# OpenShift Container Platform 4.12

## アプリケーションのビルド

OpenShift Container Platform でのアプリケーションの作成および管理

Last Updated: 2024-07-04

# OpenShift Container Platform 4.12 アプリケーションのビルド

OpenShift Container Platform でのアプリケーションの作成および管理

## 法律上の通知



## 概要

本書では、OpenShift Container Platform で実行されるユーザーによってプロビジョニングされたアプリケーションのインスタンスを作成し、管理する各種の方法について説明します。これには、プロジェクトの使用および Open Service Broker API を使用したアプリケーションのプロビジョニングについての情報が含まれます。

# 目次

# 第1章 アプリケーションのビルドの概要

OpenShift Container Platform を使用すると、Web コンソールまたはコマンドラインインターフェイス (CLI) を使用してアプリケーションを作成、編集、削除、および管理できます。

## 1.1. プロジェクトの使用

プロジェクトを使用すると、アプリケーションを分離して編成および管理できます。OpenShift Container Platform で、プロジェクトの作成、表示、削除 などを含め、プロジェクトライフサイクル全体を管理できます。

プロジェクトを作成したら、Developer パースペクティブを使用して、ユーザーに対して プロジェクトへのアクセス権の付与または取り消し と クラスターロールの管理 を行えます。また、新規プロジェクトの自動プロビジョニングに使用されるプロジェクトテンプレートを作成する際に、プロジェクト設定リソースの編集 も行えます。

CLI を使用して、OpenShift Container Platform API へのリクエストを借用して 別のユーザーとしてプロジェクトを作成 できます。新規プロジェクトの作成をリクエストすると、OpenShift Container Platform はエンドポイントを使用して、カスタマイズ可能なテンプレートに従ってプロジェクトをプロビジョニングします。クラスター管理者は、認証されたユーザーグループによる新規プロジェクトのセルフプロビジョニングを阻止 することを選択できます。

## 1.2. アプリケーションの使用

### 1.2.1. アプリケーションの作成

アプリケーションを作成するには、プロジェクトを作成しているか、適切なロールとパーミッションでプロジェクトにアクセスできる必要があります。Web コンソールの Developer パースペクティブ、インストール済みの Operator、OpenShift CLI (**oc**) のいずれかを使用して、アプリケーションを作成できます。プロジェクトに追加するアプリケーションは、Git、JAR ファイル、devfile、または開発者カタログから入手できます。

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を含むコンポーネントを使用し、OpenShift CLI (**oc**) を使用してアプリケーションを作成することもできます。OpenShift Container Platform Web コンソールを使用すると、クラスター管理者によってインストールされた Operator からアプリケーションを作成できます。

### 1.2.2. アプリケーションの保守

アプリケーションを作成したら、Web コンソールを使用して プロジェクトまたはアプリケーションのメトリクスを監視 できます。Web コンソールを使用して、アプリケーションを 編集 または 削除 することもできます。

アプリケーションの実行中は、すべてのアプリケーションリソースが使用されるわけではありません。クラスター管理者は、スケーラブルなリソースをアイドル状態 にして、リソースの消費を減らすことができます。

### 1.2.3. アプリケーションのサービスへの接続

アプリケーションはバッキングサービスを使用して、サービスプロバイダーに応じて異なるワークロードを構築および接続します。開発者として Service Binding Operator を使用すると、手作業でバインディング接続を設定する手順なしに、Operator が管理するバッキングサービスとワークロードを簡単にバインドできます。サービスバインディングは IBM Power、IBM Z、および IBM® LinuxONE 環境 にも適用できます。

### 1.2.4. アプリケーションのデプロイ

**Deployment** または **DeploymentConfig** オブジェクトを使用してアプリケーションをデプロイし、Web コンソールからそれらを 管理 できます。アプリケーションの変更またはアップグレード中のダウンタイムを短縮するのに役立つ デプロイメントストラテジー を作成できます。

アプリケーションやサービスの OpenShift Container Platform クラスターへのデプロイメントを単純化するソフトウェアパッケージマネージャーである Helm も使用できます。

## 1.3. RED HAT MARKETPLACE の使用

Red Hat Marketplace は、パブリッククラウドおよびオンプレミスで実行されるコンテナーベース環境向けの認定されたソフトウェアの検出とアクセスが可能なオープンクラウドマーケットプレースです。

# 第2章 プロジェクト

## 2.1. プロジェクトの使用

**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。

**注記**

**openshift-** および **kube-** で始まる名前のプロジェクトは デフォルトプロジェクト です。これらのプロジェクトは、Pod として実行されるクラスターコンポーネントおよび他のインフラストラクチャーコンポーネントをホストします。そのため、OpenShift Container Platform では **oc new-project** コマンドを使用して **openshift-** または **kube-** で始まる名前のプロジェクトを作成することができません。クラスター管理者は、**oc adm new-project** コマンドを使用してこれらのプロジェクトを作成できます。

**注記**

デフォルト namespace (**default**、**kube-system**、**kube-public**、**openshift-node**、**openshift-infra**、**openshift**) のいずれかに作成された Pod に SCC を割り当てることはできません。これらの namespace は Pod またはサービスを実行するために使用することはできません。

### 2.1.1. プロジェクトの作成

OpenShift Container Platform Web コンソールまたは OpenShift CLI (**oc**) を使用して、クラスター内にプロジェクトを作成できます。

#### 2.1.1.1. Web コンソールを使用したプロジェクトの作成

OpenShift Container Platform Web コンソールを使用して、クラスター内にプロジェクトを作成できます。

**注記**

**openshift-** および **kube-** で始まる名前のプロジェクトは OpenShift Container Platform によって重要 (Critical) と見なされます。そのため、OpenShift Container Platform では、Web コンソールを使用して **openshift-** で始まるプロジェクトを作成することはできません。

**前提条件**

- OpenShift Container Platform のプロジェクト、アプリケーション、および他のワークロードを作成するために適切なロールおよびパーミッションがあることを確認します。

**手順**

- **Administrator** パースペクティブを使用している場合:
  a. **Home → Projects** に移動します。
  b. **Create Project** をクリックします。

i. **Create Project** ダイアログボックスで、**Name** フィールドに、**myproject** などの一意の名前を入力します。

ii. オプション: プロジェクトの **Display name** および **Description** の詳細を追加します。

iii. **Create** をクリックします。
プロジェクトのダッシュボードが表示されます。

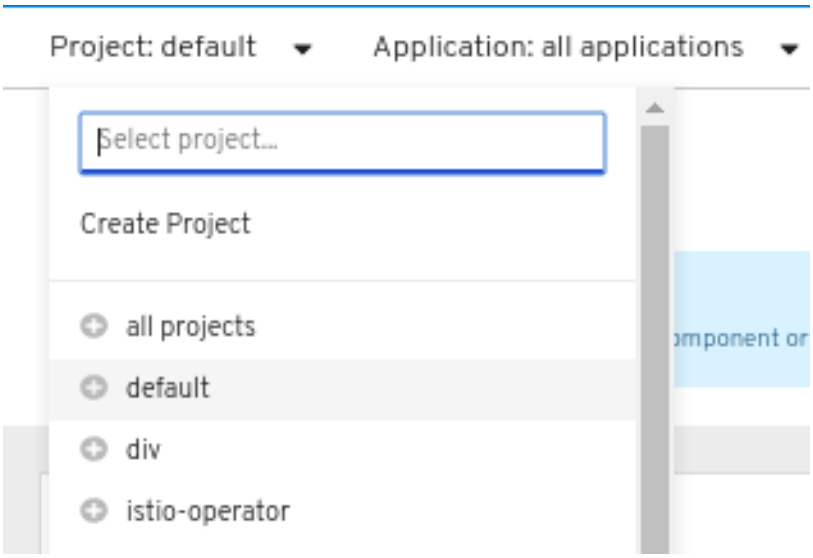
c. オプション: **Details** タブを選択して、プロジェクトの詳細を表示します。

d. オプション: プロジェクトに対する適切なパーミッションがある場合は、**Project Access** タブを使用して、プロジェクトの admin、edit、および view 権限を付与または取り消すことができます。

- **Developer** パースペクティブを使用している場合:

a. **Project** メニューをクリックし、**Create Project** を選択します。

**図2.1 Create project**




i. **Create Project** ダイアログボックスで、**Name** フィールドに、**myproject** などの一意の名前を入力します。

ii. オプション: プロジェクトの **Display name** および **Description** の詳細を追加します。

iii. **Create** をクリックします。

b. オプション: 左側のナビゲーションパネルを使用して **Project** ビューに移動し、プロジェクトのダッシュボードを表示します。

c. オプション: プロジェクトダッシュボードで **Details** タブを選択し、プロジェクトの詳細を表示します。

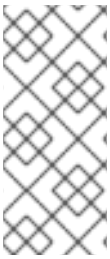
d. オプション: プロジェクトに対する適切なパーミッションがある場合は、プロジェクトダッシュボードの **Project Access** タブを使用して、プロジェクトの admin、edit、および view 権限を付与または取り消すことができます。

