Pod の作成時に、Downward API を使用してコンピューティングリソースの要求および制限についての情報を挿入し、イメージおよびアプリケーションの作成者が特定の環境用のイメージを適切に作成できるようにします。

これは、環境変数およびボリュームプラグインのいずれの方法で実行できます。

23.4.1. 環境変数の使用

1. Pod 設定の作成時に、**spec.container** フィールド内の **resources** フィールドの内容に対応する環境変数を指定します。

```
....
spec:
  containers:
    - name: test-container
      image: gcr.io/google_containers/busybox:1.24
      command: [ "/bin/sh", "-c", "env" ]
      resources:
        requests:
          memory: "32Mi"
          cpu: "125m"
        limits:
          memory: "64Mi"
          cpu: "250m"
      env:
```

```
        - name: MY_CPU_REQUEST
          valueFrom:
            resourceFieldRef:
              resource: requests.cpu
        - name: MY_CPU_LIMIT
          valueFrom:
            resourceFieldRef:
              resource: limits.cpu
        - name: MY_MEM_REQUEST
          valueFrom:
            resourceFieldRef:
              resource: requests.memory
        - name: MY_MEM_LIMIT
          valueFrom:
            resourceFieldRef:
              resource: limits.memory
....
```

リソース制限がコンテナー設定に含まれていない場合、Downward API はデフォルトでノードの CPU およびメモリーの割り当て可能な値に設定されます。

2. **pod.yaml** ファイルから Pod を作成します。

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

23.4.2. ボリュームプラグインの使用

1. Pod 設定の作成時に、**spec.volumes.downwardAPI.items** フィールドを使用して **spec.resources** フィールドに対応する必要なリソースを記述します。

```
....
spec:
  containers:
    - name: client-container
      image: gcr.io/google_containers/busybox:1.24
      command: ["sh", "-c", "while true; do echo; if [[ -e /etc/cpu_limit ]]; then cat /etc/cpu_limit;
fi; if [[ -e /etc/cpu_request ]]; then cat /etc/cpu_request; fi; if [[ -e /etc/mem_limit ]]; then cat
/etc/mem_limit; fi; if [[ -e /etc/mem_request ]]; then cat /etc/mem_request; fi; sleep 5; done"]
      resources:
        requests:
          memory: "32Mi"
          cpu: "125m"
        limits:
          memory: "64Mi"
          cpu: "250m"
      volumeMounts:
        - name: podinfo
          mountPath: /etc
          readOnly: false
  volumes:
    - name: podinfo
      downwardAPI:
        items:
          - path: "cpu_limit"
            resourceFieldRef:
```

```
            containerName: client-container
            resource: limits.cpu
          - path: "cpu_request"
            resourceFieldRef:
              containerName: client-container
              resource: requests.cpu
          - path: "mem_limit"
            resourceFieldRef:
              containerName: client-container
              resource: limits.memory
          - path: "mem_request"
            resourceFieldRef:
              containerName: client-container
              resource: requests.memory
....
```

リソース制限がコンテナー設定に含まれていない場合、Downward API はデフォルトでノードの CPU およびメモリーの割り当て可能な値に設定されます。

2. **volume-pod.yaml** ファイルから Pod を作成します。

```
$ oc create -f volume-pod.yaml
```

23.5. DOWNWARD API を使用したシークレットの使用

Pod の作成時に、Downward API を使用してシークレットを挿入し、イメージおよびアプリケーションの作成者が特定の環境用のイメージを作成できるようにできます。

23.5.1. 環境変数の使用

1. **secret.yaml** ファイルを作成します。

```
apiVersion: v1
kind: Secret
metadata:
  name: mysecret
data:
  password: cGFzc3dvcmQ=
  username: ZGV2ZWxvcGVy
type: kubernetes.io/basic-auth
```

2. secret.yaml ファイルから **Secret** を作成します。

```
oc create -f secret.yaml
```

3. 上記の **Secret** から **username** フィールドを参照する **pod.yaml** ファイルを作成します。

```
apiVersion: v1
kind: Pod
metadata:
  name: dapi-env-test-pod
spec:
  containers:
    - name: env-test-container
```

```
      image: gcr.io/google_containers/busybox
      command: [ "/bin/sh", "-c", "env" ]
      env:
        - name: MY_SECRET_USERNAME
          valueFrom:
            secretKeyRef:
              name: mysecret
              key: username
  restartPolicy: Never
```

4. **pod.yaml** ファイルから Pod を作成します。

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

5. コンテナーのログで **MY_SECRET_USERNAME** の値を確認します。

```
$ oc logs -p dapi-env-test-pod
```

23.6. DOWNWARD API を使用した CONFIGMAP の使用

Pod の作成時に、Downward API を使用して ConfigMap 値を挿入し、イメージおよびアプリケーションの作成者が特定の環境用のイメージを作成することができます。

23.6.1. 環境変数の使用

1. **configmap.yaml** ファイルを作成します。

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: myconfigmap
data:
  mykey: myvalue
```

2. **configmap.yaml** ファイルから **ConfigMap** を作成します。

```
oc create -f configmap.yaml
```

3. 上記の **ConfigMap** を参照する **pod.yaml** ファイルを作成します。

```
apiVersion: v1
kind: Pod
metadata:
  name: dapi-env-test-pod
spec:
  containers:
    - name: env-test-container
      image: gcr.io/google_containers/busybox
      command: [ "/bin/sh", "-c", "env" ]
      env:
        - name: MY_CONFIGMAP_VALUE
          valueFrom:
            configMapKeyRef:
```

```
            name: myconfigmap
            key: mykey
  restartPolicy: Never
```

4. **pod.yaml** ファイルから Pod を作成します。

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

5. コンテナーのログで **MY_CONFIGMAP_VALUE** の値を確認します。

```
$ oc logs -p dapi-env-test-pod
```

23.7. 環境変数の参照

Pod の作成時に、**$()** 構文を使用して事前に定義された環境変数の値を参照できます。環境変数の参照が解決されない場合、値は提供された文字列のままになります。

23.7.1. 環境変数の参照の使用

1. 既存の **environment variable** を参照する **pod.yaml** ファイルを作成します。

```
apiVersion: v1
kind: Pod
metadata:
  name: dapi-env-test-pod
spec:
  containers:
    - name: env-test-container
      image: gcr.io/google_containers/busybox
      command: [ "/bin/sh", "-c", "env" ]
      env:
        - name: MY_EXISTING_ENV
          value: my_value
        - name: MY_ENV_VAR_REF_ENV
          value: $(MY_EXISTING_ENV)
  restartPolicy: Never
```

2. **pod.yaml** ファイルから Pod を作成します。

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

3. コンテナーのログで **MY_ENV_VAR_REF_ENV** 値を確認します。

```
$ oc logs -p dapi-env-test-pod
```

23.7.2. 環境変数の参照のエスケープ

Pod の作成時に、二重ドル記号を使用して環境変数の参照をエスケープできます。次に値は指定された値の単一ドル記号のバージョンに設定されます。

1. 既存の **environment variable** を参照する **pod.yaml** ファイルを作成します。

```
apiVersion: v1
kind: Pod
metadata:
  name: dapi-env-test-pod
spec:
  containers:
    - name: env-test-container
      image: gcr.io/google_containers/busybox
      command: [ "/bin/sh", "-c", "env" ]
      env:
        - name: MY_NEW_ENV
          value: $$(SOME_OTHER_ENV)
  restartPolicy: Never
```

2. **pod.yaml** ファイルから Pod を作成します。

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

3. コンテナーのログで **MY_NEW_ENV** 値を確認します。

```
$ oc logs -p dapi-env-test-pod
```

第24章 PROJECTED ボリューム

24.1. 概要

Projected ボリューム は、いくつかの既存のボリュームソースを同じディレクトリーにマップします。

現時点で、以下のタイプのボリュームソースを展開できます。

- シークレット
- Config Map
- Downward API

注記

すべてのソースは Pod と同じ namespace に置かれる必要があります。

Projected ボリュームはこれらのボリュームソースの任意の組み合わせを単一ディレクトリーにマップし、ユーザーの以下の実行を可能にします。

- 単一ボリュームを、複数のシークレットのキー、configmap、および Downward API 情報で自動的に設定し、各種の情報ソースで単一ディレクトリーを合成できるようにします。
- 各項目のパスを明示的に指定して、単一ボリュームを複数シークレットのキー、configmap、および Downward API 情報で設定し、ユーザーがボリュームの内容を完全に制御できるようにします。

24.2. シナリオ例

以下の一般的なシナリオは、展開されるプロジェクトを使用する方法について示しています。

- **ConfigMap、シークレット、Downward API**Projected ボリュームを使用すると、パスワードが含まれる設定データでコンテナーをデプロイできます。これらのリソースを使用するアプリケーションは OpenStack を Kubernetes にデプロイする可能性があります。設定データは、サービスが実稼働用またはテストで使用されるかによって異なった方法でアセンブルされる必要がある可能性があります。Pod に実稼働またはテストのラベルが付けられている場合、Downward API セレクター **metadata.labels** を使用して適切な OpenStack 設定を生成できます。
- **ConfigMap + シークレット**Projected ボリュームにより、設定データおよびパスワードを使用してコンテナーをデプロイできます。たとえば、暗号化された機密タスクを Vault パスワードファイルを使用して復号化して、configmap として保存された Ansible Playbook を実行することができます。
- **ConfigMap + Downward API。** Projected ボリュームにより、Pod 名 (**metadata.name** セレクターで選択可能) を含む設定を生成できます。このアプリケーションは IP トラッキングを使用せずに簡単にソースを判別できるよう要求と共に Pod 名を渡すことができます。
- **シークレット + Downward API**Projected ボリュームにより、Pod の namespace (**metadata.namespace** セレクターで選択可能) を暗号化するためのパブリックキーとしてシークレットを使用できます。この例では、オペレーターはこのアプリケーションを使用し、暗号化されたトランスポートを使用せずに namespace 情報を安全に送信できるようになります。

24.3. POD 仕様の例

以下は、Projected ボリュームを作成するための Pod 仕様の例です。

例24.1 シークレット、Downward API および configmap を含む Pod

```
apiVersion: v1
kind: Pod
metadata:
  name: volume-test
spec:
  containers:
  - name: container-test
    image: busybox
    volumeMounts: 1
    - name: all-in-one
      mountPath: "/projected-volume" 2
      readOnly: true 3
  volumes: 4
  - name: all-in-one 5
    projected:
      defaultMode: 0400 6
      sources:
      - secret:
          name: mysecret 7
          items:
            - key: username
              path: my-group/my-username 8
      - downwardAPI: 9
          items:
            - path: "labels"
              fieldRef:
                fieldPath: metadata.labels
            - path: "cpu_limit"
              resourceFieldRef:
                containerName: container-test
                resource: limits.cpu
      - configMap: 10
          name: myconfigmap
          items:
            - key: config
              path: my-group/my-config
              mode: 0777 11
```

1. シークレットを必要とする各コンテナーの **volumeMounts** セクションを追加します。
2. シークレットが表示される未使用ディレクトリーのパスを指定します。
3. **readOnly** を **true** に設定します。
4. それぞれの Projected ボリュームソースを一覧表示するために **volumes** ブロックを追加します。
5. ボリュームの名前を指定します。

6 ファイルに実行パーミッションを設定します。

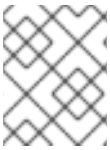
7 シークレットを追加します。シークレットオブジェクトの名前を追加します。使用する必要のあるそれぞれのシークレットは一覧表示される必要があります。

8 **mountPath** の下にシークレットへのパスを指定します。ここでシークレットファイルは **/projected-volume/my-group/my-config** に置かれます。

9 Downward API ソースを追加します。

10 ConfigMap ソースを追加します。

11 特定の展開におけるモードを設定します。

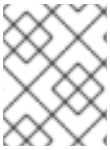


注記

Pod に複数のコンテナーがある場合、それぞれのコンテナーには **volumeMounts** セクションが必要ですが、1 つの **volumes** セクションのみが必要になります。

例24.2 デフォルト以外のパーミッションモデルが設定された複数シークレットを含む Pod

```
apiVersion: v1
kind: Pod
metadata:
  name: volume-test
spec:
  containers:
  - name: container-test
    image: busybox
    volumeMounts:
    - name: all-in-one
      mountPath: "/projected-volume"
      readOnly: true
  volumes:
  - name: all-in-one
    projected:
      defaultMode: 0755
      sources:
      - secret:
          name: mysecret
          items:
            - key: username
              path: my-group/my-username
      - secret:
          name: mysecret2
          items:
            - key: password
              path: my-group/my-password
              mode: 511
```



注記

defaultMode は展開されるレベルでのみ指定でき、各ボリュームソースには指定されません。ただし、上記のように個々の展開についての **mode** を明示的に指定できます。

24.4. パスについての留意事項

Projected ボリュームを作成する際に、ボリュームファイルのパスに関連する以下の状況について見てみましょう。

設定されるパスが同一である場合のキー間の競合

複数のキーを同じパスで設定する場合、Pod 仕様は有効な仕様として受け入れられません。以下の例では、**mysecret** および **myconfigmap** に指定されるパスは同じです。

```
apiVersion: v1
kind: Pod
metadata:
  name: volume-test
spec:
  containers:
  - name: container-test
    image: busybox
    volumeMounts:
    - name: all-in-one
      mountPath: "/projected-volume"
      readOnly: true
  volumes:
  - name: all-in-one
    projected:
      sources:
      - secret:
          name: mysecret
          items:
            - key: username
              path: my-group/data
      - configMap:
          name: myconfigmap
          items:
            - key: config
              path: my-group/data
```

設定されたパスのないキー間の競合

上記のシナリオの場合と同様に、実行時の検証が実行される唯一のタイミングはすべてのパスが Pod の作成時に認識される時です。それ以外の場合は、競合の発生時に指定された最新のリソースがこれより前のすべてのものを上書きします (これは Pod 作成後に更新されるリソースについても同様です)。

1つのパスが明示的なパスであり、もう1つのパスが自動的に展開されるパスである場合の競合

自動的に展開されるデータに一致するユーザー指定パスによって競合が生じる場合、前述のように後からのリソースがこれより前のすべてのものを上書きします。

24.5. POD の PROJECTED ボリュームの設定

以下の例は、既存のシークレットボリュームをマウントするために Projected ボリュームを使用する方法を示しています。

以下の手順は、ローカルファイルからユーザー名およびパスワードのシークレットを作成するために実行できます。その後に、シークレットを同じ共有ディレクトリーにマウントするために Projected ボリュームを使用して 1 つのコンテナーを実行する Pod を作成します。

1. シークレットが含まれるファイルを作成します。
   以下に例を示します。

   ```
   $ nano secret.yaml
   ```

   以下を入力し、パスワードおよびユーザー情報を適宜置き換えます。

   ```
   apiVersion: v1
   kind: Secret
   metadata:
     name: mysecret
   type: Opaque
   data:
     pass: MWYyZDFlMmU2N2Rm
     user: YWRtaW4=
   ```

   user および **pass** の値には、**base64** でエンコーティングされた任意の有効な文字列を使用できます。ここで使用される例は base64 でエンコーディングされた値 (**user: admin**、**pass:1f2d1e2e67df**) になります。

   ```
   $ echo -n "admin" | base64
   YWRtaW4=
   $ echo -n "1f2d1e2e67df" | base64
   MWYyZDFlMmU2N2Rm
   ```

2. 以下のコマンドを使用してシークレットを作成します。

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

   以下に例を示します。

   ```
   $ oc create -f secret.yaml
   secret "mysecret" created
   ```

3. シークレットが以下のコマンドを使用して作成されていることを確認できます。

   ```
   $ oc get secret <secret-name>
   $ oc get secret <secret-name> -o yaml
   ```

   以下に例を示します。

   ```
   $ oc get secret mysecret
   NAME       TYPE      DATA      AGE
   mysecret   Opaque    2         17h
   ```

   ```
   oc get secret mysecret -o yaml
   ```