**関連情報**

- Web コンソールを使用した利用可能なクラスターのロールのカスタマイズ

#### 2.1.1.2. CLI を使用したプロジェクトの作成

クラスター管理者が許可する場合、新規プロジェクトを作成できます。



**注記**

**openshift-** および **kube-** で始まる名前のプロジェクトは OpenShift Container Platform によって重要 (Critical) と見なされます。そのため、OpenShift Container Platform では **oc new-project** コマンドを使用して **openshift-** または **kube-** で始まる名前のプロジェクトを作成することができません。クラスター管理者は、**oc adm new-project** コマンドを使用してこれらのプロジェクトを作成できます。

**注記**

デフォルト namespace (**default**、**kube-system**、**kube-public**、**openshift-node**、**openshift-infra**、**openshift**) のいずれかに作成された Pod に SCC を割り当てることはできません。これらの namespace は Pod またはサービスを実行するために使用することはできません。

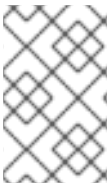
**手順**

- 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" \
    --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、新規プロジェクトを作成できるように既存プロジェクトを削除しなければならない場合があります。

### 2.1.2. プロジェクトの表示

OpenShift Container Platform Web コンソールまたは OpenShift CLI (**oc**) を使用して、クラスター内のプロジェクトを表示できます。

#### 2.1.2.1. Web コンソールを使用してプロジェクトを表示する

OpenShift Container Platform Web コンソールを使用して、アクセス権のあるプロジェクトを表示できます。

**手順**

- Administrator パースペクティブを使用している場合:
    a. ナビゲーションメニューで **Home → Projects** に移動します。

b. 表示するプロジェクトを選択します。**Overview** タブには、プロジェクトのダッシュボードが含まれています。

c. **Details** タブを選択して、プロジェクトの詳細を表示します。

d. **YAML** タブを選択して、プロジェクトリソースの YAML 設定を表示および更新します。

e. **Workloads** タブを選択して、プロジェクト内のワークロードを表示します。

f. **RoleBindings** タブを選択して、プロジェクトのロールバインディングを表示および作成します。

- **Developer** パースペクティブを使用している場合:

a. ナビゲーションメニューの **Project** ページに移動します。

b. 画面上部の **Project** ドロップダウンメニューから **All Projects** を選択し、クラスター内のすべてのプロジェクトをリスト表示します。

c. 表示するプロジェクトを選択します。**Overview** タブには、プロジェクトのダッシュボードが含まれています。

d. **Details** タブを選択して、プロジェクトの詳細を表示します。

e. プロジェクトに対する適切なパーミッションがある場合は、**Project access** タブビューを選択し、プロジェクトの権限を更新します。

#### 2.1.2.2. CLI を使用した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。

**手順**

1. プロジェクトのリストを表示するには、以下を実行します。

```
$ oc get projects
```

2. 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>
```

### 2.1.3. Developer パースペクティブを使用したプロジェクトに対するアクセスパーミッションの提供

**Developer** パースペクティブで **Project** ビューを使用し、プロジェクトに対するアクセスを付与したり、取り消したりできます。

**前提条件**

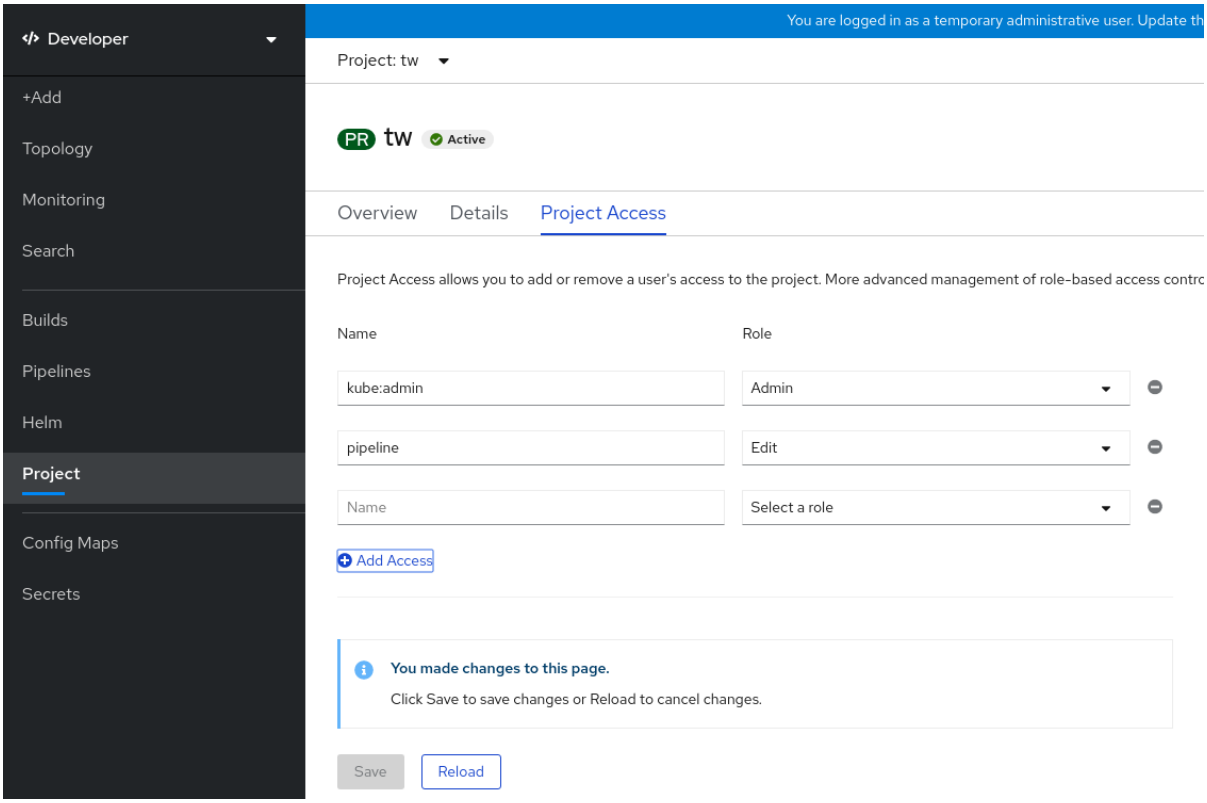
- プロジェクトを作成している。

**手順**

ユーザーをプロジェクトに追加し、**Admin**、**Edit**、または **View** アクセスをユーザーに付与するには、以下を実行します。

1. **Developer** パースペクティブで、**Project** ページに移動します。
2. **Project** メニューからプロジェクトを選択します。
3. **Project Access** タブを選択します。
4. **Add access** をクリックして、パーミッションの新規の行をデフォルトのパーミッションに追加します。

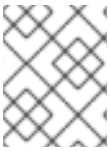
**図2.2 プロジェクトパーミッション**




5. ユーザー名を入力し、**Select a role** ドロップダウンリストをクリックし、適切なロールを選択します。
6. **Save** をクリックして新規パーミッションを追加します。

以下を使用することもできます。

- **Select a role** ドロップダウンリストを使用して、既存ユーザーのアクセスパーミッションを変更できます。
- **Remove Access** アイコンを使用して、既存ユーザーのプロジェクトへのアクセスパーミッションを完全に削除できます。



**注記**

高度なロールベースのアクセス制御は、**Administrator** パースペクティブの **Roles** および **Roles Binding** ビューで管理されます。

### 2.1.4. Web コンソールを使用した利用可能なクラスターのロールのカスタマイズ

Web コンソールの **Developer** パースペクティブでは、**Project → Project access** ページを使用して、プロジェクト管理者がプロジェクト内のユーザーにロールを付与できるようにします。デフォルトでは、プロジェクト内のユーザーに付与できるクラスターロールは、admin、edit、および view です。

クラスター管理者は、クラスター全体のすべてのプロジェクトに対して **Project access** ページでどのクラスターロールを使用できるかを定義できます。**Console** 設定リソースの **spec.customization.projectAccess.availableClusterRoles** オブジェクトをカスタマイズすることで、使用可能なロールを指定できます。

**前提条件**

- **cluster-admin** ロールを持つユーザーとしてクラスターにアクセスできる。

**手順**

1. **Administrator** パースペクティブで、**Administration → Cluster Settings** に移動します。
2. **Configuration** タブをクリックします。
3. **Configuration resource** リストから、**ConsoleOperator.openshift.io** を選択します。
4. **YAML** タブに移動し、YAML コードを表示し、編集します。
5. **spec** の YAML コードで、プロジェクトアクセスに使用可能なクラスターロールのリストをカスタマイズします。次の例では、デフォルトの **admin**、**edit**、および **view** ロールを指定します。