```
apiVersion: v1
data:
  pass: MWYyZDFlMmU2N2Rm
  user: YWRtaW4=
kind: Secret
metadata:
  creationTimestamp: 2017-05-30T20:21:38Z
  name: mysecret
  namespace: default
  resourceVersion: "2107"
  selfLink: /api/v1/namespaces/default/secrets/mysecret
  uid: 959e0424-4575-11e7-9f97-fa163e4bd54c
type: Opaque
```

4. **volumes** セクションが含まれる 以下のような Pod 設定ファイルを作成します。

```
apiVersion: v1
kind: Pod
metadata:
  name: test-projected-volume
spec:
  containers:
  - name: test-projected-volume
    image: busybox
    args:
    - sleep
    - "86400"
    volumeMounts:
    - name: all-in-one
      mountPath: "/projected-volume"
      readOnly: true
  volumes:
  - name: all-in-one
    projected:
      sources:
      - secret:
          name: user
      - secret:
          name: pass
```

5. 設定ファイルから Pod を作成します。

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

以下に例を示します。

```
$ oc create -f secret-pod.yaml
pod "test-projected-volume" created
```

6. Pod コンテナーが実行中であることを確認してから、Pod への変更を確認します。

```
$ oc get pod <name>
```

出力は以下のようになります。

```
$ oc get pod test-projected-volume
NAME                    READY     STATUS    RESTARTS   AGE
test-projected-volume   1/1       Running   0          14s
```

7. 別のターミナルで、**oc exec** コマンド を使用して実行中のコンテナーへのシェルを開きます。

```
$ oc exec -it <pod> <command>
```

以下に例を示します。

```
$ oc exec -it test-projected-volume -- /bin/sh
```

8. シェルで、**projected-volumes** ディレクトリーに展開されるソースが含まれることを確認します。

```
/ # ls
bin              home             root             tmp
dev              proc             run              usr
etc              projected-volume  sys              var
```

第25章 DAEMONSET の使用

25.1. 概要

daemonset は、OpenShift Container Platform クラスター内の特定の、またはすべてのノードで Pod のレプリカを実行するために使用できます。

daemonset を使用して共有ストレージを作成し、クラスターのすべてのノードでロギング Pod を実行するか、またはすべてのノードでモニターエージェントをデプロイします。

セキュリティー上の理由から、 クラスター管理者のみが DeamonSet を作成できます。（ユーザーへの Daemonset パーミッションの付与。）

daemonset についての詳細は、Kubernetes ドキュメント を参照してください。

重要

Daemonset のスケジューリングにはプロジェクトのデフォルトノードセレクターとの互換性がありません。これを無効にしない場合、daemonset はデフォルトのノードセレクターとのマージによって制限されます。これにより、マージされたノードセレクターで選択解除されたノードで Pod が頻繁に再作成されるようになり、クラスターに不要な負荷が加わります。

そのため、以下に留意してください。

- daemonset の使用を開始する前に、namespace のアノテーション **openshift.io/node-selector** を空の文字列に設定することで、namespace のプロジェクト全体のデフォルトのノードセレクターを無効にします。

```
# oc patch namespace myproject -p \
    '{"metadata": {"annotations": {"openshift.io/node-selector": ""}}}'
```

- 新規プロジェクトを作成している場合、**oc adm new-project --node-selector=""** を使用してデフォルトのノードセレクターを上書きします。

25.2. DAEMONSET の作成

daemonset の作成時に、**nodeSelector** フィールドは daemonset がレプリカをデプロイする必要のあるノードを指定するために使用されます。

1. daemonset yaml ファイルを定義します。

```
apiVersion: extensions/v1beta1
kind: DaemonSet
metadata:
  name: hello-daemonset
spec:
  selector:
     matchLabels:
       name: hello-daemonset 1
  template:
    metadata:
      labels:
        name: hello-daemonset 2
```

```
spec:
  nodeSelector: 3
    type: infra
  containers:
  - image: openshift/hello-openshift
    imagePullPolicy: Always
    name: registry
    ports:
    - containerPort: 80
      protocol: TCP
    resources: {}
    terminationMessagePath: /dev/termination-log
  serviceAccount: default
  terminationGracePeriodSeconds: 10
```

1. daemonset に属する Pod を判別するラベルセレクターです。
2. Pod テンプレートのラベルセレクターです。上記のラベルセレクターに一致している必要があります。
3. Pod レプリカがデプロイされる必要のあるノードを判別するノードセレクターです。

2. daemonset オブジェクトを作成します。

```
oc create -f daemonset.yaml
```

3. Pod が作成されていることを確認し、各 Pod に Pod レプリカがあることを確認するには、以下を実行します。

   a. daemonset Pod を検索します。

```
$ oc get pods
hello-daemonset-cx6md   1/1       Running   0          2m
hello-daemonset-e3md9   1/1       Running   0          2m
```

   b. Pod がノードに配置されていることを確認するために Pod を表示します。

```
$ oc describe pod/hello-daemonset-cx6md|grep Node
Node:        openshift-node01.hostname.com/10.14.20.134
$ oc describe pod/hello-daemonset-e3md9|grep Node
Node:        openshift-node02.hostname.com/10.14.20.137
```

重要

- DaemonSet の Pod テンプレートを更新しても、既存の Pod レプリカには影響はありません。
- DaemonSet を削除してから、異なるテンプレートと同じラベルセレクターを使用して新規の DaemonSet を作成する場合に、既存の Pod レプリカを、ラベルが一致していると認識するため、既存の Pod レプリカは更新されず、Pod テンプレートで一致しない場合でも新しいレプリカが作成されます。
- ノードのラベルを変更する場合には、DaemonSet は新しいラベルと一致するノードに Pod を追加し、新しいラベルと一致しないノードから Pod を削除します。

DaemonSet を更新するには、以前のレプリカまたはノードを削除して新規の pod レプリカを強制的に作成します。

第26章 POD の自動スケーリング

26.1. 概要

HorizontalPodAutoscaler オブジェクトで定義される Horizontal Pod Autoscaler は、レプリケーションコントローラーに属する Pod から収集されるメトリクスまたはデプロイメント設定に基づいて、システムがレプリケーションコントローラーまたはデプロイメント設定のスケールの増減を自動的に設定する方法を指定します。

26.2. HORIZONTAL POD AUTOSCALER の要件

Horizontal Pod Autoscaler を使用するには、クラスター管理者は クラスターメトリクスを適切に設定 している必要があります。

26.3. サポートされるメトリクス

以下のメトリクスは Horizontal Pod Autoscaler でサポートされています。

表26.1 メトリクス

メトリクス	説明	API バージョン
CPU の使用率	要求される CPU のパーセンテージ	**autoscaling/v1、autoscaling/v2beta1**
メモリーの使用率	要求されるメモリーの割合	**autoscaling/v2beta1**

26.4. 自動スケーリング

oc autoscale コマンドを使用して Horizontal Pod Autoscaler を作成し、実行する Pod の最小数および最大数を指定すると共に Pod がターゲットとして設定すべき CPU の使用率またはメモリーの使用率を指定することができます。

重要

メモリー使用率の自動スケーリングはテクノロジープレビュー機能のみとして提供されています。

Horizontal Pod Autoscaler の作成後に、これは Heapster で Pod のメトリクスのクエリーを試行します。Heapster が初期メトリクスを取得するまでに 1 分から 2 分の時間がかかる場合があります。

メトリクスが Heapster で利用可能になると、Horizontal Pod Autoscaler は必要なメトリクスの使用率に対する現在のメトリクスの使用率の割合を計算し、随時スケールアップまたはスケールダウンを実行します。スケーリングは一定間隔で実行されますが、メトリクスが Heapster に移されるまでに 1 分から 2 分の時間がかかる場合があります。

レプリケーションコントローラーの場合、このスケーリングはレプリケーションコントローラーのレプリカに直接対応します。デプロイメント設定の場合、スケーリングはデプロイメント設定のレプリカ数に直接対応します。自動スケーリングは **Complete** フェーズの最新デプロイメントにのみ適用されることに注意してください。

OpenShift Container Platform はリソースに自動的に対応し、起動時などのリソースの使用が急増した場合など必要のない自動スケーリングを防ぎます。**unready** 状態の Pod には、スケールアップ時の使用率が **0 CPU** と指定され、Autoscaler はスケールダウン時にはこれらの Pod を無視します。既知のメトリクスのない Pod にはスケールアップ時の使用率が **0% CPU**、スケールダウン時に **100% CPU** となります。これにより、HPA の決定時に安定性が増します。この機能を使用するには、readiness チェックを設定して新規 Pod が使用可能であるかどうかを判別します。

26.5. CPU 使用率の自動スケーリング

oc autoscale コマンドを使用して、少なくとも指定される時間に実行する Pod の最大数を指定します。オプションとして Pod の最小数と Pod がターゲットとする平均の CPU 使用率を指定できます。指定しない場合は、OpenShift Container Platform サーバーからのデフォルト値がそれらに指定されます。

以下に例を示します。

```
$ oc autoscale dc/frontend --min 1 --max 10 --cpu-percent=80
deploymentconfig "frontend" autoscaled
```

上記の例では、Horizontal Pod Autoscaler の **autoscaling/v1** を使用して以下の定義で Horizontal Pod Autoscaler をする作成方法を示しています。

例26.1 Horizontal Pod Autoscaler オブジェクト定義

```
apiVersion: autoscaling/v1
kind: HorizontalPodAutoscaler
metadata:
  name: frontend (1)
spec:
  scaleTargetRef:
    kind: DeploymentConfig (2)
    name: frontend (3)
    apiVersion: apps/v1 (4)
    subresource: scale
  minReplicas: 1 (5)
  maxReplicas: 10 (6)
  targetCPUUtilizationPercentage: 80 (7)
```

(1) この Horizontal Pod Autoscaler オブジェクトの名前

(2) スケーリングするオブジェクトの種類

(3) スケーリングするオブジェクトの名前

(4) スケーリングするオブジェクトの API バージョン

(5) スケールダウン時のレプリカの最小数

(6) スケールアップ時のレプリカの最大数

(7) 各 Pod が使用していることが期待される要求された CPU のパーセンテージ

または、**oc autoscale** コマンドは Horizontal Pod Autoscaler の **v2beta1** バージョンを使用する際に以下の定義を使用して Horizontal Pod Autoscaler を作成します。

```
apiVersion: autoscaling/v2beta1
kind: HorizontalPodAutoscaler
metadata:
  name: hpa-resource-metrics-cpu (1)
spec:
  scaleTargetRef:
    apiVersion: apps/v1 (2)
    kind: ReplicationController (3)
    name: hello-hpa-cpu (4)
  minReplicas: 1 (5)
  maxReplicas: 10 (6)
  metrics:
  - type: Resource
    resource:
      name: cpu
      targetAverageUtilization: 50 (7)
```

1. この Horizontal Pod Autoscaler オブジェクトの名前
2. スケーリングするオブジェクトの API バージョン
3. スケーリングするオブジェクトの種類
4. スケーリングするオブジェクトの名前
5. スケールダウン時のレプリカの最小数
6. スケールアップ時のレプリカの最大数
7. 各 Pod が使用していることが予想される要求された CPU の平均のパーセンテージ

26.6. メモリー使用率の自動スケーリング

重要

メモリー使用率の自動スケーリングはテクノロジープレビュー機能のみとして提供されています。テクノロジープレビュー機能は Red Hat の実稼働環境でのサービスレベルアグリーメント (SLA) ではサポートされていないため、Red Hat では実稼働環境での使用を推奨していません。これらの機能は、近々発表予定の製品機能をリリースに先駆けてご提供することにより、開発プロセスの中でお客様に機能性のテストとフィードバックをしていただくことを目的としています。

Red Hat のテクノロジープレビュー機能のサポートについての詳細は、https://access.redhat.com/support/offerings/techpreview/を参照してください。

CPU ベースの自動スケーリングとは異なり、メモリーベースの自動スケーリングでは、**oc autoscale** コマンドの代わりに YAML を使用してAutoscaler を指定することが必要です。オプションとして、Pod の最小数および Pod がターゲットとする必要のある平均のメモリー使用率を指定できます。 これらを指定しない場合は、OpenShift Container Platform サーバーからのデフォルト値がこれらに指定されます。

1. メモリーベースの自動スケーリングは、自動スケーリング API の **v2beta1** バージョンでのみ利用できます。以下をクラスターの **master-config.yaml** ファイルに追加してメモリーベースの自動スケーリングを有効にします。

```
...
apiServerArguments:
  runtime-config:
  - apis/autoscaling/v2beta1=true
...
```

2. 以下を **hpa.yaml** などのファイルに置きます。

```
apiVersion: autoscaling/v2beta1
kind: HorizontalPodAutoscaler
metadata:
  name: hpa-resource-metrics-memory (1)
spec:
  scaleTargetRef:
    apiVersion: apps/v1 (2)
    kind: ReplicationController (3)
    name: hello-hpa-memory (4)
  minReplicas: 1 (5)
  maxReplicas: 10 (6)
  metrics:
  - type: Resource
    resource:
      name: memory
      targetAverageUtilization: 50 (7)
```

(1) この Horizontal Pod Autoscaler オブジェクトの名前

(2) スケーリングするオブジェクトの API バージョン

(3) スケーリングするオブジェクトの種類

(4) スケーリングするオブジェクトの名前

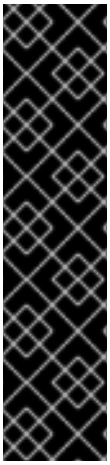
(5) スケールダウン時のレプリカの最小数

(6) スケールアップ時のレプリカの最大数

(7) 各 Pod が使用していることが予想される要求されたメモリーの平均のパーセンテージ

3. 次に上記のファイルから Autoscaler を作成します。

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



重要

メモリーベースの自動スケーリングを機能させるには、メモリー使用量がレプリカ数と比例して増減する必要があります。平均的には以下のようになります。

- レプリカ数が増えると、Pod ごとのメモリー (作業セット) の使用量が全体的に減少します。
- レプリカ数が減ると、Pod ごとのメモリー使用量が全体的に増加します。

OpenShift web コンソールを使用して、アプリケーションのメモリー動作を確認し、メモリーベースの自動スケーリングを使用する前にアプリケーションがそれらの要件を満たしていることを確認します。

26.7. HORIZONTAL POD AUTOSCALER の表示

Horizontal Pod Autoscaler のステータスを表示するには、以下を実行します。

- **oc get** コマンドを使用して CPU 使用率および Pod 制限の情報を表示します。

```
$ oc get hpa/hpa-resource-metrics-cpu
NAME                     REFERENCE                                TARGET    CURRENT  MINPODS
MAXPODS    AGE
hpa-resource-metrics-cpu    DeploymentConfig/default/frontend/scale   80%       79%      1
10         8d
```

出力には以下が含まれます。

  - **Target**。デプロイメント設定で制御されるすべての Pod でターゲットに設定された平均 CPU 使用率です。
  - **Current**。デプロイメント設定で制御されるすべての Pod における現在の CPU 使用率。
  - **Minpods/Maxpods**。Autoscaler で設定できるレプリカの最小数および最大数です。

- Horizontal Pod Autoscaler オブジェクトの詳細情報を参照するには、**oc describe** コマンドを使用します。