```
apiVersion: operator.openshift.io/v1
kind: Console
metadata:
  name: cluster
# ...
spec:
  customization:
    projectAccess:
      availableClusterRoles:
      - admin
      - edit
      - view
```

6. **Save** をクリックして、**Console** 設定リソースへの変更を保存します。

**検証**

1. **Developer** パースペクティブで、**Project** ページに移動します。
2. **Project** メニューからプロジェクトを選択します。
3. **Project access** タブを選択します。
4. **Role** 列のメニューをクリックし、使用可能なロールが **Console** リソース設定に適用した設定と一致することを確認します。

### 2.1.5. プロジェクトへの追加

Developer パースペクティブの +Add ページを使用して、プロジェクトに項目を追加できます。

**前提条件**

- プロジェクトを作成している。

**手順**

1. Developer パースペクティブで、+Add ページに移動します。
2. Project メニューからプロジェクトを選択します。
3. +Add ページで項目をクリックし、ワークフローに従います。

**注記**

また、**Add* ページの検索機能を使用して、プロジェクトに追加する追加アイテムを見つけます。画面上部の Add の下にある** をクリックし、検索フィールドにコンポーネントの名前を入力します。

### 2.1.6. プロジェクトのステータスの確認

OpenShift Container Platform Web コンソールまたは OpenShift CLI (**oc**) を使用して、プロジェクトのステータスを表示できます。

#### 2.1.6.1. Web コンソールを使用したプロジェクトのステータスの確認

Web コンソールを使用して、プロジェクトのステータスを確認できます。

**前提条件**

- プロジェクトを作成している。

**手順**

- Administrator パースペクティブを使用している場合:
  a. Home → Projects に移動します。
  b. 一覧からプロジェクトを選択します。
  c. Overview ページで、プロジェクトのステータスを確認します。
- Developer パースペクティブを使用している場合:
  a. Project ページに移動します。
  b. Project メニューからプロジェクトを選択します。
  c. Overview ページで、プロジェクトのステータスを確認します。

#### 2.1.6.2. CLI を使用したプロジェクトのステータスの確認

OpenShift CLI (**oc**) を使用して、プロジェクトのステータスを確認できます。

**前提条件**

- OpenShift CLI (**oc**) がインストールされている。
- プロジェクトを作成している。

**手順**

1. プロジェクトに切り替えます。

```
$ oc project <project_name> 1
```

1 **<project-name>** は、プロジェクト名に置き換えます。

2. プロジェクトの概要を取得します。

```
$ oc status
```

### 2.1.7. プロジェクトの削除

OpenShift Container Platform Web コンソールまたは OpenShift CLI (**oc**) を使用して、プロジェクトを削除できます。

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** 状態のプロジェクトからすべてのコンテンツをクリアしてから、最終的にプロジェクトを削除します。プロジェクトのステータスが **Terminating** の場合、新規のコンテンツをプロジェクトに追加することはできません。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。

#### 2.1.7.1. Web コンソールを使用したプロジェクトの削除

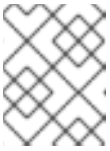
Web コンソールを使用してプロジェクトを削除できます。

**前提条件**

- プロジェクトを作成している。
- プロジェクトを削除するために必要なパーミッションを持っている。

**手順**

- **Administrator** パースペクティブを使用している場合:
  a. **Home → Projects** に移動します。
  b. 一覧からプロジェクトを選択します。
  c. プロジェクトの **Actions** ドロップダウンメニューをクリックし、**Delete Project** を選択します。



**注記**

プロジェクトを削除するために必要なパーミッションがない場合は、**Delete Project** オプションは選択できません。

1. **Delete Project?** ペインで、プロジェクトの名前を入力して削除を確認します。
2. **Delete** をクリックします。

- **Developer** パースペクティブを使用している場合:
  a. **Project** ページに移動します。
  b. **Project** メニューから削除するプロジェクトを選択します。
  c. プロジェクトの **Actions** ドロップダウンメニューをクリックし、**Delete Project** を選択します。

**注記**

プロジェクトを削除するために必要なパーミッションがない場合は、**Delete Project** オプションは選択できません。

1. **Delete Project?** ペインで、プロジェクトの名前を入力して削除を確認します。
2. **Delete** をクリックします。

#### 2.1.7.2. CLI を使用したプロジェクトの削除

OpenShift CLI (**oc**) を使用してプロジェクトを削除できます。

**前提条件**

- OpenShift CLI (**oc**) がインストールされている。
- プロジェクトを作成している。
- プロジェクトを削除するために必要なパーミッションを持っている。

**手順**

1. プロジェクトを削除します。

```
$ oc delete project <project_name> 1
```

1 **<project_name>** を、削除するプロジェクトの名前に置き換えます。

## 2.2. 別のユーザーとしてのプロジェクトの作成

権限の借用機能により、別のユーザーとしてプロジェクトを作成することができます。

### 2.2.1. API の権限借用

OpenShift Container Platform API への要求を、別のユーザーから発信されているかのように設定できます。詳細は、Kubernetes ドキュメントの User impersonation を参照してください。

### 2.2.2. プロジェクト作成時のユーザー権限の借用

プロジェクト要求を作成する際に別のユーザーの権限を借用できます。**system:authenticated:oauth** はプロジェクト要求を作成できる唯一のブートストラップグループであるため、そのグループの権限を借用する必要があります。

**手順**

- 別のユーザーの代わりにプロジェクト要求を作成するには、以下を実行します。

```
$ oc new-project <project> --as=<user> \
    --as-group=system:authenticated --as-group=system:authenticated:oauth
```

## 2.3. プロジェクト作成の設定

OpenShift Container Platform では、**プロジェクト** は関連するオブジェクトをグループ分けし、分離するために使用されます。Web コンソールまたは **oc new-project** コマンドを使用して新規プロジェクトの作成要求が実行されると、OpenShift Container Platform のエンドポイントは、カスタマイズ可能なテンプレートに応じてプロジェクトをプロビジョニングするために使用されます。

クラスター管理者は、開発者やサービスアカウントが独自のプロジェクトを作成し、プロジェクトの **セルフプロビジョニング** を実行することを許可し、その方法を設定できます。

### 2.3.1. プロジェクト作成について

OpenShift Container Platform API サーバーは、クラスターのプロジェクト設定リソースの **projectRequestTemplate** パラメーターで識別されるプロジェクトテンプレートに基づいて新規プロジェクトを自動的にプロビジョニングします。パラメーターが定義されない場合、API サーバーは要求される名前でプロジェクトを作成するデフォルトテンプレートを作成し、要求するユーザーをプロジェクトの **admin** (管理者) ロールに割り当てます。

プロジェクト要求が送信されると、API はテンプレートで以下のパラメーターを置き換えます。

**表2.1 デフォルトのプロジェクトテンプレートパラメーター**

| パラメーター | 説明 |
|---|---|
| **PROJECT_NAME** | プロジェクトの名前。必須。 |
| **PROJECT_DISPLAYNAME** | プロジェクトの表示名。空にできます。 |
| **PROJECT_DESCRIPTION** | プロジェクトの説明。空にできます。 |
| **PROJECT_ADMIN_USER** | 管理ユーザーのユーザー名。 |
| **PROJECT_REQUESTING_USER** | 要求するユーザーのユーザー名。 |

API へのアクセスは、**self-provisioner** ロールと **self-provisioners** のクラスターロールバインディングで開発者に付与されます。デフォルトで、このロールはすべての認証された開発者が利用できます。

### 2.3.2. 新規プロジェクトのテンプレートの変更

クラスター管理者は、デフォルトのプロジェクトテンプレートを変更し、新規プロジェクトをカスタム要件に基づいて作成することができます。

独自のカスタムプロジェクトテンプレートを作成するには、以下を実行します。

**手順**

1. **cluster-admin** 権限を持つユーザーとしてログインしている。
2. デフォルトのプロジェクトテンプレートを生成します。

```
$ oc adm create-bootstrap-project-template -o yaml > template.yaml
```

3. オブジェクトを追加するか、既存オブジェクトを変更することにより、テキストエディターで生成される **template.yaml** ファイルを変更します。
4. プロジェクトテンプレートは、**openshift-config** namespace に作成される必要があります。変更したテンプレートを読み込みます。

```
$ oc create -f template.yaml -n openshift-config
```

5. Web コンソールまたは CLI を使用し、プロジェクト設定リソースを編集します。
   - Web コンソールの使用
     i. **Administration → Cluster Settings** ページに移動します。
     ii. **Configuration** をクリックし、すべての設定リソースを表示します。
     iii. **Project** のエントリーを見つけ、**Edit YAML** をクリックします。
   - CLI の使用
     i. **project.config.openshift.io/cluster** リソースを編集します。

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

6. **spec** セクションを、**projectRequestTemplate** および **name** パラメーターを組み込むように更新し、アップロードされたプロジェクトテンプレートの名前を設定します。デフォルト名は **project-request** です。

**カスタムプロジェクトテンプレートを含むプロジェクト設定リソース**