```
$ oc describe hpa/hpa-resource-metrics-cpu
Name:                           hpa-resource-metrics-cpu
Namespace:                      default
Labels:                         <none>
CreationTimestamp:              Mon, 26 Oct 2015 21:13:47 -0400
Reference:                      DeploymentConfig/default/frontend/scale
Target CPU utilization:         80% 1
Current CPU utilization:        79% 2
Min replicas:                   1 3
Max replicas:                   4 4
ReplicationController pods:     1 current / 1 desired
Conditions: 5
  Type              Status  Reason              Message
  ----              ------  ------              -------
  AbleToScale       True    ReadyForNewScale    the last scale time was sufficiently old
as to warrant a new scale
  ScalingActive     True    ValidMetricFound    the HPA was able to successfully calculate
```

```
a replica count from pods metric http_requests
  ScalingLimited       False   DesiredWithinRange      the desired replica count is within the
acceptable range
Events:
```

1. 各 Pod が使用していることが予想される要求されたメモリーの平均のパーセンテージ。
2. デプロイメント設定で制御されるすべての Pod における現在の CPU 使用率。
3. スケールダウン時のレプリカの最小数。
4. スケールアップ時のレプリカの最大数。
5. オブジェクトが **v2alpha1** API を使用している場合、状況条件が表示されます。

26.7.1. Horizontal Pod Autoscaler の状況条件の表示

状況条件セットを使用して、Horizontal Pod Autoscaler がスケーリングできるかどうかや、現時点でこれがいずれかの方法で制限されているかどうかを判別できます。

Horizontal Pod Autoscaler の状況条件は、自動スケーリング API の **v2beta1** バージョンで利用できます。

```
kubernetesMasterConfig:
  ...
  apiServerArguments:
    runtime-config:
    - apis/autoscaling/v2beta1=true
```

以下の状況条件が設定されます。

- **AbleToScale** は Horizontal Pod Autoscaler がスケールをフェッチし、更新できるかどうかや、いずれかのバックオフ条件がスケーリングを防いでいないかどうかを示します。
  - **True** 条件はスケーリングが許可されることを示します。
  - **False** 条件は指定される理由によりスケーリングが許可されないことを示します。
- **ScalingActive** は Horizontal Pod Autoscaler が有効にされており (ターゲットのレプリカ数がゼロでない)、必要なメトリクス (scale) を計算できるかどうかを示します。
  - **True** 条件はメトリクスが適切に機能していることを示します。
  - **False** 条件は通常フェッチするメトリクスに関する問題を示します。
- **ScalingLimited** は、レプリカの最大または最小数に達したために自動スケーリングが許可されないことを示します。
  - **True** 条件は、スケーリングするためにレプリカの最小または最大数を引き上げるか、または引き下げる必要があることを示します。
  - **False** 条件は、要求されたスケーリングが許可されることを示します。

この行を追加または編集する必要がある場合には、OpenShift Container Platform サービスを再起動します。

```
# systemctl restart atomic-openshift-master-api atomic-openshift-master-controllers
```

Horizontal Pod Autoscaler に影響を与える条件を表示するには、**oc describe hpa** を使用します。条件は **status.conditions** フィールドに表示されます。

```
$ oc describe hpa cm-test
Name:                         cm-test
Namespace:                    prom
Labels:                       <none>
Annotations:                  <none>
CreationTimestamp:            Fri, 16 Jun 2017 18:09:22 +0000
Reference:                    ReplicationController/cm-test
Metrics:                      ( current / target )
  "http_requests" on pods:    66m / 500m
Min replicas:                 1
Max replicas:                 4
ReplicationController pods:   1 current / 1 desired
Conditions: 1
  Type              Status  Reason              Message
  ----              ------  ------              -------
  AbleToScale       True    ReadyForNewScale    the last scale time was sufficiently old as to warrant
a new scale
  ScalingActive     True    ValidMetricFound    the HPA was able to successfully calculate a replica
count from pods metric http_request
  ScalingLimited    False   DesiredWithinRange  the desired replica count is within the acceptable
range
Events:
```

1 Horizontal Pod Autoscaler の状況メッセージです。

- **AbleToScale** 条件では、HPA がスケールを取得して更新できるか、またバックオフ関連の条件によりスケーリングを防止できるかどうかを指定します。
- **ScalingActive** の条件は、HPA を有効にするか (たとえば、ターゲットのレプリカ数は 0 でないなど)、任意のスケーリングを計算できるかどうかを指定します。「False」の状態は通常、メトリクスの取得に問題があることを示します。
- **ScalingLimited** の条件は、スケーリングが Horiontal Pod Autoscaler の最大値または最小値で制限されていることを示します。「**True**」の状態は通常、Horizontal Pod Autoscaler で最小または最大レプリカ数の制限の増減を実行する必要があることを示します。

以下は、スケーリングできない Pod の例です。

```
Conditions:
  Type          Status    Reason          Message
  ----          ------    ------          -------
  AbleToScale   False     FailedGetScale  the HPA controller was unable to get the target's current
scale: replicationcontrollers/scale.extensions "hello-hpa-cpu" not found
```

以下は、スケーリングに必要なメトリクスを取得できなかった Pod の例です。

```
Conditions:
  Type              Status    Reason              Message
  ----              ------    ------              -------
```

```
  AbleToScale           True    SucceededGetScale       the HPA controller was able to get the target's
current scale
  ScalingActive         False   FailedGetResourceMetric   the HPA was unable to compute the replica
count: unable to get metrics for resource cpu: no metrics returned from heapster
```

以下は、要求される自動スケーリングが要求される最小数よりも小さい場合の Pod の例です。

```
Conditions:
  Type            Status  Reason            Message
  ----            ------  ------            -------
  AbleToScale     True    ReadyForNewScale    the last scale time was sufficiently old as to warrant
a new scale
  ScalingActive   True    ValidMetricFound    the HPA was able to successfully calculate a replica
count from pods metric http_request
  ScalingLimited  False   DesiredWithinRange  the desired replica count is within the acceptable
range
Events:
```

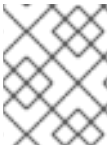
第27章 ボリュームの管理

27.1. 概要

コンテナーはデフォルトで永続性がある訳ではありません。再起動時にそれらのコンテンツはクリアされます。 ボリュームとは Pod およびコンテナーで利用可能なマウントされたファイルシステムのことであり、これらは数多くのホストのローカルまたはネットワーク割り当てストレージのエンドポイントでサポートされる場合があります。

ボリュームのファイルシステムにエラーが含まれないようにし、かつエラーが存在する場合はそれを修復するために、OpenShift Container Platform は **mount** ユーティリティーの前に **fsck** ユーティリティーを起動します。これはボリュームを追加するか、または既存ボリュームを更新する際に実行されます。

最も単純なボリュームタイプは **emptyDir** です。管理者はユーザーによる Pod に自動的に割り当てられる永続ボリュームの要求を許可することもできます。



注記

emptyDir ボリュームストレージは、FSGroup パラメーターがクラスター管理者によって有効にされている場合は Pod の FSGroup に基づいてクォータで制限できます。

CLI コマンドの **oc volume** を使用して、レプリケーションコントローラー や デプロイメント設定 などの Pod テンプレートを持つオブジェクトのボリュームおよびボリュームマウントを 追加 し、更新 し、または 削除 することができます。また、Pod または Pod テンプレートを持つオブジェクトのボリュームを一覧表示することもできます。

27.2. 一般的な CLI の使用方法

oc volume コマンドは以下の一般的な構文を使用します。

```
$ oc volume <object_selection> <operation> <mandatory_parameters> <optional_parameters>
```

このトピックでは、**<object_selection>** の **<object_type>/<name>** 形式を後に説明する例で使用しています。ただし、以下のオプションのいずれかを使用できます。

表27.1 オブジェクトの選択

構文	説明	例
<object_type> <name>	タイプ **<object_type>** の **<name>** を選択します。	**deploymentConfig registry**
<object_type>/<name>	タイプ **<object_type>** の **<name>** を選択します。	**deploymentConfig/registry**
<object_type> --selector=<object_label_selector>	所定のラベルセレクターに一致するタイプ **<object_type>** のリソースを選択します。	**deploymentConfig --selector="name=registry"**

構文	説明	例
<object_type> --all	タイプ **<object_type>** のすべてのリソースを選択します。	**deploymentConfig --all**
-f または **--filename=<file_name>**	リソースを編集するために使用するファイル名、ディレクトリー、または URL です。	**-f registry-deployment-config.json**

<operation> には、**--add**、**--remove**、または **--list** のいずれかを使用できます。

いずれの **<mandatory_parameters>** または **<optional_parameters>** も選択された操作に固有のものであり、これらについては後のセクションで説明します。

27.3. ボリュームの追加

ボリューム、ボリュームマウントまたはそれらの両方を Pod テンプレートに追加するには、以下を実行します。

```
$ oc volume <object_type>/<name> --add [options]
```

表27.2 ボリュームを追加するためのサポートされるオプション

オプション	詳細	Default
--name	ボリュームの名前。	指定がない場合は、自動的に生成されます。
-t, --type	ボリュームソースの名前。サポートされる値は **emptyDir**、**hostPath**、**secret**、**configmap**、**persistentVolumeClaim** または **projected** です。	**emptyDir**
-c, --containers	名前でコンテナーを選択します。すべての文字に一致するワイルドカード **'*'** を取ることもできます。	**'*'**
-m, --mount-path	選択されたコンテナー内のマウントパス。	
--path	ホストパス。**--type=hostPath** の必須パラメーターです。	
--secret-name	シークレットの名前。**--type=secret** の必須パラメーターです。	

オプション	詳細	Default
--configmap-name	configmap の名前。**--type=configmap** の必須のパラメーターです。	
--claim-name	永続ボリューム要求 (PVC) の名前。**--type=persistentVolumeClaim** の必須パラメーターです。	
--source	JSON 文字列としてのボリュームソースの詳細。必要なボリュームソースが **--type** でサポートされない場合に推奨されます。	
-o, --output	サーバー上で更新せずに変更したオブジェクトを表示します。サポートされる値は **json**、**yaml** です。	
--output-version	指定されたバージョンで変更されたオブジェクトを出力します。	**api-version**

例
新規ボリュームソース **emptyDir** をデプロイメント設定の **レジストリー** に追加します。

```
$ oc volume dc/registry --add
```

レプリケーションコントローラー **r1** のシークレット **$ecret** を使用してボリューム **v1** を追加し、コンテナー内の **/data** でマウントします。

```
$ oc volume rc/r1 --add --name=v1 --type=secret --secret-name='$ecret' --mount-path=/data
```

要求名 **pvc1** を使って既存の永続ボリューム **v1** をディスク上のデプロイメント設定 **dc.json** に追加し、ボリュームをコンテナー **c1** の **/data** でマウントし、サーバー上でデプロイメント設定を更新します。

```
$ oc volume -f dc.json --add --name=v1 --type=persistentVolumeClaim \
  --claim-name=pvc1 --mount-path=/data --containers=c1
```

すべてのレプリケーションコントローラーについてリビジョン **5125c45f9f563** を使い、Git リポジトリー **https://github.com/namespace1/project1** に基づいてボリューム **v1** を追加します。

```
$ oc volume rc --all --add --name=v1 \
  --source='{"gitRepo": {
              "repository": "https://github.com/namespace1/project1",
              "revision": "5125c45f9f563"
            }}'
```

27.4. ボリュームの更新

既存のボリュームまたはボリュームマウントを更新することは、ボリュームの追加と同様ですが、**--overwrite** オプションを使用します。

```
$ oc volume <object_type>/<name> --add --overwrite [options]
```

例
レプリケーションコントローラーの既存ボリューム **r1** の既存のボリューム **v1** を 既存の Persistent Volume Claim (永続ボリューム要求、PVC) **pvc1** に置き換えます。

```
$ oc volume rc/r1 --add --overwrite --name=v1 --type=persistentVolumeClaim --claim-name=pvc1
```

デプロイメント設定 **d1** のマウントポイントをボリューム **v1** の **/opt** に変更します。

```
$ oc volume dc/d1 --add --overwrite --name=v1 --mount-path=/opt
```

27.5. ボリュームの削除

Pod テンプレートからボリュームまたはボリュームマウントを削除するには、以下を実行します。

```
$ oc volume <object_type>/<name> --remove [options]
```

表27.3 ボリュームを削除するためにサポートされるオプション

オプション	詳細	Default
--name	ボリュームの名前。	
-c, --containers	名前でコンテナーを選択します。すべての文字に一致するワイルドカード **'*'** を取ることもできます。	**'*'**
--confirm	複数のボリュームを 1 度に削除することを示します。	
-o, --output	サーバー上で更新せずに変更したオブジェクトを表示します。サポートされる値は **json**、**yaml** です。	
--output-version	指定されたバージョンで変更されたオブジェクトを出力します。	**api-version**

例
デプロイメント設定 **d1** からボリューム **v1** を削除します。

```
$ oc volume dc/d1 --remove --name=v1
```

デプロイメント設定 **d1** のコンテナー **c1** からボリューム **v1** をアンマウントし、**d1** のコンテナーで参照されていない場合はボリューム **v1** を削除します。

```
$ oc volume dc/d1 --remove --name=v1 --containers=c1
```

レプリケーションコントローラー **r1** のすべてのボリュームを削除します。

```
$ oc volume rc/r1 --remove --confirm
```

27.6. ボリュームの一覧表示

Pod または Pod テンプレートのボリュームまたはボリュームマウントを一覧表示するには、以下を実行します。

```
$ oc volume <object_type>/<name> --list [options]
```

ボリュームのサポートされているオプションを一覧表示します。

オプション	詳細	Default
--name	ボリュームの名前。	
-c, --containers	名前でコンテナーを選択します。すべての文字に一致するワイルドカード **'*'** を取ることもできます。	**'*'**

例
Pod **p1** のすべてのボリュームを一覧表示します。

```
$ oc volume pod/p1 --list
```

すべてのデプロイメント設定で定義されるボリューム **v1** を一覧表示します。

```
$ oc volume dc --all --name=v1
```

27.7. サブパスの指定

volumeMounts.subPath プロパティを使用して、ボリュームのルートではなく、ボリューム内の**サブパス**を指定します。**subPath** を使用すると、1 つの Pod でボリュームを共有して複数の用途に使用できます。

ボリューム内のファイルの一覧を表示するには、**oc rsh** コマンドを実行します。

```
$ oc rsh <pod>
sh-4.2$ ls /path/to/volume/subpath/mount
example_file1 example_file2 example_file3
```

subPath を指定します。

subPath の使用例

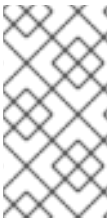
```
apiVersion: v1
kind: Pod
metadata:
  name: my-site
spec:
    containers:
    - name: mysql
      image: mysql
      volumeMounts:
      - mountPath: /var/lib/mysql
        name: site-data
        subPath: mysql (1)
    - name: php
      image: php
      volumeMounts:
      - mountPath: /var/www/html
        name: site-data
        subPath: html (2)
    volumes:
    - name: site-data
      persistentVolumeClaim:
        claimName: my-site-data
```

1. データベースは **mysql** フォルダーに保存されます。
2. HTML コンテンツは **html** フォルダーに保存されます。

第28章 永続ボリュームの使用

28.1. 概要

PersistentVolume オブジェクトは OpenShift Container Platform クラスターのストレージリソースです。ストレージは、クラスター管理者が **PersistentVolume** オブジェクトを GCE Persistent Disk、AWS Elastic Block Store (EBS) および NFS マウントなどのソースから作成することによってプロビジョニングできます。



注記

『 Installation and Configuration Guide 』では、NFS、GlusterFS、Ceph RBD、OpenStack Cinder、AWS EBS、GCE Persistent Disk、iSCSI、および Fibre Channel を使用して、永続ストレージを OpenShift Container Platform クラスターにプロビジョニングする方法についてのクラスター管理者向けの手順を説明しています。

ストレージは、リソースの要求 (claim) を指定することにより利用可能になります。ストレージリソースの要求は **PersistentVolumeClaim** オブジェクトを使用して実行できます。 この要求 (claim) は、通常は要求する内容に一致するボリュームとペアになります。

28.2. ストレージの要求

ストレージの要求は、プロジェクトで **PersistentVolumeClaim** オブジェクトを作成して実行できます。

Persistent Volume Claim (永続ボリューム要求、PVC) オブジェクト定義