```
apiVersion: config.openshift.io/v1
kind: Project
metadata:
# ...
spec:
```

```
  projectRequestTemplate:
    name: <template_name>
# ...
```

7. 変更を保存した後、変更が正常に適用されたことを確認するために、新しいプロジェクトを作成します。

### 2.3.3. プロジェクトのセルフプロビジョニングの無効化

認証されたユーザーグループによる新規プロジェクトのセルフプロビジョニングを禁止することができます。

**手順**

1. **cluster-admin** 権限を持つユーザーとしてログインしている。
2. 以下のコマンドを実行して、**self-provisioners** クラスターロールバインディングの使用を確認します。

```
$ oc describe clusterrolebinding.rbac self-provisioners
```

**出力例**

```
Name:  self-provisioners
Labels:  <none>
Annotations: rbac.authorization.kubernetes.io/autoupdate=true
Role:
  Kind: ClusterRole
  Name: self-provisioner
Subjects:
  Kind Name    Namespace
  ---- ----    ---------
  Group system:authenticated:oauth
```

**self-provisioners** セクションのサブジェクトを確認します。

3. **self-provisioner** クラスターロールをグループ **system:authenticated:oauth** から削除します。

- **self-provisioners** クラスターロールバインディングが **self-provisioner** ロールのみを **system:authenticated:oauth** グループにバインドする場合、以下のコマンドを実行します。

```
$ oc patch clusterrolebinding.rbac self-provisioners -p '{"subjects": null}'
```

- **self-provisioners** クラスターロールバインディングが **self-provisioner** ロールを **system:authenticated:oauth** グループ以外のユーザー、グループまたはサービスアカウントにバインドする場合、以下のコマンドを実行します。

```
$ oc adm policy \
    remove-cluster-role-from-group self-provisioner \
    system:authenticated:oauth
```

4. ロールへの自動更新を防ぐには、**self-provisioners** クラスターロールバインディングを編集します。自動更新により、クラスターロールがデフォルトの状態にリセットされます。

   - CLI を使用してロールバインディングを更新するには、以下を実行します。

     i. 以下のコマンドを実行します。

     ```
     $ oc edit clusterrolebinding.rbac self-provisioners
     ```

     ii. 表示されるロールバインディングで、以下の例のように **rbac.authorization.kubernetes.io/autoupdate** パラメーター値を **false** に設定します。

     ```
     apiVersion: authorization.openshift.io/v1
     kind: ClusterRoleBinding
     metadata:
       annotations:
         rbac.authorization.kubernetes.io/autoupdate: "false"
     # ...
     ```

   - 単一コマンドを使用してロールバインディングを更新するには、以下を実行します。

     ```
     $ oc patch clusterrolebinding.rbac self-provisioners -p '{ "metadata": { "annotations": {
     "rbac.authorization.kubernetes.io/autoupdate": "false" } } }'
     ```

5. 認証されたユーザーとしてログインし、プロジェクトのセルフプロビジョニングを実行できないことを確認します。

   ```
   $ oc new-project test
   ```

   **出力例**

   ```
   Error from server (Forbidden): You may not request a new project via this API.
   ```

   組織に固有のより有用な説明を提供できるようこのプロジェクト要求メッセージをカスタマイズすることを検討します。

### 2.3.4. プロジェクト要求メッセージのカスタマイズ

プロジェクトのセルフプロビジョニングを実行できない開発者またはサービスアカウントが Web コンソールまたは CLI を使用してプロジェクト作成要求を行う場合、以下のエラーメッセージがデフォルトで返されます。