```
apiVersion: "v1"
kind: "PersistentVolumeClaim"
metadata:
  name: "claim1"
spec:
  accessModes:
    - "ReadWriteOnce"
  resources:
    requests:
      storage: "1Gi"
  volumeName: "pv0001"
```

28.3. ボリュームと要求のバインディング

PersistentVolume は固有のリソースです。**PersistentVolumeClaim** はストレージサイズなどの特定の属性を持つリソースの要求です。これら 2 者の間には、要求を利用可能なボリュームに一致させ、要求とボリュームをバインドするプロセスがあります。これにより、要求は Pod のボリュームとして使用できます。OpenShift Container Platform は要求をサポートするボリュームを検索し、これを Pod にマウントします。

CLI を使用してクエリーを実行し、要求またはボリュームがバインドされているかどうかを判別できます。

```
$ oc get pvc
NAME      LABELS    STATUS    VOLUME
```

```
claim1      map[]     Bound     pv0001

$ oc get pv
NAME            LABELS          CAPACITY          ACCESSMODES       STATUS    CLAIM
pv0001          map[]           5368709120        RWO               Bound     yournamespace / claim1
```

28.4. POD のボリュームとしての要求

PersistentVolumeClaim は Pod によってボリュームとして使用されます。OpenShift Container Platform は Pod と同じ namespace の指定された名前で要求を検索し、この要求を使用してこれに対応するマウントするボリュームを検索します。

要求を含む Pod の定義

```
apiVersion: "v1"
kind: "Pod"
metadata:
  name: "mypod"
  labels:
    name: "frontendhttp"
spec:
  containers:
    -
      name: "myfrontend"
      image: openshift/hello-openshift
      ports:
        -
          containerPort: 80
          name: "http-server"
      volumeMounts:
        -
          mountPath: "/var/www/html"
          name: "pvol"
  volumes:
    -
      name: "pvol"
      persistentVolumeClaim:
        claimName: "claim1"
```

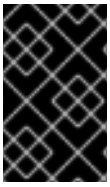
28.5. ボリュームと要求の事前バインディング

PersistentVolumeClaim をバインドする **PersistentVolume** を正確に把握している場合、**volumeName** フィールドを使用して PV を PVC に指定できます。この方法は、通常のマッチングおよびバインディングプロセスを省略します。PVC は **volumeName** に指定される同じ名前を持つ PV にのみバインドできます。この名前を持つ PV が存在し、**Available** の場合、PV と PVC は、PV が PVC のラベルセレクター、アクセスモードおよびリソース要求を満たすかどうかに関係なくバインドされます。

例28.1 volumeName のある Persistent Volume Claim (永続ボリューム要求、PVC) オブジェクト定義

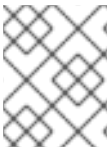
```
apiVersion: "v1"
kind: "PersistentVolumeClaim"
metadata:
```

```
    name: "claim1"
  spec:
    accessModes:
      - "ReadWriteOnce"
    resources:
      requests:
        storage: "1Gi"
    volumeName: "pv0001"
```



重要

claimRefs を設定する機能は、説明されているユースケースにおける一時的な回避策です。ボリュームを要求できるユーザーを制限する長期的なソリューションは開発中です。



注記

クラスター管理者は、ユーザーの代わりに **claimRefs** を設定する前に セレクターとラベルによるボリュームのバインディング を設定することをまず検討する必要があります。

クラスター管理者が独自の要求に対してのみにボリュームを「予約」し、それ以外の要求がその独自の要求がボリュームにバインドされる前にバインドされないようにすることもできます。この場合、管理者は **claimRef** フィールドを使用して PVC を PV に指定できます。PV は **claimRef** で指定される同じ名前および namesapce を持つ PVC にのみバインドできます。PVC のアクセスモードおよびリソース要求の条件は、ラベルセレクターが無視される場合でも PV および PVC がバインドされるために満たされる必要があります。

claimRef での永続ボリュームオブジェクト定義

```
apiVersion: v1
kind: PersistentVolume
metadata:
  name: pv0001
spec:
  capacity:
    storage: 1Gi
  accessModes:
  - ReadWriteOnce
  nfs:
    path: /tmp
    server: 172.17.0.2
  persistentVolumeReclaimPolicy: Recycle
  claimRef:
    name: claim1
    namespace: default
```

volumeName を PVC に指定しても、その PVC がバインドされる前に異なる PVC が指定された PV にバインドされることを防ぐ訳ではありません。要求は PV が **Available** になるまで **Pending** のままになります。

claimRef を PV に指定しても、指定された PVC が異なる PV にバインドされることを防ぐ訳ではありません。PVC は通常のバインディングプロセスに基づいてバインドする別の PV を自由に選択できます。そのため、これらのシナリオを避け、要求が必要なボリュームにバインドされるようにするに

は、**volumeName** および **claimRef** の両方が指定される必要があります。

pv.kubernetes.io/bound-by-controller アノテーションについて **Bound** PV および PVC ペアを検査することにより、**volumeName** および/または **claimRef** の設定のマッチングおよびバインディングプロセスへの影響を確認できます。**volumeName** および/または **claimRef** を独自に設定した PV および PVC にはこのアノテーションがありませんが、通常の PV および PVC ではこれは **"yes"** に設定されています。

PV の **claimRef** が一部の PVC 名および namespace に設定されていて、PV が **Retain** または **Recycle** 回収ポリシーに応じて回収されている場合、その **claimRef** は PVC または namespace 全体が存在しなくなっても同じ PVC 名および namespace に設定されたままになります。

第29章 永続ボリュームの拡張

29.1. PERSISTENT VOLUME CLAIM (永続ボリューム要求、PVC) の拡張を有効化

ボリューム拡張はテクノロジープレビュー機能であるため、OpenShift Container Platform 3.9 クラスターではデフォルトで有効にされません。OpenShift Container Platform 管理者がこの機能を特定のユースケースで有効にしたい理由が他にもある可能性があります。

注記

Red Hat のテクノロジープレビュー機能のサポートについての詳細は、https://access.redhat.com/support/offerings/techpreview/を参照してください。

OpenShift Container Platform ユーザーによる Persistent Volume Claim (永続ボリューム要求、PVC) の拡張を可能にするには、OpenShift Container Platform 管理者は、**allowVolumeExpansion** を **true** に設定して StorageClass を作成するか、または更新する必要があります。このクラスから作成された PVC のみを拡張することができます。

これとは別に、OpenShift Container Platform 管理者は **ExpandPersistentVolumes** 機能フラグを有効にし、**PersistentVolumeClaimResize** 受付コントローラーをオンにする必要があります。**PersistentVolumeClaimResize** 受付 コントローラー についての詳細は、「受付コントローラー」を参照してください。

機能ゲートを有効にするには、システム全体で **ExpandPersistentVolumes** を **true** に設定します。

1. クラスターのすべてのノードで node-config.yaml を設定します。

```
# cat /etc/origin/node/node-config.yaml
...
kubeletArguments:
...
  feature-gates:
  - ExpandPersistentVolumes=true
# systemctl restart atomic-openshift-node
```

2. マスター API およびコントローラーマネージャーで **ExpandPersistentVolumes** 機能ゲートを有効にします。

```
# cat /etc/origin/master/master-config.yaml
...
kubernetesMasterConfig:
  apiServerArguments:
  ...
    feature-gates:
    - ExpandPersistentVolumes=true
  controllerArguments:
    ...
    feature-gates:
    - ExpandPersistentVolumes=true

# systemctl restart atomic-openshift-master-api
```

29.2. GLUSTERFS ベースの PERSISTENT VOLUME CLAIM (永続ボリューム要求、PVC) の拡張

OpenShift Container Platform 管理者が **allowVolumeExpansion** を **true** に設定して StorageClass を作成すると、そのクラスから PVC を作成でき、以降は必要に応じてその PVC を編集し、新規サイズを要求できます。

以下に例を示します。

```
kind: PersistentVolumeClaim
apiVersion: v1
metadata:
  name: gluster-mysql
spec:
  storageClass: "storageClassWithFlagSet"
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: 8Gi (1)
```

(1) **spec.resources.requests** を更新して拡張されたボリュームを要求できます。

29.3. ファイルシステムを搭載した PERSISTENT VOLUME CLAIM (永続ボリューム要求、PVC) の拡張

ファイルサイズのサイズ変更を必要とするボリュームタイプ (GCE PD、EBS、および Cinder など) に基づいて PVC を拡張するには 2 つの手順からなるプロセスが必要です。通常このプロセスでは、CloudProvider でボリュームオブジェクトを拡張してから実際のノードでファイルシステムを拡張します。

ノードでのファイルシステムの拡張は、新規 Pod がボリュームと共に起動する場合にのみ実行されます。

以下のプロセスは、 **allowVolumeExpansion** が **true** に設定された状態で PVC が StorageClass から作成されていることを前提としています。

1. **spec.resources.requests** を編集して PVC を編集し、新規サイズを要求します。CloudProvider オブジェクトのサイズ変更が終了すると、PVC は **FileSystemResizePending** に設定されます。

2. 条件を確認するために以下のコマンドを入力します。

```
oc describe pvc <pvc_name>
```

CloudProvider オブジェクトのサイズ変更が終了すると、永続ボリューム (PV) オブジェクトは **PersistentVolume.Spec.Capacity** に新規に要求されたサイズを反映します。この時点で、PVC から新規 Pod を作成または再作成してファイルシステムのサイズ変更を終了することができます。Pod が実行されている場合、新たに要求されたサイズが利用可能になり、**FileSystemResizePending** 条件が PVC から削除されます。

29.4. ボリューム拡張時に障害からの復旧

マスターまたはノードで基礎となるストレージの拡張に失敗した場合に、OpenShift Container Platform の管理者は手動で PVC の状態を復旧し、管理者の介入なしにコントローラーによって継続的に再試行されるリサイズ要求を取り消します。

現時点で、これは以下の手順によって手動で実行できます。

1. **Retain** 回収ポリシーで要求 (PVC) にバインドされている PV をマークします。これは、PV を編集し、**persistentVolumeReclaimPolicy** を **Retain** に変更することで実行できます。

2. PVC を削除します (後ほど再作成されます)。

3. 新規作成された PVC が **Retain** とマークされた PV にバインドするには、手動で PV を編集し、PV 仕様から **claimRef** エントリーを削除します。これで、PV は **Available** とマークされます。PVC の事前バインドに関する情報は、「ボリュームと要求の事前バインド」を参照してください。

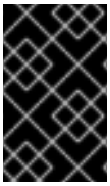
4. 小さ目のサイズか、または基礎となるストレージプロバイダーによって割り当て可能なサイズで PVC を再作成します。また、PVC の **volumeName** フィールドを PV の名前に設定します。これにより、PVC はプロビジョニングされた PV のみにバインドされます。

5. PV で回収ポリシーを復元します。

第30章 リモートコマンドの実行

30.1. 概要

CLI を使用してコンテナーでリモートコマンドを実行できます。これにより、コンテナーでルーチン操作を実行するための一般的な Linux コマンドを実行できます。



重要

セキュリティー保護の理由により、**oc exec** コマンドは、コマンドが **cluster-admin** ユーザーによって実行されている場合を除き、特権付きコンテナーにアクセスしようとしても機能しません。詳細は、「CLI 操作」のトピック を参照してください。

30.2. 基本的な使用方法

リモートコンテナーコマンド実行のサポートは CLI に組み込まれて います。

```
$ oc exec <pod> [-c <container>] <command> [<arg_1> ... <arg_n>]
```

以下に例を示します。

```
$ oc exec mypod date
Thu Apr  9 02:21:53 UTC 2015
```

30.3. プロトコル

クライアントは要求を Kubernetes API サーバーに対して実行してコンテナーのリモートコマンドの実行を開始します。

```
/proxy/minions/<node_name>/exec/<namespace>/<pod>/<container>?command=<command>
```

上記の URL には以下が含まれます。

- **<node_name>** はノードの FQDN です。
- **<namespace>** はターゲット Pod の namespace です。
- **<pod>** はターゲット Pod の名前です。
- **<container>** はターゲットコンテナーの名前です。
- **<command>** は実行される必要なコマンドです。

以下は例になります。

```
/proxy/minions/node123.openshift.com/exec/myns/mypod/mycontainer?command=date
```

さらに、クライアントはパラメーターを要求に追加して以下について指示します。

- クライアントはリモートクライアントのコマンドに入力を送信する (標準入力: stdin)。
- クライアントのターミナルは TTY である。

- リモートコンテナーのコマンドは標準出力 (stdout) からクライアントに出力を送信する。
- リモートコンテナーのコマンドは標準エラー出力 (stderr) からクライアントに出力を送信する。

exec 要求の API サーバーへの送信後、クライアントは多重化ストリームをサポートするものに接続をアップグレードします。 現在の実装では **SPDY** を使用しています。

クライアントは標準入力 (stdin)、標準出力 (stdout)、および標準エラー出力 (stderr) 用にそれぞれのストリームを作成します。ストリームを区別するために、クライアントはストリームの **streamType** ヘッダーを **stdin**、**stdout**、または **stderr** のいずれかに設定します。

リモートコマンド実行要求の処理が終了すると、クライアントはすべてのストリームやアップグレードされた接続および基礎となる接続を閉じます。

注記

詳細については、『 Architecture 』ガイドを参照してください。

第31章 ファイルのコンテナーから/へのコピー

31.1. 概要

CLI を使用してコンテナーのリモートディレクトリーに/からローカルファイルをコピーできます。これは、バックアップと復元を実行するためにデータベースアーカイブを Pod にコピーし、Pod からコピーするのに役立つツールです。また、実行中の Pod がソースファイルのホットリロードをサポートする場合に、ソースコードの変更を開発のデバッグ目的で実行中の Pod にコピーするために使用できます。

31.2. 基本的な使用方法

ローカルファイルをコンテナーから/にコピーするためのサポートは、CLI に組み込まれて います。

```
$ oc rsync <source> <destination> [-c <container>]
```

たとえば、ローカルディレクトリーを Pod ディレクトリーにコピーするには、以下を実行します。

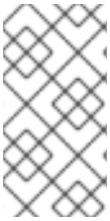
```
$ oc rsync /home/user/source devpod1234:/src
```

または、Pod ディレクトリーをローカルディレクトリーにコピーするには、以下を実行します。

```
$ oc rsync devpod1234:/src /home/user/source
```

31.3. データベースのバックアップおよび復元

oc rsync を使用して、データベースアーカイブを既存のデータベースコンテナーから新規データベースコンテナーの永続ボリュームディレクトリーにコピーします。



注記

MySQL は以下の例で使用されています。**mysql|MYSQL** を **pgsql|PGSQL** または **mongodb|MONGODB** に置き換え、移行ガイド を参照してサポートされているデータベースイメージに対応するコマンドを確認してください。この例では既存のデータベースコンテナーを使用していることを前提としています。

1. 実行中のデータベース Pod から既存のデータベースをバックアップします。

```
$ oc rsh <existing db container>
# mkdir /var/lib/mysql/data/db_archive_dir
# mysqldump --skip-lock-tables -h ${MYSQL_SERVICE_HOST} -P
${MYSQL_SERVICE_PORT:-3306} \
  -u ${MYSQL_USER} --password="$MYSQL_PASSWORD" --all-databases >
/var/lib/mysql/data/db_archive_dir/all.sql
# exit
```

2. ローカルマシンに対してアーカイブファイルのリモート同期を実行します。

```
$ oc rsync <existing db container with db archive>:/var/lib/mysql/data/db_archive_dir /tmp/.
```

3. 上記で作成されたデータベースアーカイブを読み込む 2 つ目の MySQL Pod を起動します。MySQL Pod には固有の **DATABASE_SERVICE_NAME** がなければなりません。

```
$ oc new-app mysql-persistent \
  -p MYSQL_USER=<archived mysql username> \
  -p MYSQL_PASSWORD=<archived mysql password> \
  -p MYSQL_DATABASE=<archived database name> \
  -p DATABASE_SERVICE_NAME='mysql2' 1
$ oc rsync /tmp/db_archive_dir new_dbpod1234:/var/lib/mysql/data
$ oc rsh new_dbpod1234
```

1 **mysql** はデフォルトです。この例では **mysql2** が作成されます。

4. 適切なコマンドを使用してコピーされたデータベースアーカイブディレクトリーから新規のデータベースコンテナーにデータベースを復元します。

MySQL

```
$ cd /var/lib/mysql/data/db_archive_dir
$ mysql -u root
$ source all.sql
$ GRANT ALL PRIVILEGES ON <dbname>.* TO '<your username>'@'localhost'; FLUSH
PRIVILEGES;
$ cd ../; rm -rf /var/lib/mysql/data/db_backup_dir
```

これで、アーカイブされたデータベースを使って 2 つの MySQL データベース Pod がプロジェクトで実行されていることになります。

31.4. 要件

oc rsync コマンドは、クライアントのマシンでローカルの **rsync** コマンドを使用します。この場合、リモートコンテナーにも **rsync** コマンドがあることが必要になります。

rsync がローカルまたはリモートコンテナーにない場合、tar アーカイブがローカルに作成され、**tar** がファイルを展開するために使用されるコンテナーに送信されます。**tar** がリモートコンテナーで利用できない場合にコピーは失敗します。

tar のコピー方法は **rsync** と同様に機能する訳ではありません。たとえば、**rsync** は宛先ディレクトリーを作成し (存在しない場合)、ソースと宛先間の差分のファイルのみを送信します。

注記

Windows では、**cwRsync** クライアントが **oc rsync** コマンドで使用するためにインストールされ、PATH に追加される必要があります。

31.5. COPY SOURCE の指定

oc rsync コマンドのソース引数はローカルディレクトリーまた Pod ディレクトリーのいずれかを参照する必要があります。個々のファイルは現時点ではサポートされていません。

Pod ディレクトリーを指定する場合、ディレクトリー名の前に Pod 名を付ける必要があります。

```
<pod name>:<dir>
```

標準の **rsync** の場合と同様に、ディレクトリー名がパスセパレーター (/) で終了する場合、ディレクトリーの内容のみが宛先にコピーされます。そうでない場合は、ディレクトリー自体がその内容すべてと共に宛先にコピーされます。

31.6. COPY DESTINATION の指定

oc rsync コマンドの宛先引数はディレクトリーを参照する必要があります。ディレクトリーが存在せず、**rsync** がコピーに使用される場合、ディレクトリーが作成されます。

31.7. 宛先でのファイルの削除

--delete フラグは、ローカルディレクトリーにないリモートディレクトリーにあるファイルを削除するために使用できます。

31.8. ファイル変更についての継続的な同期

--watch オプションを使用すると、コマンドはソースパスでファイルシステムの変更をモニターし、変更が生じるとそれらを同期します。この引数を指定すると、コマンドは無期限に実行されます。

同期は短い非表示期間の後に実行され、急速に変化するファイルシステムによって同期呼び出しが継続的に実行されないようにします。

--watch オプションを使用する場合、動作は通常 **oc rsync** に渡される引数の使用を含め **oc rsync** を繰り返し手動で起動する場合と同様になります。そのため、**--delete** などの **oc rsync** の手動の呼び出しで使用される同じフラグでこの動作を制御できます。

31.9. 高度な RSYNC 機能

oc rsync コマンドは標準の **rsync** ほどコマンドラインのオプションを表示しません。**oc rsync** で利用できない標準の **rsync** コマンドラインオプションを使用する場合 (**--exclude-from=FILE** オプションなど)、以下のように標準の **rsync** の **--rsh** (**-e**) オプションまたは **RSYNC_RSH** 環境変数を回避策として使用することができます。

```
$ rsync --rsh='oc rsh' --exclude-from=FILE SRC POD:DEST
```

または

```
$ export RSYNC_RSH='oc rsh'
$ rsync --exclude-from=FILE SRC POD:DEST
```

上記の例のいずれも標準の **rsync** をリモートシェルプログラムとして **oc rsh** を使用するように設定して リモート Pod に接続できるようにします。これらは **oc rsync** を実行する代替方法となります。

第32章 ポート転送

32.1. 概要

OpenShift Container Platform は Kubernetes に組み込まれた機能を利用してPod へのポート転送をサポートします。詳細は、「 アーキテクチャー 」を参照してください。

CLI を使用して 1 つ以上のローカルポートを Pod に転送できます。これにより、指定されたポートまたはランダムのポートでローカルにリッスンでき、Pod の所定ポートへ/からデータを転送できます。

32.2. 基本的な使用方法

ポート転送のサポートは CLI に組み込まれて います。

```
$ oc port-forward <pod> [<local_port>:]<remote_port> [...[<local_port_n>:]<remote_port_n>]
```

CLI はユーザーによって指定されたそれぞれのローカルポートでリッスンし、以下で説明されているプロトコルで転送を実行します。

ポートは以下の形式を使用して指定できます。

5000	クライアントはポート 5000 でローカルにリッスンし、Pod の 5000 に転送します。
6000:5000	クライアントはポート 6000 でローカルにリッスンし、Pod の 5000 に転送します。
:5000 または **0:5000**	クライアントは空きのローカルポートを選択し、Pod の 5000 に転送します。

たとえば、ポート **5000** および **6000** でローカルにリッスンし、Pod のポート **5000** および **6000** へ/からデータを転送するには、以下を実行します。

```
$ oc port-forward <pod> 5000 6000
```

ポート **8888** でローカルにリッスンし、Pod の **5000** に転送するには、以下を実行します。

```
$ oc port-forward <pod> 8888:5000
```

空きポートでローカルにリッスンし、Pod の **5000** に転送するには、以下を実行します。

```
$ oc port-forward <pod> :5000
```

または、以下を実行します。

```
$ oc port-forward <pod> 0:5000
```

32.3. プロトコル

クライアントは Kubernetes API サーバーに対して要求を実行して Pod へのポート転送を実行します。

```
/proxy/minions/<node_name>/portForward/<namespace>/<pod>
```

上記の URL には以下が含まれます。

- **<node_name>** はノードの FQDN です。
- **<namespace>** はターゲット Pod の namespace です。
- **<pod>** はターゲット Pod の名前です。

以下に例を示します。

```
/proxy/minions/node123.openshift.com/portForward/myns/mypod
```

ポート転送要求を API サーバーに送信した後に、クライアントは多重化ストリームをサポートするものに接続をアップグレードします。現在の実装では SPDY を使用しています。

クライアントは Pod のターゲットポートを含む **port** ヘッダーでストリームを作成します。ストリームに書き込まれるすべてのデータは Kubelet 経由でターゲット Pod およびポートに送信されます。同様に、転送された接続で Pod から送信されるすべてのデータはクライアントの同じストリームに送信されます。

クライアントは、ポート転送要求が終了するとすべてのストリーム、アップグレードされた接続および基礎となる接続を閉じます。

注記

詳細については、『 Architecture 』ガイドを参照してください。

第33章 共有メモリー

33.1. 概要

Linux には、System V と POSIX という 2 つのタイプの共有メモリーオブジェクトがあります。Pod のコンテナーは Pod インフラストラクチャーコンテナーの IPC namespace を共有し、System V 共有メモリーオブジェクトを共有できます。本書ではそれらが POSIX 共有メモリーオブジェクトを共有する方法についても説明します。

33.2. POSIX 共有メモリー

POSIX 共有メモリーでは、tmpfs が **/dev/shm** にマウントされる必要があります。Pod のコンテナーはそれらのマウント namespace を共有しないため、ボリュームを使用して同じ **/dev/shm** を Pod の各コンテナーに提供します。以下の例では、2 つのコンテナー間で POSIX 共有メモリーをセットアップする方法を示しています。

shared-memory.yaml

```
---
apiVersion: v1
id: hello-openshift
kind: Pod
metadata:
  name: hello-openshift
  labels:
    name: hello-openshift
spec:
  volumes:                          1
    - name: dshm
      emptyDir:
        medium: Memory
  containers:
    - image: kubernetes/pause
      name: hello-container1
      ports:
        - containerPort: 8080
          hostPort: 6061
      volumeMounts:                 2
        - mountPath: /dev/shm
          name: dshm
    - image: kubernetes/pause
      name: hello-container2
      ports:
        - containerPort: 8081
          hostPort: 6062
      volumeMounts:                 3
        - mountPath: /dev/shm
          name: dshm
```

1. tmpfs ボリューム **dshm** を指定します。
2. **dshm** で **hello-container1** の POSIX 共有メモリーを有効にします。

3 **dshm** で **hello-container2** の POSIX 共有メモリーを有効にします。

shared-memory.yaml ファイルを使用して Pod を作成します。

```
$ oc create -f shared-memory.yaml
```

第34章 アプリケーションの正常性

34.1. 概要

ソフトウェアのシステムでは、コンポーネントは一時的な問題 (一時的に接続が失われるなど)、設定エラー、または外部の依存関係に関する問題などにより正常でなくなることがあります。OpenShift Container Platform アプリケーションには、正常でないコンテナーを検出し、これに対応するための数多くのオプションがあります。

34.2. プローブを使用したコンテナーのヘルスチェック

プローブは実行中のコンテナーで定期的に実行する Kubernetes の動作です。現時点では、2 つのタイプのプローブがあり、それぞれが目的別に使用されています。

liveness プローブ	liveness プローブは、liveness プローブが設定されているコンテナーが実行中であるかどうかを判別します。liveness プローブが失敗すると、kubelet はその再起動ポリシーに基づいてコンテナーを強制終了します。Pod 設定の **template.spec.containers.livenessprobe** スタンザを設定して liveness チェックを設定します。
readiness プローブ	readiness プローブはコンテナーが要求を提供できるかどうかを判別します。readiness プローブがコンテナーで失敗する場合、エンドポイントコントローラーはコンテナーの IP アドレスがすべてのエンドポイントから削除されるようにします。readiness プローブはコンテナーが実行中の場合でも、それがプロキシーからトラフィックを受信しないようにエンドポイントコントローラーに対して信号を送るために使用できます。Pod 設定の **template.spec.containers.readinessprobe** スタンザを設定して readiness チェックを設定します。

プローブの正確なタイミングは、秒単位で表される 2 つのフィールドで制御されます。

フィールド	説明
initialDelaySeconds	コンテナーのプローブ開始後の待機時間。
timeoutSeconds	プローブが終了するまでの待機時間 (デフォルト: **1**)。この時間を過ぎると、OpenShift Container Platform はプローブが失敗したものとみなします。

どちらのプローブも以下の 3 つの方法で設定できます。

HTTP チェック

kubelet は web hook を使用してコンテナーの正常性を判別します。このチェックは HTTP の応答コードが 200 から 399 までの値の場合に正常とみなされます。以下は、HTTP チェック方法を使用した readiness チェックの例です。

例34.1 Readiness HTTP チェック

```
...
readinessProbe:
```

```
    httpGet:
      path: /healthz
      port: 8080
    initialDelaySeconds: 15
    timeoutSeconds: 1
  ...
```

HTTP チェックは、これが完全に初期化されている場合は HTTP ステータスコードを返すアプリケーションに適しています。

コンテナー実行チェック

kubeletは、コンテナー内でコマンドを実行します。ステータス 0 でチェックを終了すると正常であるとみなされます。以下はコンテナー実行方法を使用した liveness チェックの例です。

例34.2 Liveness コンテナー実行チェック

```
  ...
  livenessProbe:
    exec:
      command:
      - cat
      - /tmp/health
    initialDelaySeconds: 15
  ...
```

注記

timeoutSeconds パラメーターは、コンテナー実行チェックの Readiness および Liveness プローブには影響はありません。OpenShift Container Platform はコンテナーへの実行呼び出しでタイムアウトにならないため、タイムアウトをプローブ自体に実装できます。プローブでタイムアウトを実装する 1 つの方法として、**timeout** パラメーターを使用して liveness プローブおよび readiness プローブを実行できます。

```
[...]
      livenessProbe:
       exec:
         command:
           - /bin/bash
           - '-c'
           - timeout 60 /opt/eap/bin/livenessProbe.sh 1
       timeoutSeconds: 1
       periodSeconds: 10
       successThreshold: 1
       failureThreshold: 3
[...]
```

1 タイムアウト値およびプローブスクリプトへのパスです。

TCP ソケットチェック

kubelet はコンテナーに対してソケットを開くことを試行します。コンテナーはチェックで接続を確立できる場合にのみ正常であるとみなされます。以下は TCP ソケットチェック方法を使用した liveness チェックの例です。

例34.3 Liveness TCP ソケットチェック

```
...
livenessProbe:
  tcpSocket:
    port: 8080
  initialDelaySeconds: 15
  timeoutSeconds: 1
...
```

TCP ソケットチェック は、初期化が完了するまでリスニングを開始しないアプリケーションに適しています。

ヘルスチェックについての詳細は、Kubernetes ドキュメント を参照してください。

第35章 イベント

35.1. 概要

OpenShift Container Platform のイベントは OpenShift Container Platform クラスターの API オブジェクトに対して発生するイベントに基づいてモデル化されます。イベントにより、OpenShift Container Platform はリソースに依存しない方法で現実に生じているイベントについての情報を記録できます。また、開発者および管理者が統一された方法でシステムコンポーネントについての情報を使用できるようにします。

35.2. CLI によるイベントの表示

以下のコマンドを使って所定プロジェクトのイベントの一覧を取得できます。