```
You may not request a new project via this API.
```

クラスター管理者はこのメッセージをカスタマイズできます。これを、組織に固有の新規プロジェクトの要求方法の情報を含むように更新することを検討します。以下に例を示します。

- プロジェクトを要求するには、システム管理者 (**projectname@example.com**) に問い合わせてください。
- 新規プロジェクトを要求するには、**https://internal.example.com/openshift-project-request** にあるプロジェクト要求フォームに記入します。

プロジェクト要求メッセージをカスタマイズするには、以下を実行します。

**手順**

1. Web コンソールまたは CLI を使用し、プロジェクト設定リソースを編集します。
   - Web コンソールの使用
     i. **Administration → Cluster Settings** ページに移動します。
     ii. **Configuration** をクリックし、すべての設定リソースを表示します。
     iii. **Project** のエントリーを見つけ、**Edit YAML** をクリックします。
   - CLI の使用
     i. **cluster-admin** 権限を持つユーザーとしてログインしている。
     ii. **project.config.openshift.io/cluster** リソースを編集します。

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

2. **spec** セクションを、**projectRequestMessage** パラメーターを含むように更新し、値をカスタムメッセージに設定します。

   **カスタムプロジェクト要求メッセージを含むプロジェクト設定リソース**

   ```
   apiVersion: config.openshift.io/v1
   kind: Project
   metadata:
   # ...
   spec:
     projectRequestMessage: <message_string>
   # ...
   ```

   以下に例を示します。

   ```
   apiVersion: config.openshift.io/v1
   kind: Project
   metadata:
   # ...
   spec:
     projectRequestMessage: To request a project, contact your system administrator at
   projectname@example.com.
   # ...
   ```

3. 変更を保存した後に、プロジェクトをセルフプロビジョニングできない開発者またはサービスアカウントとして新規プロジェクトの作成を試行し、変更が正常に適用されていることを確認します。

# 第3章 アプリケーションの作成

## 3.1. DEVELOPER パースペクティブを使用したアプリケーションの作成

Web コンソールの **Developer** パースペクティブでは、**+Add** ビューからアプリケーションおよび関連サービスを作成し、それらを OpenShift Container Platform にデプロイするための以下のオプションが提供されます。

- **リソースの使用**: 開発者コンソールを使い始めるには、これらのリソースを使用します。**Options** メニュー  を使用してヘッダーを非表示にすることができます。
    - **サンプルを使用したアプリケーションの作成**: 既存のコードサンプルを使用して、OpenShift Container Platform でアプリケーションの作成を開始します。
    - **ガイド付きドキュメントを使用してビルド**: ガイド付きドキュメントを参照してアプリケーションを構築し、主なコンセプトや用語に慣れてください。
    - **新規開発者機能の確認**: **Developer** パースペクティブの新機能およびリソースを紹介します。
- **Developer catalog**: Developer Catalog で、イメージビルダーに必要なアプリケーション、サービス、またはソースを選択し、プロジェクトに追加します。
    - **All Services**: カタログを参照し、OpenShift Container Platform 全体でサービスを検出します。
    - **Database**: 必要なデータベースサービスを選択し、アプリケーションに追加します。
    - **Operator Backed**: 必要な Operator 管理サービスを選択し、デプロイします。
    - **Helm Chart**: 必要な Helm チャートを選択し、アプリケーションおよびサービスのデプロイメントを単純化します。
    - **Devfile**: **Devfile レジストリー**から devfile を選択して、開発環境を宣言的に定義します。
    - **Event Source**: 特定のシステムからイベントソースを選択し、関心のあるイベントクラスを登録します。

      **注記**

      RHOAS Operator がインストールされている場合には、マネージドサービスオプションも利用できます。
- **Git repository**: **From Git**、**From Devfile** または **From Dockerfile** オプションを使用して Git リポジトリーから既存のコードベース、Devfile、または Dockerfile をインポートし、OpenShift Container Platform でアプリケーションをビルドしてデプロイします。
- **Container Image**: イメージストリームまたはレジストリーからの既存イメージを使用し、これを OpenShift Container Platform にデプロイします。
- **Pipelines**: Tekton パイプラインを使用して OpenShift Container Platform でソフトウェア配信プロセスの CI/CD パイプラインを作成します。

- **Serverless**: Serverless オプションを検査して、OpenShift Container Platform でステートレスおよびサーバーレスアプリケーションを作成、ビルド、デプロイします。
    - **Channel**: Knative チャネルを作成し、インメモリーの信頼性の高い実装を備えたイベント転送および永続化層を作成します。
- **Samples**: 利用可能なサンプルアプリケーションを確認して、アプリケーションをすばやく作成、ビルド、デプロイします。
- **Quick Starts**: アプリケーションを作成、インポート、および実行するためのクイックスタートオプションを調べて、ステップバイステップの手順とタスクを使用します。
- **From Local Machine**: From Local Machine タイルを確認して、ローカルマシンのファイルをインポートまたはアップロードし、簡単にアプリケーションをビルドしてデプロイします。
    - **Import YAML**: YAML ファイルをアップロードし、アプリケーションをビルドしてデプロイするためのリソースを定義します。
    - **Upload JAR file**: JAR ファイルをアップロードして Java アプリケーションをビルドおよびデプロイします。
- **Share my Project**: このオプションを使用して、プロジェクトにユーザーを追加または削除し、アクセシビリティオプションを提供します。
- **Helm Chart リポジトリー**: このオプションを使用して、namespace に Helm Chart リポジトリーを追加します。
- **リソースの並べ替え**: これらのリソースを使用して、ナビゲーションペインに追加済みのピン留めされたリソースを並べ替えます。ナビゲーションウィンドウでピン留めされたリソースにカーソルを合わせると、その左側にドラッグアンドドロップアイコンが表示されます。ドラッグしたリソースは、それが属するセクションにのみドロップできます。

**Pipelines**、**Event Source**、**Import Virtual Machines** などの特定のオプションは、OpenShift Pipelines Operator、OpenShift Serverless Operator、および OpenShift Virtualization Operator がインストールされる場合にのみそれぞれ表示されることに注意してください。

### 3.1.1. 前提条件

**Developer** パースペクティブを使用してアプリケーションを作成するには、以下を確認してください。

- Web コンソールにログイン している。
- OpenShift Container Platform でアプリケーションおよび他のワークロードを作成するために、プロジェクトを作成しているか、適切な ロールと権限 を持つプロジェクトにアクセスできる。

前述の前提条件に加えてサーバーレスアプリケーションを作成するには、以下を確認します。

- OpenShift Serverless Operator がインストールされている。
- **knative-serving** namespace に **KnativeServing** リソースを作成している。

### 3.1.2. サンプルアプリケーションの作成

**Developer** パースペクティブの **+Add** フローでサンプルアプリケーションを使用し、アプリケーションをすぐに作成し、ビルドし、デプロイできます。

**前提条件**

- OpenShift Container Platform Web コンソールにログインしており、**Developer** パースペクティブにいます。

**手順**

1. **+Add** ビューで、**Samples** タイルをクリックして **Samples** ページを表示します。
2. **Samples** ページで、利用可能なサンプルアプリケーションの 1 つを選択し、**Create Sample Application** フォームを表示します。
3. **Create Sample Application Form**
   - **Name** フィールドには、デフォルトでデプロイメント名が表示されます。この名前は必要に応じて変更することができます。
   - **Builder Image Version** では、ビルダーイメージがデフォルトで選択されます。**Builder Image Version** ドロップダウンリストを使用してイメージバージョンを変更できます。
   - Git リポジトリー URL のサンプルは、デフォルトで追加されます。
4. **Create** をクリックしてサンプルアプリケーションを作成します。サンプルアプリケーションのビルドステータスが **Topology** ビューに表示されます。サンプルアプリケーションの作成後、デプロイメントがアプリケーションに追加されていることを確認できます。

### 3.1.3. Quick Starts を使用したアプリケーションの作成

**Quick Starts** ページでは、OpenShift Container Platform でアプリケーションを作成、インポート、および実行する方法を、段階的な手順とタスクとともに示します。

**前提条件**

- OpenShift Container Platform Web コンソールにログインしており、**Developer** パースペクティブにいます。

**手順**

1. **+Add** ビューで、**Getting Started resources→ Build with guided documentation → View all quick starts** リンクをクリックして、**Quick Starts** ページを表示します。
2. **Quick Starts** ページで、使用するクイックスタートのタイルをクリックします。
3. **Start** をクリックして、クイックスタートを開始します。
4. 表示される手順を実行します。

### 3.1.4. Git のコードベースのインポートおよびアプリケーションの作成

**Developer** パースペクティブを使用し、GitHub で既存のコードベースを使用して OpenShift Container Platform でアプリケーションを作成し、ビルドし、デプロイすることができます。

以下の手順では、**Developer** パースペクティブの **From Git** オプションを使用してアプリケーションを作成します。

**手順**

1. **+Add** ビューで、**Git Repository** タイルの **From Git** をクリックし、**Import from git** フォームを表示します。

2. **Git** セクションで、アプリケーションの作成に使用するコードベースの Git リポジトリー URL を入力します。たとえば、このサンプル nodejs アプリケーションの URL **https://github.com/sclorg/nodejs-ex** を入力します。その後、URL は検証されます。

3. オプション: **Show Advanced Git Options** をクリックし、以下のような詳細を追加できます。

   - **Git Reference**: アプリケーションのビルドに使用する特定のブランチ、タグ、またはコミットのコードを参照します。

   - **Context Dir**: アプリケーションのビルドに使用するアプリケーションのソースコードのサブディレクトリーを指定します。

   - **Source Secret** プライベートリポジトリーからソースコードをプルするための認証情報で **Secret Name** を作成します。

4. オプション: Git リポジトリーを使用して devfile、Dockerfile、またはビルダーイメージをインポートして、デプロイメントをさらにカスタマイズできるようになりました。

   - Git リポジトリーに devfile、Dockerfile、またはビルダーイメージが含まれる場合には、これらは自動的に検出され、それぞれのパスフィールドに設定されます。devfile、Dockerfile、およびビルダーイメージが同じリポジトリーで検出されると、devfile はデフォルトで選択されます。

   - ファイルのインポートタイプを編集して、別のストラテジーを選択し、**Edit import strategy** オプションをクリックします。

   - 複数の devfile、Dockerfile、またはビルダーイメージを検出された場合に、特定の devfile、Dockerfile、またはビルダーイメージをインポートするにはコンテキストディレクトリーを起点とした相対パスを指定します。

5. Git URL の検証後に、推奨されるビルダーイメージが選択されて星マークが付けられます。ビルダーイメージが自動検出されていない場合は、ビルダーイメージを選択します。**https://github.com/sclorg/nodejs-ex** Git URL の場合、Node.js ビルダーイメージがデフォルトで選択されます。

   a. オプション:**Builder Image Version** ドロップダウンリストを使用してバージョンを指定します。

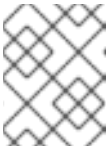
   b. オプション:**Edit import strategy** を使用して、別のストラテジーを選択します。

   c. オプション:Node.js ビルダーイメージの場合、**Run command** フィールドを使用して、アプリケーションを実行するためにコマンドを上書きします。

6. **General** セクションで、以下を実行します。

   a. **Application** フィールドに、アプリケーションを分類するために一意の名前 (**myapp** など) を入力します。アプリケーション名が namespace で一意であることを確認します。

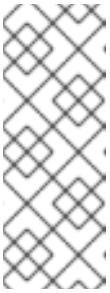
   b. **Name** フィールドで、既存のアプリケーションが存在しない場合に、このアプリケーション用に作成されたリソースが Git リポジトリー URL をベースとして自動的に設定されることを確認します。既存のアプリケーションがある場合には、既存のアプリケーション内でそのコンポーネントをデプロイしたり、新しいアプリケーションを作成したり、またはコンポーネントをいずれにも割り当てない状態にしたりすることができます。



**注記**

リソース名は namespace で一意である必要があります。エラーが出る場合はリソース名を変更します。

7. **Resources** セクションで、以下を選択します。
   - **Deployment**: 単純な Kubernetes スタイルのアプリケーションを作成します。
   - **Deployment Config**: OpenShift Container Platform スタイルのアプリケーションを作成します。
   - **Serverless Deployment**: Knative サービスを作成します。



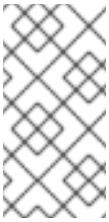