```
$ oc get events [-n <project>]
```

35.3. コンソールでのイベントの表示

Web コンソールの **Browse → Events** ページでプロジェクトのイベントを表示できます。Pod およびデプロイメントなどの他の多くのオブジェクトには独自の **Events** タブもあり、これはオブジェクトに関連するイベントを表示します。

35.4. 総合的なイベント一覧

このセクションでは、OenShift Container Platform のイベントについて説明します。

表35.1 設定イベント

名前	説明
FailedValidation	Pod 設定の検証に失敗しました。

表35.2 コンテナーイベント

名前	説明
BackOff	バックオフ (再起動) によりコンテナーが失敗しました。
Created	コンテナーが作成されました。
Failed	プル/作成/起動が失敗しました。
Killing	コンテナーを強制終了しています。
Started	コンテナーが起動しました。
Preempting	他の Pod を退避します。

名前	説明
ExceededGracePeriod	コンテナーランタイムは、指定の猶予期間以内に Pod を停止しませんでした。

表35.3 正常性イベント

名前	説明
Unhealthy	コンテナーが正常ではありません。

表35.4 イメージイベント

名前	説明
BackOff	バックオフ (コンテナー起動、イメージのプル)。
ErrImageNeverPull	イメージの NeverPull Policy の違反があります。
Failed	イメージのプルに失敗しました。
InspectFailed	イメージの検査に失敗しました。
Pulled	イメージのプルに成功し、コンテナーイメージがマシンにすでに置かれています。
Pulling	イメージをプルしています。

表35.5 イメージマネージャーイベント

名前	説明
FreeDiskSpaceFailed	空きディスク容量に関連する障害が発生しました。
InvalidDiskCapacity	無効なディスク容量です。

表35.6 ノードイベント

名前	説明
FailedMount	ボリュームのマウントに失敗しました。

名前	説明
HostNetworkNotSupported	ホストのネットワークがサポートされていません。
HostPortConflict	ホスト/ポートの競合
InsufficientFreeCPU	空き CPU が十分にありません。
InsufficientFreeMemory	空きメモリーが十分にありません。
KubeletSetupFailed	Kubelet のセットアップに失敗しました。
NilShaper	シェイパーが定義されていません。
NodeNotReady	ノードの準備ができていません。
NodeNotSchedulable	ノードがスケジュール可能ではありません。
NodeReady	ノードの準備ができています。
NodeSchedulable	ノードがスケジュール可能です。
NodeSelectorMismatching	ノードセレクターの不一致があります。
OutOfDisk	ディスクの空き容量が不足しています。
Rebooted	ノードが再起動しました。
Starting	kubelet を起動しています。
FailedAttachVolume	ボリュームの割り当てに失敗しました。
FailedDetachVolume	ボリュームの割り当て解除に失敗しました。
VolumeResizeFailed	ボリュームの拡張/縮小に失敗しました。

名前	説明
VolumeResizeSuccessful	正常にボリュームを拡張/縮小しました。
FileSystemResizeFailed	ファイルシステムの拡張/縮小に失敗しました。
FileSystemResizeSuccessful	正常にファイルシステムが拡張/縮小されました。
FailedUnMount	ボリュームのマウント解除に失敗しました。
FailedMapVolume	ボリュームのマッピングに失敗しました。
FailedUnmapDevice	デバイスのマッピング解除に失敗しました。
AlreadyMountedVolume	ボリュームがすでにマウントされています。
SuccessfulDetachVolume	ボリュームの割り当てが正常に解除されました。
SuccessfulMountVolume	ボリュームが正常にマウントされました。
SuccessfulUnMountVolume	ボリュームのマウントが正常に解除されました。
ContainerGCFailed	コンテナーのガベージコレクションに失敗しました。
ImageGCFailed	イメージのガベージコレクションに失敗しました。
FailedNodeAllocatableEnforcement	システム予約の Cgroup 制限の実施に失敗しました。
NodeAllocatableEnforced	システム予約の Cgroup 制限を有効にしました。
UnsupportedMountOption	マウントオプションが非対応です。
SandboxChanged	Pod のサンドボックスが変更されました。

名前	説明
FailedCreatePodSandBox	Pod のサンドボックスの作成に失敗しました。
FailedPodSandBoxStatus	Pod サンドボックスの状態取得に失敗しました。

表35.7 Pod ワーカーイベント

名前	説明
FailedSync	Pod の同期が失敗しました。

表35.8 システムイベント

名前	説明
SystemOOM	クラスターに OOM (out of memory) 状態が発生しました。

表35.9 Pod イベント

名前	説明
FailedKillPod	Pod の停止に失敗しました。
FailedCreatePodContainer	Pod コンテナーの作成に失敗しました。
Failed	Pod データディレクトリーの作成に失敗しました。
NetworkNotReady	ネットワークの準備ができていません。
FailedCreate	作成エラー: **<error-msg>**.
SuccessfulCreate	作成された Pod: **<pod-name>**.
FailedDelete	削除エラー: **<error-msg>**.
SuccessfulDelete	削除した Pod: **<pod-id>**.

表35.10 Horizontal Pod AutoScaler イベント

名前	説明
SelectorRequired	セレクターが必要です。
InvalidSelector	セレクターを適切な内部セレクターオブジェクトに変換できませんでした。
FailedGetObjectMetric	HPA はレプリカ数を計算できませんでした。
InvalidMetricSourceType	不明なメトリクスソースタイプです。
ValidMetricFound	HPA は正常にレプリカ数を計算できました。
FailedConvertHPA	指定の HPA への変換に失敗しました。
FailedGetScale	HPA コントローラーは、ターゲットの現在のスケーリングを取得できませんでした。
SucceededGetScale	HPA コントローラーは、ターゲットの現在のスケールを取得できました。
FailedComputeMetricsReplicas	表示されているメトリクスに基づく必要なレプリカ数の計算に失敗しました。
FailedRescale	新しいサイズ: **<size>**; 理由:**<msg>**; エラー:**<error-msg>**
SuccessfulRescale	新しいサイズ: **<size>**; 理由:**<msg>**.
FailedUpdateStatus	状況の更新に失敗しました。

表35.11 ネットワークイベント (openshift-sdn)

名前	説明
Starting	OpenShift-SDN を開始します。
NetworkFailed	Pod のネットワークインターフェースがなくなり、Pod が停止します。

表35.12 ネットワークイベント (kube-proxy)

名前	説明
NeedPods	サービスポート **<serviceName>:<port>** は Pod が必要です。

表35.13 ボリュームイベント

名前	説明
FailedBinding	利用可能な永続ボリュームがなく、ストレージクラスが設定されていません。
VolumeMismatch	ボリュームサイズまたはクラスが要求の内容と異なります。
VolumeFailedRecycle	再利用 Pod の作成エラー
VolumeRecycled	ボリュームの再利用時に発生します。
RecyclerPod	Pod の再利用時に発生します。
VolumeDelete	ボリュームの削除時に発生します。
VolumeFailedDelete	ボリュームの削除時のエラー。
ExternalProvisioning	要求のボリュームが手動または外部ソフトウェアでプロビジョニングされる場合に発生します。
ProvisioningFailed	ボリュームのプロビジョニングに失敗しました。
ProvisioningCleanupFailed	プロビジョニングしたボリュームの消去エラー
ProvisioningSucceeded	ボリュームが正常にプロビジョニングされる場合に発生します。
WaitForFirstConsumer	Pod のスケジューリングまでバインドが遅延します。

表35.14 ライフサイクルフック

名前	説明
FailedPostStartHook	ハンドラーが Pod の起動に失敗しました。
FailedPreStopHook	ハンドラーが pre-stop に失敗しました。
UnfinishedPreStopHook	Pre-stop フックが完了しませんでした。

表35.15 デプロイメント

名前	説明
DeploymentCancellationFailed	デプロイメントのキャンセルに失敗しました。
DeploymentCancelled	デプロイメントがキャンセルされました。
DeploymentCreated	新規レプリケーションコントローラーが作成されました。
IngressIPRangeFull	サービスに割り当てる Ingress IP がありません。

表35.16 スケジューラーイベント

名前	説明
FailedScheduling	Pod のスケジューリングに失敗: **<pod-namespace>/<pod-name>**。このイベントは **AssumePodVolumes** の失敗、バインドの拒否など、複数の理由で発生します。
Preempted	ノード **<node-name>** にある **<preemptor-namespace>/<preemptor-name>**
Scheduled	**<node-name>** に **<pod-name>** が正常に割り当てられました。

表35.17 DaemonSet イベント

名前	説明
SelectingAll	この DaemonSet は全 Pod を選択しています。空でないセレクターが必要です。
FailedPlacement	**<node-name>** への Pod の配置に失敗しました。
FailedDaemonPod	ノード **<node-name>** で問題のあるデーモン Pod **<pod-name>** が見つかりました。この Pod の終了を試行します。

表35.18 LoadBalancer サービスイベント

名前	説明
CreatingLoadBalancerFailed	ロードバランサーの作成エラー
DeletingLoadBalancer	ロードバランサーを削除します。

名前	説明
EnsuringLoadBalancer	ロードバランサーを確保します。
EnsuredLoadBalancer	ロードバランサーを確保しました。
UnAvailableLoadBalancer	**LoadBalancer** サービスに利用可能なノードがありません。
LoadBalancerSourceRanges	新規の **LoadBalancerSourceRanges** を表示します。例: **<old-source-range>** → **<new-source-range>**
LoadbalancerIP	新しい IP アドレスを表示します。例: **<old-ip>** → **<new-ip>**
ExternalIP	外部 IP アドレスを表示します。例: **Added: <external-ip>**
UID	新しい UID を表示します。例: **<old-service-uid>** → **<new-service-uid>**
ExternalTrafficPolicy	新しい **ExternalTrafficPolicy** を表示します。例: **<old-policy>** → **<new-ploicy>**
HealthCheckNodePort	新しい **HealthCheckNodePort** を表示します。例: **<old-node-port>** → **new-node-port>**
UpdatedLoadBalancer	新規ホストでロードバランサーを更新しました。
LoadBalancerUpdateFailed	新規ホストでのロードバランサーの更新に失敗しました。
DeletingLoadBalancer	ロードバランサーを削除します。
DeletingLoadBalancerFailed	ロードバランサーの削除エラー。
DeletedLoadBalancer	ロードバランサーを削除しました。

第36章 環境変数の管理

36.1. 環境変数の設定および設定解除

OpenShift Container Platform は **oc set env** コマンドを提供して、レプリケーションコントローラーまたはデプロイメント設定などの Pod テンプレート を持つオブジェクトの環境変数の設定または設定解除を実行します。また、Pod および Pod テンプレートを持つオブジェクトの環境変数を一覧表示します。このコマンドは **BuildConfig** オブジェクトで使用することもできます。

36.2. 環境変数の一覧表示

Pod または Pod テンプレートの環境変数を一覧表示するには、以下を実行します。

```
$ oc set env <object-selection> --list [<common-options>]
```

この例では、Pod **p1** のすべての環境変数を一覧表示します。

```
$ oc set env pod/p1 --list
```

36.3. 環境変数の設定

Pod テンプレートに環境変数を設定するには、以下を実行します。

```
$ oc set env <object-selection> KEY_1=VAL_1 ... KEY_N=VAL_N [<set-env-options>] [<common-options>]
```

環境オプションを設定します。

オプション	説明
-e, --env=<KEY>=<VAL>	環境変数のキーと値のペアを設定します。
--overwrite	既存の環境変数の更新を確定します。

以下の例では、両方のコマンドがデプロイメント設定 **registry** で環境変数 **STORAGE** を変更します。最初に値 **/data** を追加します。2 番目の更新（値 **/opt**)。

```
$ oc set env dc/registry STORAGE=/data
$ oc set env dc/registry --overwrite STORAGE=/opt
```

以下の例では、現在のシェルで **RAILS_** で始まる名前を持つ環境変数を検索し、それらをサーバーのレプリケーションコントローラー **r1** に追加します。

```
$ env | grep RAILS_ | oc set env rc/r1 -e -
```

以下の例では、**rc.json** で定義されたレプリケーションコントローラーを変更しません。代わりに、更新された環境 **STORAGE=/local** を含む YAML オブジェクトを新規ファイル **rc.yaml** に書き込みます。

```
$ oc set env -f rc.json STORAGE=/opt -o yaml > rc.yaml
```

36.3.1. 自動的に追加された環境変数

表36.1 自動的に追加された環境変数

変数名
<SVCNAME>_SERVICE_HOST
<SVCNAME>_SERVICE_PORT

使用例

TCP ポート 53 を公開し、クラスター IP アドレス 10.0.0.11 が割り当てられたサービス **KUBERNETES** は以下の環境変数を生成します。

```
KUBERNETES_SERVICE_PORT=53
MYSQL_DATABASE=root
KUBERNETES_PORT_53_TCP=tcp://10.0.0.11:53
KUBERNETES_SERVICE_HOST=10.0.0.11
```

注記

oc rsh コマンドを使用してコンテナーに対して SSH を実行し、**oc set env** を実行して利用可能なすべての変数を一覧表示します。

36.4. 環境変数の設定解除

Pod テンプレートで環境変数を設定解除するには、以下を実行します。

```
$ oc set env <object-selection> KEY_1- ... KEY_N- [<common-options>]
```

重要

末尾のハイフン (**-**, U+2D) は必須です。

この例では、環境変数 **ENV1** および **ENV2** をデプロイメント設定 **d1** から削除します。

```
$ oc set env dc/d1 ENV1- ENV2-
```

これは、すべてのレプリケーションコントローラーから環境変数 **ENV** を削除します。

```
$ oc set env rc --all ENV-
```

これは、レプリケーションコントローラー **r1** のコンテナー **c1** から環境変数 **ENV** を削除します。

```
$ oc set env rc r1 --containers='c1' ENV-
```

第37章 ジョブ

37.1. 概要

レプリケーションコントローラー とは対照的に、ジョブは Pod を任意の数のレプリカと共に完了するまで実行します。ジョブはタスクの全体的な進捗状況を追跡し、アクティブな Pod、成功および失敗した Pod についての情報でそのステータスを更新します。ジョブを削除すると、作成した Pod レプリカが削除されます。ジョブは Kubernetes API の一部で、他の オブジェクトタイプ と同様に **oc** コマンドで管理できます。

ジョブについての詳細は、Kubernetes のドキュメント を参照してください。

37.2. ジョブの作成

ジョブ設定は以下の主な部分で構成されます。

- Pod テンプレート: Pod が作成するアプリケーションを記述します。
- オプションの **parallelism** パラメーター: ジョブの実行に使用する、並行して実行される Pod のレプリカ数を指定します。これが指定されていない場合、デフォルトは **completions** パラメーターの値に設定されます。
- オプションの **completions** パラメーター: ジョブの実行に使用する、並行して実行される Pod の数を指定します。指定されていない場合、デフォルトで 1 の値に設定されます。

以下は、**job** リソースのサンプルです。

```
apiVersion: batch/v1
kind: Job
metadata:
  name: pi
spec:
  parallelism: 1    (1)
  completions: 1    (2)
  template:         (3)
    metadata:
      name: pi
    spec:
      containers:
      - name: pi
        image: perl
        command: ["perl",  "-Mbignum=bpi", "-wle", "print bpi(2000)"]
      restartPolicy: OnFailure    (4)
```

1. ジョブが並行して実行する Pod のレプリカ数のオプションの値です。 デフォルトでは **completions** の値に設定されます。
2. ジョブを完了としてマークするために必要な Pod の正常な完了数のオプションの値です。 デフォルトは 1 に設定されます。
3. コントローラーが作成する Pod のテンプレートです。
4. Pod の再起動ポリシー。これは、ジョブコントローラーには適用されません。詳細は、「既知の制限事項」 を参照してください。

oc run を使用して単一コマンドからジョブを作成し、起動することもできます。以下のコマンドは直前の例に指定されている同じジョブを作成し、これを起動します。

```
$ oc run pi --image=perl --replicas=1  --restart=OnFailure \
    --command -- perl -Mbignum=bpi -wle 'print bpi(2000)'
```

37.2.1. 既知の制限事項

ジョブ仕様の再起動ポリシーは **Pod** にのみ適用され、**ジョブコントローラー** には適用されません。ただし、ジョブコントローラーはジョブを完了まで再試行するようハードコーディングされます。

そのため **restartPolicy: Never** または **--restart=Never** により、**restartPolicy: OnFailure** または **--restart=OnFailure** と同じ動作が実行されます。つまり、ジョブが失敗すると、成功するまで (または手動で破棄されるまで) 自動で再起動します。このポリシーは再起動するサブシステムのみを設定します。

Never ポリシーでは、**ジョブコントローラー** が再起動を実行します。それぞれの再試行時に、ジョブコントローラーはジョブステータスの失敗数を増分し、新規 Pod を作成します。これは、それぞれの試行が失敗するたびに Pod の数が増えることを意味します。

OnFailure ポリシーでは、**kubelet** が再起動を実行します。それぞれの試行によりジョブステータスでの失敗数が増分する訳ではありません。さらに、kubelet は同じノードで Pod の起動に失敗したジョブを再試行します。

37.3. ジョブのスケーリング

ジョブは **oc scale** コマンドを **--replicas** オプションと共に使用してスケールアップしたり、スケールダウンしたりすることができます。これはジョブの場合には **spec.parallelism** パラメーターを変更します。これにより、並行して実行されている Pod のレプリカ数が変更され、ジョブが実行されます。

以下のコマンドは上記のジョブサンプルを使用し、**parallelism** パラメーターを 3 に設定します。

```
$ oc scale job pi --replicas=3
```

注記

レプリケーションコントローラーのスケーリングでは、**oc scale** コマンドを **--replicas** オプションと共に使用しますが、レプリケーションコントローラー設定の **replicas** パラメーターを変更します。

37.4. 最長期間の設定

ジョブ を定義する際に、**activeDeadlineSeconds** フィールドを設定して最長期間を定義することができます。これが秒単位で指定され、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、実施される最長期間はありません。

最長期間は、最初の Pod がスケジュールされた時点から計算され、ジョブが有効である期間を定義します。これは実行の全体の時間を追跡し、完了の数 (タスクを実行するために必要な Pod のレプリカ数) とは無関係に追跡されます。指定されたタイムアウトに達すると、ジョブは OpenShift Container Platform で終了されます。

以下の例は、**activeDeadlineSeconds** フィールドを 30 分に指定する **ジョブ** の一部を示しています。

```
spec:
  activeDeadlineSeconds: 1800
```

37.5. ジョブ失敗のバックオフポリシー

ジョブは、設定の論理的なエラーなどの理由により再試行の設定回数を超えた後に失敗とみなされる場合があります。ジョブの再試行回数を指定するには、**.spec.backoffLimit** プロパティーを設定します。このフィールドはデフォルトで 6 に設定されます。ジョブに関連付けられた失敗した Pod は 6 分を上限として指数関数的バックオフ遅延値 (**10s**、**20s**、 **40s** ...) に基づいて再作成されます。この制限は、コントローラーのチェック間で失敗した Pod が新たに生じない場合に再設定されます。

第38章 OPENSHIFT PIPELINE

38.1. 概要

OpenShift Pipeline により、OpenShift 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 のドメイン固有言語 (DSL) (OpenShift 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。

38.2. OPENSHIFT JENKINS クライアントプラグイン

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 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 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。

このプラグインのインストールおよび設定についての詳細は、「 Configuring Pipeline Execution 」を参照してください。

38.2.1. OpenShift DSL

OpenShift Jenkins クライアントプラグインは、Jenkins スレーブから OpenShift API と通信するために Fluent (流れるような) スタイルの DSL を提供します。OpenShift DSL は Groovy 構文をベースとしており、作成、ビルド、デプロイ、および削除などのアプリケーションのライフサイクルを制御する方法を提供します。

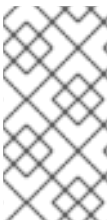
API の詳細は、実行中の Jenkins インスタンス内にあるプラグインのオンラインドキュメントに記載されています。これを検索するには、以下を実行します。

- 新規のパイプラインアイテムを作成します。
- DSL テキスト領域の下にある **Pipeline Syntax** をクリックします。
- 左側のナビゲーションメニューから、**Global Variables Reference** をクリックします。

38.3. JENKINS PIPELINE ストラテジー

プロジェクト内で OpenShift 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 root で **jenkinsfile** を使用するようにデフォルト設定されますが、以下の設定オプションも提供します。

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



注記

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

Jenkins Pipeline ストラテジーについての詳細は、「Pipeline ストラテジーのオプション」を参照してください。

38.4. JENKINSFILE

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 **JenkinsPiplineStrategy** セクションの **jenkinsfilePath** フィールドで指定される名前のファイル (指定される場合)。 指定されない場合は、リポジトリーのルートに設定されます。

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

38.5. チュートリアル

Jenkins Pipeline を使用したアプリケーションのビルドおよびデプロイについての詳細は、「Jenkins パイプラインのチュートリアル」を参照してください。

38.6. 詳細トピック

38.6.1. Jenkins 自動プロビジョニングの無効化

パイプラインのビルド設定が作成される場合、OpenShift は現時点で現行プロジェクトでプロビジョニングされた Jenkins マスター Pod があるかどうかを確認します。Jenkins マスターが見つからない場合、これが自動的に作成されます。この動作が必要でないか、または OpenShift の外部にある Jenkins サーバーを使用する場合は、これを無効にすることができます。

詳細は、「 Configuring Pipeline Execution 」を参照してください。

38.6.2. スレーブ Pod の設定

Kubernetes プラグインも公式の Jenkins イメージに事前にインストールされます。このプラグインによって、Jenkins マスターは OpenShift でスレーブ Pod を作成し、Pod に特定ジョブの特定ランタイムを提供すると同時に、実行中のジョブをそれらに委任して拡張性を実現できます。

Kubernetes プラグインを使用してスレーブ Pod を作成する方法についての詳細は、Kubernetes プラグインを参照してください。

第39章 CRON ジョブ

39.1. 概要

Cron ジョブ は、ジョブの実行スケジュールを指定できるようにすることで通常のジョブに基づいてビルドされます。Cron ジョブは Kubernetes API の一部であり、他の オブジェクトタイプ と同様に **oc** コマンドで管理できます。

警告

Cron ジョブはスケジュールの実行時間ごとに約 1 回ずつジョブオブジェクトを作成しますが、ジョブの作成に失敗したり、2 つのジョブが作成される可能性のある状況があります。そのため、ジョブはべき等である必要があり、履歴制限を設定する必要があります。

39.2. CRON ジョブの作成

Cron ジョブの設定は以下の主な部分で構成されます。

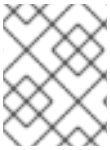
- cron 形式で指定されるスケジュール。
- 次のジョブの作成時に使用されるジョブテンプレート。
- ジョブを開始するためのオプションの期限 (秒単位)(何らかの理由によりスケジュールされた時間が経過する場合)。ジョブの実行が行われない場合、ジョブの失敗としてカウントされます。これが指定されない場合は期間が設定されません。
- **ConcurrencyPolicy**: オプションの同時実行ポリシー。Cron ジョブ内での同時実行ジョブを処理する方法を指定します。以下の同時実行ポリシーの 1 つのみを指定できます。 これが指定されない場合、同時実行を許可するようにデフォルト設定されます。
  - **Allow**: Cron ジョブを同時に実行できます。
  - **Forbid**: 同時実行を禁止し、直前の実行が終了していない場合は次の実行を省略します。
  - **Replace**: 同時に実行されているジョブを取り消し、これを新規ジョブに置き換えます。
- Cron ジョブの停止を許可するオプションのフラグ。これが **true** に設定されている場合、後続のすべての実行が停止されます。

以下は、**CronJob** リソースのサンプルです。

```
apiVersion: batch/v1beta1
kind: CronJob
metadata:
  name: pi
spec:
  schedule: "*/1 * * * *" 1
  jobTemplate:            2
    spec:
```

```
  template:
    metadata:
      labels:          3
        parent: "cronjobpi"
    spec:
      containers:
      - name: pi
        image: perl
        command: ["perl",  "-Mbignum=bpi", "-wle", "print bpi(2000)"]
      restartPolicy: OnFailure 4
```

1. ジョブのスケジュールです。この例では、ジョブは 1 分ごとに実行されます。
2. ジョブテンプレートです。これは、ジョブの例と同様です。
3. この Cron ジョブで生成されるジョブのラベルを設定します。
4. Pod の再起動ポリシー。これは、ジョブコントローラーには適用されません。詳細は、「既知の問題および制限」を参照してください。



注記

すべての cron ジョブ **schedule** の時間は、ジョブが実行されるマスターのタイムゾーンをベースとします。

oc run を使用して単一コマンドから cron ジョブを作成し、起動することもできます。以下のコマンドは直前の例で指定されている同じ cron ジョブを作成し、これを起動します。

```
$ oc run pi --image=perl --schedule='*/1 * * * *' \
    --restart=OnFailure --labels parent="cronjobpi" \
    --command -- perl -Mbignum=bpi -wle 'print bpi(2000)'
```

oc runで、**--schedule** オプションは cron 形式のスケジュールを受け入れます。

注記

Cron ジョブの作成時に、**oc run** は **Never** または **OnFailure** 再起動ポリシー (**--restart**) のみをサポートします。

ヒント

必要なくなった Cron ジョブを削除します。

```
$ oc delete cronjob/<cron_job_name>
```

これを実行することで、不要なアーティファクトの生成を防げます。

39.3. CRON ジョブ後のクリーンアップ

.spec.successfulJobsHistoryLimit と **.spec.failedJobsHistoryLimit** のフィールドはオプションです。これらのフィールドでは、完了したジョブと失敗したジョブのそれぞれを保存する数を指定します。デフォルトで、これらのジョブの保存数はそれぞれ **3** と **1** に設定されます。制限に **0** を設定すると、終了後に対応する種類のジョブのいずれも保持しません。

Cron ジョブはジョブや Pod などのアーティファクトリソースをそのままにすることがあります。ユーザーは履歴制限を設定して古いジョブとそれらの Pod が適切に消去されるようにすることが重要です。現時点で、これに対応する 2 つのフィールドが Cron ジョブ仕様にあります。

```
apiVersion: batch/v1beta1
kind: CronJob
metadata:
  name: pi
spec:
  successfulJobsHistoryLimit: 3 (1)
  failedJobsHistoryLimit: 1 (2)
  schedule: "*/1 * * * *"
  jobTemplate:
    spec:
    ...
```

(1)(1)(1) 保持する成功した終了済みジョブの数 (デフォルトは 3 に設定)。

(2)(2)(2) 保持する失敗した終了済みジョブの数 (デフォルトは 1 に設定)。

第40章 CREATE FROM URL

40.1. 概要

Create From URL (URL からの作成) は、イメージストリーム、イメージタグ、またはテンプレートから URL を構築できるようにする機能です。

Create from URL は、明示的にホワイトリスト化された namespace のイメージストリームまたはテンプレートでのみ機能します。ホワイトリストには、デフォルトで **openshift** namespace が含まれます。namespace をホワイトリストに追加するには、「Configuring the Create From URL Namespace Whitelist」を参照してください。

カスタムボタンを定義できます。

これらのボタンは、適切なクエリー文字列で定義された URL パターンを利用します。ユーザーにはプロジェクトを選択することを求めるプロンプトが出されます。次に Create from URL ワークフローが続きます。

40.2. イメージストリームおよびイメージタグの使用

40.2.1. クエリー文字列パラメーター

名前	説明	必須	スキーマ	デフォルト
imageStream	使用されるイメージストリームで定義される **metadata.name** の値。	true	文字列	
imageTag	使用されるイメージストリームで定義される **spec.tags.name** の値。	true	文字列	
namespace	使用するイメージストリームおよびイメージタグを含む namespace の名前。	false	文字列	**openshift**
name	このアプリケーション用に作成されるリソースを識別します。	false	文字列	

名前	説明	必須	スキーマ	デフォルト
sourceURI	アプリケーションのソースコードを含む Git リポジトリー URL。	false	文字列	
sourceRef	**sourceURI** で指定されるアプリケーションソースコードのブランチ、タグ、またはコミット。	false	文字列	
contextDir	**sourceURI** で指定されるアプリケーションソースコードのサブディレクトリー。 ビルドのコンテキストディレクトリーとして使用されます。	false	文字列	

注記

パラメーター値の 予約された文字 は URL エンコーディングされている必要があります。

40.2.1.1. 例

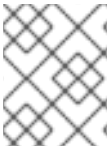
```
 create?
imageStream=nodejs&imageTag=4&name=nodejs&sourceURI=https%3A%2F%2Fgithub.com%2Fopenshift%2Fnodejs-ex.git&sourceRef=master&contextDir=%2F
```

40.3. テンプレートの使用

40.3.1. クエリー文字列パラメーター

名前	説明	必須	スキーマ	デフォルト
template	使用されるテンプレートで定義される **metadata.name** の値。	true	文字列	

名前	説明	必須	スキーマ	デフォルト
templateParamsMap	テンプレートパラメーター名と上書き対象の対応する値が含まれる JSON パラメーターマップ。	false	JSON	
namespace	使用するテンプレートを含む namespace の名前。	false	文字列	**openshift**



注記

パラメーター値の 予約された文字 は URL エンコーディングされている必要があります。

40.3.1.1. 例

```
 create?template=nodejs-mongodb-example&templateParamsMap=
{"SOURCE_REPOSITORY_URL"%3A"https%3A%2F%2Fgithub.com%2Fopenshift%2Fnodejs-ex.git"}
```

第41章 カスタムリソース定義からのオブジェクトの作成

41.1. KUBERNETES カスタムリソース定義

Kubernetes API では、リソースは特定の種類の API オブジェクトのコレクションを保管するエンドポイントです。たとえば、ビルトインされた Pod リソースには Pod オブジェクトのコレクションが含まれます。

カスタムリソースは、Kubernetes API を拡張するか、またはプロジェクトまたはクラスターに独自の API を導入することを可能にするオブジェクトです。

カスタムリソース定義 (CRD) ファイルは、独自のオブジェクトの種類を定義し、API サーバーがライフサイクル全体を処理できるようにします。

注記

クラスター管理者のみが CRD を作成できますが、読み取りと書き込みのパーミッションがある場合には、CRD からオブジェクトを作成できます。

41.2. CRD からのカスタムオブジェクトの作成

カスタムオブジェクトには、任意の JSON コードを含むカスタムフィールドを含めることができます。

前提条件

- CRD を作成します。

手順

1. カスタムオブジェクトの YAML 定義を作成します。以下の定義例では、**cronSpec** と **image** のカスタムフィールドが **CronTab** タイプのカスタムオブジェクトに設定されます。このタイプは、カスタムリソース定義オブジェクトの **spec.kind** フィールドから取得します。

   カスタムオブジェクトの YAML ファイルの例

```
apiVersion: "stable.example.com/v1" 1
kind: CronTab 2
metadata:
  name: my-new-cron-object 3
  finalizers: 4
  - finalizer.stable.example.com
spec: 5
  cronSpec: "* * * * /5"
  image: my-awesome-cron-image
```

   1 カスタムリソース定義からグループ名および API バージョン (名前/バージョン) を指定します。

   2 カスタムリソース定義のタイプを指定します。

   3 オブジェクトの名前を指定します。

   4 オブジェクトのファイナライザーを指定します (ある場合)。ファイナライザーは、コントローラーがオブジェクトの削除前に完了する必要のある条件を実装できるようにします。

5 オブジェクトのタイプに固有の条件を指定します。

2. オブジェクトファイルの作成後に、オブジェクトを作成します。

```
oc create -f <file-name>.yaml
```

41.3. カスタムオブジェクトの管理

オブジェクトを作成した後には、カスタムリソースを管理できます。

前提条件

- カスタムリソース定義 (CRD) を作成します。
- CRD からオブジェクトを作成します。

手順

1. 特定の種類のカスタムリソースについての情報を取得するには、以下を入力します。

```
oc get <kind>
```

以下に例を示します。

```
oc get crontab

NAME                 KIND
my-new-cron-object   CronTab.v1.stable.example.com
```

リソース名では大文字と小文字が区別されず、CRD で定義される単数形または複数形のいずれか、および任意の短縮名を指定できることに注意してください。以下に例を示します。

```
oc get crontabs
oc get crontab
oc get ct
```

2. カスタムリソースの未加工の YAML データも確認することができます。