**注記**

ユーザー設定 ページを参照し、**アプリケーション → リソースタイプ** フィールドをクリックして、インポートのデフォルトのリソース設定を設定できます。**Serverless Deployment** オプションは、Serverless Operator がクラスターにインストールされている場合にのみ、**Import from Git** フォームに表示されます。詳細は、OpenShift Serverless のドキュメントを参照してください。

8. **Pipelines** セクションで、 **Add Pipeline** を選択してから **Show Pipeline Visualization** をクリックし、アプリケーションのパイプラインを表示します。デフォルトのパイプラインが選択されますが、アプリケーションで利用可能なパイプラインのリストから必要なパイプラインを選択できます。
9. オプション: **Advanced Options** セクションでは、**Target port** および **Create a route to the application** がデフォルトで選択されるため、公開されている URL を使用してアプリケーションにアクセスできます。
   アプリケーションがデフォルトのパブリックポート 80 でデータを公開しない場合は、チェックボックスの選択を解除し、公開する必要のあるターゲットポート番号を設定します。
10. オプション: 以下の高度なオプションを使用してアプリケーションをさらにカスタマイズできます。

    **Routing**

    **Routing** のリンクをクリックして、以下のアクションを実行できます。
    - ルートのホスト名をカスタマイズします。
    - ルーターが監視するパスを指定します。
    - ドロップダウンリストから、トラフィックのターゲットポートを選択します。
    - **Secure Route** チェックボックスを選択してルートを保護します。必要な TLS 終端タイプを選択し、各ドロップダウンリストから非セキュアなトラフィックについてのポリシーを設定します。



**注記**

サーバーレスアプリケーションの場合、Knative サービスが上記のすべてのルーティングオプションを管理します。ただし、必要に応じて、トラフィックのターゲットポートをカスタマイズできます。ターゲットポートが指定されていない場合、デフォルトポートの **8080** が使用されます。

**ドメインマッピング**

Serverless Deployment を作成する場合、作成時に Knative サービスにカスタムドメインマッピングを追加できます。

- Advanced options セクションで、Show advanced Routing options をクリックします。
  - サービスにマッピングするドメインマッピング CR がすでに存在する場合は、Domain mapping のドロップダウンメニューから選択できます。
  - 新規ドメインマッピング CR を作成する場合は、ドメイン名をボックスに入力し、Create オプションを選択します。たとえば、**example.com** と入力すると、Create オプションは Create "example.com" になります。

**ヘルスチェック**

Health Checks リンクをクリックして、Readiness、Liveness、および Startup プローブをアプリケーションに追加します。すべてのプローブに事前に設定されたデフォルトデータが実装され、必要に応じてデフォルトデータでプローブを追加したり、必要に応じてこれをカスタマイズしたりできます。
ヘルスプローブをカスタマイズするには、以下を実行します。

- Add Readiness Probe をクリックし、必要に応じてコンテナーが要求を処理する準備ができているかどうかを確認するためにパラメーターを変更し、チェックマークを選択してプローブを追加します。
- Add Liveness Probe をクリックし、必要に応じてコンテナーが実行中かどうかを確認するためにパラメーターを変更し、チェックマークを選択してプローブを追加します。
- Add Startup Probe をクリックし、必要に応じてコンテナー内のアプリケーションが起動しているかどうかを確認するためにパラメーターを変更し、チェックマークを選択してプローブを追加します。
  それぞれのプローブについて、ドロップダウンリストから要求タイプ (**HTTP GET**、**Container Command**、**TCP Socket**) を指定できます。選択した要求タイプに応じてフォームが変更されます。次に、プローブの成功および失敗のしきい値、コンテナーの起動後の最初のプローブ実行までの秒数、プローブの頻度、タイムアウト値など、他のパラメーターのデフォルト値を変更できます。

**ビルド設定およびデプロイメント**

Build Configuration および Deployment リンクをクリックして、それぞれの設定オプションを表示します。オプションの一部はデフォルトで選択されています。必要なトリガーおよび環境変数を追加して、オプションをさらにカスタマイズできます。
サーバーレスアプリケーションの場合、Deployment オプションは表示されません。これは、Knative 設定リソースが **DeploymentConfig** リソースの代わりにデプロイメントの必要な状態を維持するためです。

**スケーリング**

**Scaling** リンクをクリックして、最初にデプロイするアプリケーションの Pod 数またはインスタンス数を定義します。
サーバーレスデプロイメントを作成する場合、以下の設定を行うこともできます。

- **Min Pods** は、Knative サービスである時点で実行する必要がある Pod 数の下限を決定します。これは、**minScale** 設定としても知られています。
- **Max Pods** は、Knative サービスである時点で実行できる Pod 数の上限を決定します。これは、**maxScale** 設定としても知られています。
- **Concurrency target** は、ある時点でアプリケーションの各インスタンスに対して必要な同時リクエストの数を決定します。
- **Concurrency limit** は、ある時点でアプリケーションの各インスタンスに対して許容される同時リクエストの数の制限を決定します。
- **Concurrency utilization** は、Knative が追加のトラフィックを処理するために追加の Pod をスケールアップする際に満たす必要のある同時リクエストの制限のパーセンテージを決定します。
- **Autoscale window** は、Autoscaler がパニックモードではない場合に、スケーリングの決定を行う際のインプットを提供するためにメトリクスの平均値を計算する期間を定義します。この期間中にリクエストが受信されなかった場合、サービスはゼロにスケーリングされます。Autoscale window のデフォルト期間は **60s** です。これは stable window としても知られています。

**リソースの制限**

**Resource Limit** リンクをクリックして、コンテナーが実行時に保証または使用が許可されている **CPU** および **メモリー** リソースの量を設定します。

**ラベル**

**Labels** リンクをクリックして、カスタムラベルをアプリケーションに追加します。

11. **Create** をクリックしてアプリケーションを作成し、成功の通知が表示されます。**Topology** ビューでアプリケーションのビルドステータスを確認できます。

### 3.1.5. JAR ファイルをアップロードして Java アプリケーションをデプロイする

Web コンソールの **Developer** パースペクティブで、以下のオプションを使用して JAR ファイルをアップロードできます。

- **Developer** パースペクティブの **+Add** ビューに移動し、**From Local Machine** タイルで **Upload JAR file** をクリックします。JAR ファイルを参照および選択するか、JAR ファイルをドラッグしてアプリケーションをデプロイします。
- **Topology** ビューに移動し、**Upload JAR file** オプションを使用するか、JAR ファイルをドラッグしてアプリケーションをデプロイします。
- **Topology** ビューのコンテキストメニューで **Upload JAR file** オプションを使用して JAR ファイルをアップロードしてアプリケーションをデプロイします。

**前提条件**

- クラスター管理者が Cluster Samples Operator をインストールしている。

- OpenShift Container Platform Web コンソールにアクセスでき、**Developer** パースペクティブを使用している。

**手順**

1. **Topology** ビューで、任意の場所を右クリックして **Add to Project** メニューを表示します。
2. **Add to Project** メニューにカーソルを置いてメニューオプションを表示し、**Upload JAR file** オプションを選択して **Upload JAR file** フォームを確認します。または、JAR ファイルを **Topology** ビューにドラッグできます。
3. **JAR file** フィールドで、ローカルマシンで必要な JAR ファイルを参照し、これをアップロードします。または、JAR ファイルを フィールドにドラッグできます。互換性のないタイプのファイルが **Topology** ビューにドラッグされると、トーストアラートが右側に表示されます。互換性のないファイルタイプがアップロードフォームのフィールドにドロップされると、フィールドエラーが表示されます。
4. デフォルトで、ランタイムアイコンとビルダーイメージが選択されています。ビルダーイメージが自動検出されていない場合は、ビルダーイメージを選択します。必要に応じて、**Builder Image Version** のドロップダウンリストを使用してバージョンを変更できます。
5. オプション: **Application Name** フィールドに、リソースのラベル付けに使用する一意のアプリケーション名を入力します。
6. **Name** フィールドに、関連付けられたリソースに名前を付けるために一意のコンポーネント名を入力します。
7. オプション: **Advanced options → Resource type** ドロップダウンリストを使用して、デフォルトのリソースタイプのリストから別のリソースタイプを選択します。
8. **Advanced options** メニューで **Create a Route to the Application**をクリックし、デプロイされたアプリケーションのパブリック URL を設定します。
9. **Create** をクリックしてアプリケーションをデプロイします。JAR ファイルがアップロードされたことを通知するトースト通知が表示されます。トースト通知には、ビルドログを表示するリンクも含まれます。

**注記**

ビルドの実行中にブラウザータブを閉じようとすると、Web アラートが表示されます。

JAR ファイルのアップロードとアプリケーションのデプロイメントが完了すると、**Topology** ビューにアプリケーションが表示されます。

### 3.1.6. Devfile レジストリーを使用した devfile へのアクセス

**Developer** パースペクティブの **+Add** フローで devfile を使用して、アプリケーションを作成できます。**+Add** フローは、devfile コミュニティーレジストリー との完全なインテグレーションを提供します。devfile は、ゼロから設定せずに開発環境を記述できる移植可能な YAML ファイルです。**Devfile レジストリー** を使用すると、事前に設定された devfile を使用してアプリケーションを作成できます。

**手順**

1. **Developer Perspective → +Add → Developer Catalog → All Services** に移動します。**Developer Catalog** で利用可能なすべてのサービスの一覧が表示されます。

2. **Type** で、**Devfiles** をクリックして、特定の言語またはフレームワークをサポートする devfiles を参照します。あるいは、キーワードフィルターを使用して、名前、タグ、または説明を使用して特定の devfile を検索できます。

3. アプリケーションの作成に使用する devfile をクリックします。devfile タイルに、devfile の名前、説明、プロバイダー、および ドキュメントなど、devfile の詳細が表示されます。

4. **Create** をクリックしてアプリケーションを作成し、**Topology** ビューでアプリケーションを表示します。

### 3.1.7. Developer Catalog を使用したサービスまたはコンポーネントのアプリケーションへの追加

Developer Catalog を使用して、データベース、ビルダーイメージ、Helm チャートなどの Operator がサポートするサービスに基づいてアプリケーションとサービスをデプロイします。Developer Catalog には、プロジェクトに追加できるアプリケーションコンポーネント、サービス、イベントソース、または Source-to-Image ビルダーのコレクションが含まれます。クラスター管理者は、カタログで利用可能なコンテンツをカスタマイズできます。

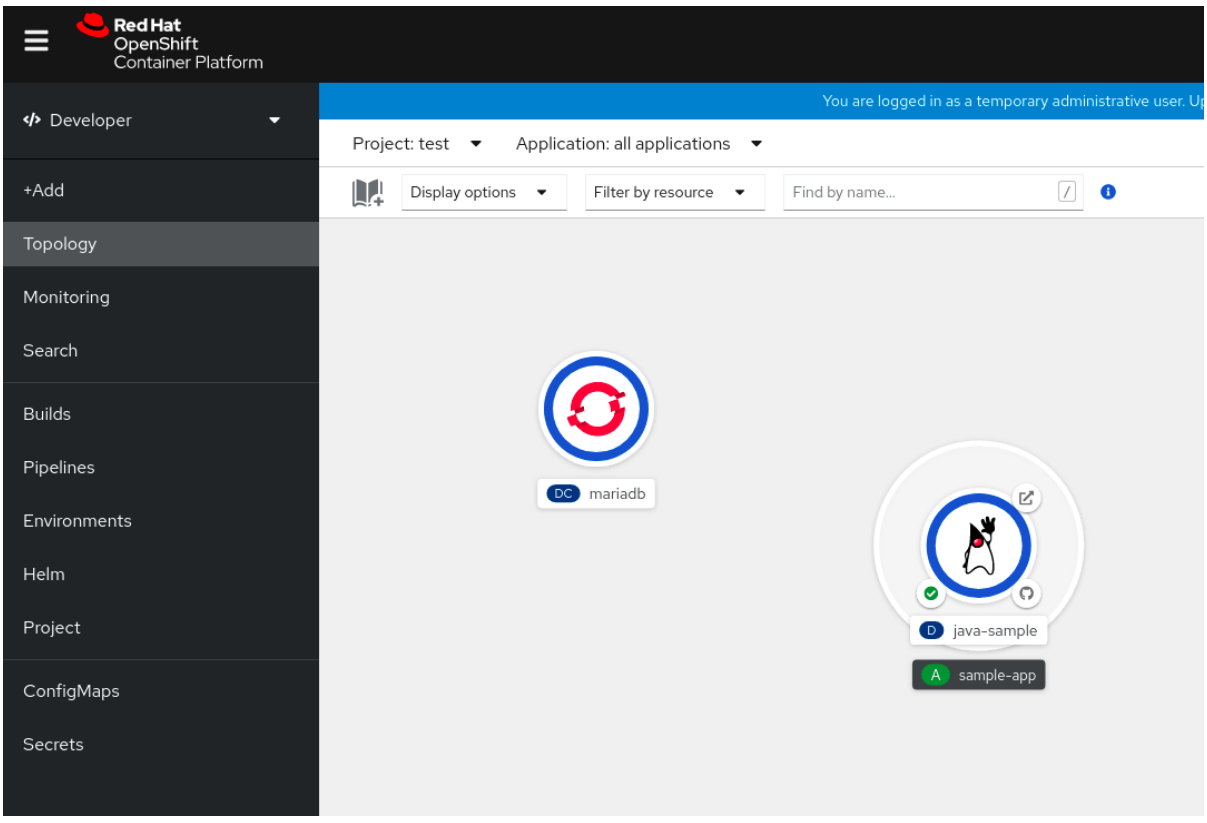
**手順**

1. **Developer** パースペクティブで、**+Add** に移動して、**Developer Catalog** タイルから **All Services** をクリックし、**Developer Catalog** で利用可能なすべてのサービスを表示します。

2. **All Services** で、サービスの種類またはプロジェクトに追加する必要のあるコンポーネントを選択します。この例では、**Databases** を選択してすべてのデータベースサービスを一覧表示し、**MariaDB** をクリックしてサービスの詳細を表示します。

3. **Instantiate Template** をクリックして、**MariaDB** サービスの詳細情報を含む自動的に設定されたテンプレートを表示し、**Create** をクリックして **Topology** ビューで MariaDB サービスを作成し、これを表示します。

**図3.1 トポロジーの MariaDB**



### 3.1.8. 関連情報

- OpenShift Serverless の Knative ルーティング設定についての詳細は、Routingを参照してください。
- OpenShift Serverless のドメインマッピング設定についての詳細は、Configuring a custom domain for a Knative serviceを参照してください。
- OpenShift Serverless の Knative 自動スケーリング設定についての詳細は、Autoscalingを参照してください。
- プロジェクトに新規ユーザーを追加する方法の詳細は、プロジェクトの使用 を参照してください。
- Helm チャートリポジトリーの作成の詳細は Helm Chart リポジトリーの作成 を参照してください。

## 3.2. インストールされた OPERATOR からのアプリケーションの作成

**Operator** は、Kubernetes アプリケーションをパッケージ化し、デプロイし、管理する方法です。クラスター管理者によってインストールされる Operator を使用して、アプリケーションを OpenShift Container Platform で作成できます。

以下では、開発者を対象に、OpenShift Container Platform Web コンソールを使用して、インストールされた Operator からアプリケーションを作成する例を示します。

**関連情報**

- Operator の仕組みおよび Operator Lifecycle Manager の OpenShift Container Platform への統合方法に関する詳細は、Operator ガイドを参照してください。

### 3.2.1. Operator を使用した etcd クラスターの作成

この手順では、Operator Lifecycle Manager (OLM) で管理される etcd Operator を使用した新規 etcd クラスターの作成について説明します。

**前提条件**

- OpenShift Container Platform 4.12 クラスターにアクセスできる
- 管理者によってクラスター全体に etcd Operator がすでにインストールされている。

**手順**

1. この手順を実行するために OpenShift Container Platform Web コンソールで新規プロジェクトを作成します。この例では、**my-etcd** というプロジェクトを使用します。
2. **Operators → Installed Operators** ページに移動します。クラスター管理者によってクラスターにインストールされ、使用可能にされた Operator がクラスターサービスバージョン (CSV) のリストとしてここに表示されます。CSV は Operator によって提供されるソフトウェアを起動し、管理するために使用されます。

**ヒント**

以下を使用して、CLI でこのリストを取得できます。

```
$ oc get csv
```

3. **Installed Operators** ページで、etcd Operator をクリックして詳細情報および選択可能なアクションを表示します。
**Provided APIs** に表示されているように、この Operator は 3 つの新規リソースタイプを利用可能にします。これには、**etcd クラスター** (**EtcdCluster** リソース) のタイプが含まれます。これらのオブジェクトは、**Deployment** または **ReplicaSet** などの組み込み済みのネイティブ Kubernetes オブジェクトと同様に機能しますが、これらには etcd を管理するための固有のロジックが含まれます。

4. 新規 etcd クラスターを作成します。

   a. **etcd Cluster** API ボックスで、**Create instance** をクリックします。

   b. 次の画面では、クラスターのサイズなど **EtcdCluster** オブジェクトのテンプレートを起動する最小条件への変更を加えることができます。ここでは **Create** をクリックして確定します。これにより、Operator がトリガーされ、Pod、サービス、および新規 etcd クラスターの他のコンポーネントが起動します。

5. **example** etcd クラスターをクリックしてから **Resources** タブをクリックして、プロジェクトに Operator によって自動的に作成され、設定された数多くのリソースが含まれることを確認します。
Kubernetes サービスが作成され、プロジェクトの他の Pod からデータベースにアクセスできることを確認します。

6. 所定プロジェクトで **edit** ロールを持つすべてのユーザーは、クラウドサービスのようにセルフサービス方式でプロジェクトにすでに作成されている Operator によって管理されるアプリケーションのインスタンス (この例では etcd クラスター) を作成し、管理し、削除することができます。この機能を持つ追加のユーザーを有効にする必要がある場合、プロジェクト管理者は以下のコマンドを使用してこのロールを追加できます。