```
oc get <kind> -o yaml
```

```
oc get ct -o yaml

apiVersion: v1
items:
- apiVersion: stable.example.com/v1
  kind: CronTab
  metadata:
    clusterName: ""
    creationTimestamp: 2017-05-31T12:56:35Z
    deletionGracePeriodSeconds: null
    deletionTimestamp: null
    name: my-new-cron-object
    namespace: default
```

```
  resourceVersion: "285"
  selfLink: /apis/stable.example.com/v1/namespaces/default/crontabs/my-new-cron-object
  uid: 9423255b-4600-11e7-af6a-28d2447dc82b
spec:
  cronSpec: '* * * * /5' 1
  image: my-awesome-cron-image 2
```

1 2 オブジェクトの作成に使用した YAML からのカスタムデータが表示されます。

第42章 アプリケーションメモリーのサイジング

42.1. 概要

ここでは、アプリケーション開発者が OpenShift Container Platform を使用して以下を実行する際に役立つ情報を提供します。

1. コンテナー化されたアプリケーションコンポーネントのメモリーおよびリスク要件を判別し、それらの要件を満たすようコンテナーメモリーパラメーターを設定する
2. コンテナー化されたアプリケーションランタイム (OpenJDK など) を、設定されたコンテナーメモリーパラメーターに基づいて最適に実行されるよう設定する
3. コンテナーでの実行に関連するメモリー関連のエラー状態を診断し、これを解決する

42.2. 背景情報

まず OpenShift Container Platform によるコンピュートリソースの管理方法の概要をよく読んでから次の手順に進むことをお勧めします。

アプリケーションメモリーのサイジングについては、以下が主要なポイントになります。

- 各種のリソース (メモリー、cpu、ストレージ) に応じて、OpenShift Container Platform ではオプションの **要求** および **制限** の値を Pod の各コンテナーに設定できます。ここでは、メモリー要求とメモリー制限のみに言及します。
- **メモリー要求**
  - メモリー要求値は、指定される場合 OpenShift Container Platform スケジューラーに影響を与えます。スケジューラーは、コンテナーのノードへのスケジュール時にメモリー要求を考慮し、コンテナーの使用のために選択されたノードで要求されたメモリーをフェンスオフします。
  - ノードのメモリーが使い切られると、OpenShift Container Platform はメモリー使用がメモリー要求を最も超過しているコンテナーのエビクションを優先します。メモリー消費の深刻な状況が生じる場合、ノードの OOM killer は同様のメトリクスに基づいてコンテナーでプロセスを選択し、これを強制終了する場合があります。
- **メモリー制限**
  - メモリー制限値が指定されている場合、コンテナーのすべてのプロセスに割り当て可能なメモリーにハード制限を指定します。
  - コンテナーのすべてのプロセスで割り当てられるメモリーがメモリー制限を超過する場合、ノードの OOM killer はコンテナーのプロセスをすぐに選択し、これを強制終了します。
  - メモリー要求とメモリー制限の両方が指定される場合、メモリー制限の値はメモリー要求の値よりも大きいか、またはこれと等しくなければなりません。
- **管理**
  - クラスター管理者はメモリーの要求値、制限値、これらの両方に対してクォータを割り当てるか、いずれにも割り当てないようにすることができます。

- クラスター管理者はメモリーの要求値、制限値またはこれらの両方についてデフォルト値を割り当てることも、それらのいずれにもデフォルト値を割り当てないようにすることもできます。
- クラスター管理者は、クラスターのオーバーコミットを管理するために開発者が指定するメモリー要求の値を上書きできます。これは OpenShift Online などで行われます。

42.3. ストラテジー

OpenShift Container Platform でアプリケーションメモリーをサイジングする手順は以下の通りです。

1. **予想されるコンテナーのメモリー使用の判別**
   必要時に予想される平均およびピーク時のコンテナーのメモリー使用を判別します (例: 別の負荷テストを実行)。コンテナーで並行して実行されている可能性のあるすべてのプロセスを必ず考慮に入れるようにしてください。 たとえば、メインのアプリケーションは付属スクリプトを生成しているかどうかを確認します。

2. **リスク選好 (risk appetite) の判別**
   エビクションのリスク選好を判別します。リスク選好のレベルが低い場合、コンテナーは予想されるピーク時の使用量と安全マージンのパーセンテージに応じてメモリーを要求します。リスク選好が高くなる場合、予想される平均の使用量に応じてメモリーを要求することがより適切な場合があります。

3. **コンテナーのメモリー要求の設定**
   上記に基づいてコンテナーのメモリー要求を設定します。要求がアプリケーションのメモリー使用をより正確に表示することが望ましいと言えます。要求が高すぎる場合には、クラスターおよびクォータの使用が非効率となります。要求が低すぎる場合、アプリケーションのエビクションの可能性が高くなります。

4. **コンテナーのメモリー制限の設定 (必要な場合)**
   必要時にコンテナーのメモリー制限を設定します。制限を設定すると、コンテナーのすべてのプロセスのメモリー使用量の合計が制限を超える場合にコンテナーのプロセスがすぐに強制終了されるため、いくつかの利点をもたらします。まずは予期しないメモリー使用の超過を早期に明確にする (「fail fast (早く失敗する)」) ことができ、次にプロセスをすぐに中止できます。

   一部の OpenShift Container Platform クラスターでは制限値を設定する必要があります。 制限に基づいて要求を上書きする場合があります。 また、一部のアプリケーションイメージは、要求値よりも検出が簡単なことから設定される制限値に依存します。

   メモリー制限が設定される場合、これは予想されるピーク時のコンテナーのメモリー使用量と安全マージンのパーセンテージよりも低い値に設定することはできません。

5. **アプリケーションが調整されていることの確認**
   適切な場合は、設定される要求および制限値に関連してアプリケーションが調整されていることを確認します。この手順は、JVM などのメモリーをプールするアプリケーションにおいてとくに当てはまります。残りの部分では、これについて説明します。

42.4. OPENSHIFT CONTAINER PLATFORM での OPENJDK のサイジング

デフォルトの OpenJDK 設定はコンテナー化された環境では機能しません。 コンテナーで OpenJDK を実行する場合は常に追加の Java メモリー設定を指定することがルールとなっているためです。

JVM のメモリーレイアウトは複雑で、バージョンに依存しており、本書ではこれについて詳細には説明しません。ただし、コンテナーで OpenJDK を実行する際のスタートにあたって少なくとも以下の 3 つのメモリー関連のタスクが主なタスクになります。

1. JVM 最大ヒープサイズを上書きする。
2. JVM が未使用メモリーをオペレーティングシステムに解放するよう促す (適切な場合)。
3. コンテナー内のすべての JVM プロセスが適切に設定されていることを確認する。

コンテナーでの実行に向けて JVM ワークロードを最適に調整する方法については本書では扱いませんが、これには複数の JVM オプションを追加で設定することが必要になる場合があります。

42.4.1. JVM 最大ヒープサイズの上書き

数多くの Java ワークロードにおいて、JVM ヒープはメモリーの最大かつ単一のコンシューマーです。現時点で OpenJDK は、OpenJDK がコンテナー内で実行されているかにかかわらず、ヒープに使用されるコンピュートノードのメモリーの最大 1/4 (1/**-XX:MaxRAMFraction**) を許可するようデフォルトで設定されます。そのため、コンテナーのメモリー制限も設定されている場合には、この動作をオーバーライドすることが **必須** です。

上記を実行する方法として、2 つ以上の方法を使用できます:

1. コンテナーのメモリー制限が設定されており、JVM で実験的なオプションがサポートされている場合には、**-XX:+UnlockExperimentalVMOptions -XX:+UseCGroupMemoryLimitForHeap** を設定します。
   これにより、**-XX:MaxRAM** がコンテナーのメモリー制限に設定され、最大ヒープサイズ (**-XX:MaxHeapSize** / **-Xmx**) が 1/**-XX:MaxRAMFraction** に設定されます (デフォルトでは 1/4)。
2. **-XX:MaxRAM**、**-XX:MaxHeapSize** または **-Xmx** のいずれかを直接上書きします。
   このオプションには、値のハードコーディングが必要になりますが、安全マージンを計算できるという利点があります。

42.4.2. JVM が未使用メモリーをオペレーティングシステムに解放するよう促す

デフォルトで、OpenJDK は未使用メモリーをオペレーティングシステムに積極的に返しません。これは多くのコンテナー化された Java ワークロードには適していますが、例外として、コンテナー内に JVM と共存する追加のアクティブなプロセスがあるワークロードの場合を考慮する必要があります。それらの追加のプロセスはネイティブのプロセスである場合や追加の JVM の場合、またはこれら 2 つの組み合わせである場合もあります。

OpenShift Container Platform Jenkins maven スレーブイメージ は、以下の JVM 引数を使用して JVM に未使用メモリーをオペレーティングシステムに解放するよう促します： **-XX:+UseParallelGC -XX:MinHeapFreeRatio=5 -XX:MaxHeapFreeRatio=10 -XX:GCTimeRatio=4 -XX:AdaptiveSizePolicyWeight** =90 .これらの引数は、割り当てられたメモリーが使用中のメモリー (**-XX:MaxHeapFreeRatio**) の 110% を超え、ガベージコレクター (**-XX:GCTimeRatio**) での CPU 時間の 20% を使用する場合は常にヒープメモリーをオペレーティングシステムに返すことが意図されています。アプリケーションのヒープ割り当てが初期のヒープ割り当て (**-XX:InitialHeapSize** / **-Xms** で上書きされる) を下回ることはありません。詳細情報については、「Tuning Java's footprint in OpenShift (Part 1)」、「Tuning Java's footprint in OpenShift (Part 2) 」、および「 OpenJDK and Containers」を参照してください。

42.4.3. コンテナー内のすべての JVM プロセスが適切に設定されていることを確認する

複数の JVM が同じコンテナーで実行される場合、それらすべてが適切に設定されていることを確認する必要があります。多くのワークロードでは、それぞれの JVM に memory budget のパーセンテージを付与する必要があります。 これにより大きな安全マージンが残される場合があります。

多くの Java ツールは JVM を設定するために各種の異なる環境変数 (**JAVA_OPTS**、**GRADLE_OPTS**、**MAVEN_OPTS** など) を使用します。 適切な設定が適切な JVM に渡されていることを確認するのが容易でない場合もあります。

JAVA_TOOL_OPTIONS 環境変数は常に OpenJDK によって使用され、**JAVA_TOOL_OPTIONS** で指定される値は JVM コマンドラインで指定される他のオプションによって上書きされます。デフォルトで、OpenShift Container Platform Jenkins maven スレーブイメージ は **JAVA_TOOL_OPTIONS="-XX:+UnlockExperimentalVMOptions -XX:+UseCGroupMemoryLimitForHeap -Dsun.zip.disableMemoryMapping=true"** を設定してこれらのオプションがスレーブイメージで実行されるすべての JVM ワークロードに対してデフォルトで使用されるようにします。これは、追加のオプションが不要になることを保証する訳ではありませんが、開始時には役立ちます。

42.5. POD 内でのメモリー要求および制限の検索

Pod 内からメモリー要求および制限を動的に検出するアプリケーションは Downward API を使用する必要があります。以下のスニペットはこれがどのように実行されるかを示しています。

```
apiVersion: v1
kind: Pod
metadata:
  name: test
spec:
  containers:
  - name: test
    image: fedora:latest
    command:
    - sleep
    - "3600"
    env:
    - name: MEMORY_REQUEST
      valueFrom:
        resourceFieldRef:
          containerName: test
          resource: requests.memory
    - name: MEMORY_LIMIT
      valueFrom:
        resourceFieldRef:
          containerName: test
          resource: limits.memory
    resources:
      requests:
        memory: 384Mi
      limits:
        memory: 512Mi
```

```
# oc rsh test
$ env | grep MEMORY | sort
MEMORY_LIMIT=536870912
MEMORY_REQUEST=402653184
```

メモリー制限値は、**/sys/fs/cgroup/memory/memory.limit_in_bytes** ファイルによってコンテナー内から読み取ることもできます。

42.6. OOM による強制終了の診断

OpenShift Container Platform は、コンテナーのすべてのプロセスのメモリー使用量の合計がメモリー制限を超えるか、またはノードのメモリーを使い切られるなどの深刻な状態が生じる場合にコンテナーのプロセスを強制終了する場合があります。

プロセスが OOM によって強制終了される場合、コンテナーがすぐに終了する場合もあれば、終了しない場合もあります。コンテナーの PID 1 プロセスが **SIGKILL** を受信する場合、コンテナーはすぐに終了します。それ以外の場合、コンテナーの動作は他のプロセスの動作に依存します。

コンテナーがすぐに終了しない場合、OOM による強制終了は以下のように検出できます。

1. コンテナーのプロセスは SIGKILL シグナルを受信したことを示すコード 137 で終了する。
2. **/sys/fs/cgroup/memory/memory.oom_control** の oom_kill カウンターの増分が確認されます。

```
$ grep '^oom_kill ' /sys/fs/cgroup/memory/memory.oom_control
oom_kill 0
$ sed -e '' </dev/zero  # provoke an OOM kill
Killed
$ echo $?
137
$ grep '^oom_kill ' /sys/fs/cgroup/memory/memory.oom_control
oom_kill 1
```

Pod の 1 つ以上のプロセスが OOM で強制終了され、Pod がこれに続いて終了する場合 (即時であるかどうかは問わない)、フェーズは **Failed**、理由は **OOMKilled** になります。OOM で強制終了された Pod は **restartPolicy** の値によって再起動する場合があります。再起動されない場合は、ReplicationController などのコントローラーが Pod の失敗したステータスを認識し、古い Pod に置き換わる新規 Pod を作成します。

再起動されない場合、Pod のステータスは以下のようになります。

```
$ oc get pod test
NAME      READY     STATUS      RESTARTS   AGE
test      0/1       OOMKilled   0          1m

$ oc get pod test -o yaml
...
status:
  containerStatuses:
  - name: test
    ready: false
    restartCount: 0
    state:
      terminated:
        exitCode: 137
        reason: OOMKilled
  phase: Failed
```

再起動される場合、そのステータスは以下のようになります。

```
$ oc get pod test
NAME      READY     STATUS    RESTARTS   AGE
test      1/1       Running   1          1m

$ oc get pod test -o yaml
...
status:
  containerStatuses:
  - name: test
    ready: true
    restartCount: 1
    lastState:
      terminated:
        exitCode: 137
        reason: OOMKilled
    state:
      running:
  phase: Running
```

42.7. エビクトされた POD の診断

OpenShift Container Platform は、ノードのメモリーが使い切られとそのノードから Pod をエビクトする場合があります。メモリー消費の度合いによって、エビクションは正常に行われる場合もあれば、そうでない場合もあります。正常なエビクションは、各コンテナーのメインプロセス (PID 1) が SIGTERM シグナルを受信してから、プロセスがすでに終了していない場合は後になって SIGKILL シグナルを受信することを意味します。正常ではないエビクションは各コンテナーのメインプロセスが SIGKILL シグナルを即時に受信することを示します。

エビクトされた Pod のフェーズは **Failed** に、理由 は **Evicted** になります。この場合、**restartPolicy** の値に関係なく再起動されません。ただし、ReplicationController などのコントローラーは Pod の失敗したステータスを認識し、古い Pod に置き換わる新規 Pod を作成します。

```
$ oc get pod test
NAME      READY     STATUS    RESTARTS   AGE
test      0/1       Evicted   0          1m

$ oc get pod test -o yaml
...
status:
  message: 'Pod The node was low on resource: [MemoryPressure].'
  phase: Failed
  reason: Evicted
```