```
$ oc policy add-role-to-user edit <user> -n <target_project>
```

これで、etcd クラスターは Pod が正常でなくなったり、クラスターのノード間で移行する際の障害に対応し、データのリバランスを行います。最も重要な点として、適切なアクセスを持つクラスター管理者または開発者は独自のアプリケーションでデータベースを簡単に使用できるようになります。

## 3.3. CLI を使用したアプリケーションの作成

OpenShift Container Platform CLI 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を含むコンポーネントから OpenShift Container Platform アプリケーションを作成できます。

**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。

### 3.3.1. 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。

**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、**Deployment** オブジェクト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、パイプライン、ソース、または docker 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、ソースビルドの場合は、適切な言語のビルダーイメージを検出します。

#### 3.3.1.1. Local

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** コマンドを実行してバイナリービルドを作成します。

#### 3.3.1.2. リモート

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、ビルド設定に挿入され、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
```

#### 3.3.1.3. ビルドストラテジーの検出

OpenShift Container Platform は、特定のファイルを検出し、使用するビルドストラテジーを自動的に判別します。

- 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 はパイプラインビルドストラテジーを生成します。

**注記**

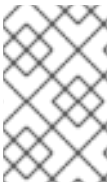
**pipeline** ビルドストラテジーは非推奨になりました。代わりに Red Hat OpenShift Pipelines を使用することを検討してください。

- 新規アプリケーションの作成時に Dockerfile 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 docker ビルドストラテジーを生成します。

- Jenkins ファイルも Dockerfile も検出されない場合、OpenShift Container Platform はソースビルドストラテジーを生成します。

**--strategy** フラグを **docker**、**pipeline**、または **source** に設定して、自動的に検出されたビルドストラテジーを上書きします。

```
$ 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。すべてのソースビルドには、**git remote -v** を使用する必要があります。

#### 3.3.1.4. 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3.1 new-app が検出する言語**

| 言語 | ファイル |
|---|---|
| **dotnet** | **project.json**、***.csproj** |
| **jee** | **pom.xml** |
| **nodejs** | **app.json**、**package.json** |
| **perl** | **cpanfile**、**index.pl** |
| **php** | **composer.json**、**index.php** |

| 言語 | ファイル |
| --- | --- |
| **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、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。

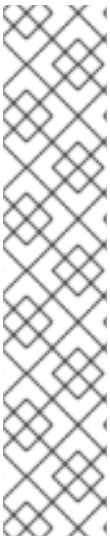
~ をセパレーターとして使用し、イメージ (イメージストリームまたはコンテナーの仕様) とリポジトリーを指定して、ビルダーが特定のソースリポジトリーを使用するようにイメージを上書きすることができます。この方法を使用すると、ビルドストラテジーの検出および言語の検出は実行されない点に留意してください。

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
```



**注記**

言語の検出では、リポジトリーのクローンを作成し、検査できるように Git クライアントをローカルにインストールする必要があります。Git が使用できない場合、**<image>~<repository>** 構文を指定し、リポジトリーで使用するビルダーイメージを指定して言語の検出手順を回避することができます。

**-i <image> <repository>** 呼び出しでは、アーティファクトのタイプを判別するために **new-app** が **repository** のクローンを試行する必要があります。そのため、これは Git が利用できない場合には失敗します。

**-i <image> --code <repository>** 呼び出しでは、**image** がソースコードのビルダーとして使用されるか、データベースイメージの場合のように別個にデプロイされる必要があるかどうかを判別するために、**new-app** が **repository** のクローンを作成する必要があります。

### 3.3.2. 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、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** 引数を使用したコンテナーイメージなのか、**-i|--image-stream** 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。

#### 3.3.2.1. Docker Hub MySQL イメージ

たとえば、Docker Hub 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
```

#### 3.3.2.2. プライベートレジストリーのイメージ

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し、コンテナー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
```

#### 3.3.2.3. 既存のイメージストリームおよびオプションのイメージストリームタグ

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
```

### 3.3.3. 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。

現在のプロジェクトのテンプレートライブラリーにアプリケーションテンプレートをアップロードします。以下の例では、**examples/sample-app/application-template-stibuild.json** というファイルからアプリケーションテンプレートをアップロードします。

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

次に、アプリケーションテンプレートを参照して新規アプリケーションを作成します。この例では、テンプレート名は **ruby-helloworld-sample** です。

```
$ oc new-app ruby-helloworld-sample
```

OpenShift Container Platform にテンプレートファイルを保存せずに、ローカルファイルシステムでテンプレートファイルを参照して新規アプリケーションを作成するには、**-f|--file** 引数を使用します。以下に例を示します。

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

#### 3.3.3.1. 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必要がある場合は、以下のように **--param-file=-** を使用します。以下は、**helloworld.params** というファイルの例です。

```
ADMIN_USERNAME=admin
ADMIN_PASSWORD=mypassword
```

テンプレートをインスタンス化する時に、ファイルのパラメーターを参照します。

```
$ oc new-app ruby-helloworld-sample --param-file=helloworld.params
```

### 3.3.4. アプリケーション作成の変更

**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現在のプロジェクトに作成され、これらのオブジェクトに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が割り当てられます。ただし、**new-app** でこの動作を変更することができます。

**表3.2 new-app 出力オブジェクト**

| オブジェクト | 説明 |
|---|---|
| **BuildConfig** | **BuildConfig** オブジェクトは、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** オブジェクト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** オブジェクトでは、通常 2 つのイメージストリームが作成されます。1 つ目は、インプットイメージを表します。ソースビルドの場合、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** オブジェクトは、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、単純に **oc expose** コマンドを使用し、追加のサービスを生成することができます。 |

| オブジェクト | 説明 |
| --- | --- |
| 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。 |

#### 3.3.4.1. 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。以下は、**postgresql.env** というファイルの例です。

```
POSTGRESQL_USER=user
POSTGRESQL_DATABASE=db
POSTGRESQL_PASSWORD=password
```

ファイルから変数を読み取ります。

```
$ 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=-
```

**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。

#### 3.3.4.2. 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。以下は、**ruby.env** というファイルの例です。

```
HTTP_PROXY=http://myproxy.net:1337/
GEM_HOME=~/.gem
```

ファイルから変数を読み取ります。

```
$ 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=-
```

#### 3.3.4.3. 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
```

#### 3.3.4.4. 作成前の出力の表示

**new-app** コマンドの実行に関する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、以下を実行します。

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

ファイルを編集します。

```
$ vi myapp.yaml
```

ファイルを参照して新規アプリケーションを作成します。

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

#### 3.3.4.5. 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
```

#### 3.3.4.6. 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、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。

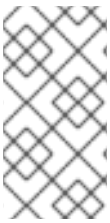
```
$ oc new-app https://github.com/openshift/ruby-hello-world -n myproject
```

#### 3.3.4.7. 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。

#### 3.3.4.8. 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。グループ化するイメージを指定するには **+** 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
```

#### 3.3.4.9. 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
```

# 第4章 TOPOLOGY ビューを使用したアプリケーション構成の表示

Web コンソールの **Developer** パースペクティブにある **Topology** ビューは、プロジェクト内のすべてのアプリケーション、それらのビルドステータスおよびアプリケーションに関連するコンポーネントとサービスを視覚的に表示します。

## 4.1. 前提条件

**Topology** ビューでアプリケーションを表示し、それらと対話するには、以下を確認します。

- Web コンソールにログイン している。
- OpenShift Container Platform でアプリケーションおよび他のワークロードを作成するための適切なプロジェクト内の ロールおよび権限 がある。
- **Developer** パースペクティブを使用して OpenShift Container Platform でアプリケーションを作成し、デプロイしている。
- **Developer** パースペクティブを使用している。

## 4.2. アプリケーションのトポロジーの表示

**Developer** パースペクティブの左側のナビゲーションパネルを使用すると、**Topology** ビューに移動できます。アプリケーションをデプロイしたら、**Graph view** に自動的に移動します。ここでは、アプリケーション Pod のステータスの確認、パブリック URL でのアプリケーションへの迅速なアクセス、ソースコードへのアクセスとその変更、最終ビルドのステータスの確認ができます。ズームインおよびズームアウトにより、特定のアプリケーションの詳細を表示することができます。

**Topology** ビューは、**List** ビューを使用してアプリケーションを監視するオプションも提供します。**List view** アイコン (  ) を使用してすべてのアプリケーションの一覧を表示し、 **Graph view** アイコン (  ) を使用してグラフビューに戻します。

以下を使用して、必要に応じてビューをカスタマイズできます。

- **Find by name** フィールドを使用して、必要なコンポーネントを見つけます。検索結果は表示可能な領域外に表示される可能性があります。その場合、画面の左下のツールバーで **Fit to Screen** をクリックし、**Topology** ビューのサイズを変更して、すべてのコンポーネントを表示します。
- **Display Options** ドロップダウンリストを使用して、各種アプリケーショングループの **Topology** ビューを設定します。選択可能なオプションは、プロジェクトにデプロイされるコンポーネントのタイプによって異なります。
  - **Expand** グループ
    - Virtual Machines: 仮想マシンを表示または非表示にするためにこれを切り替えます。
    - Application Groupings: アプリケーショングループとそれに関連するアラートの概要を使用して、アプリケーショングループをカードにまとめるには、これをクリアします。
    - Helm Releases: 指定のリリースの概要を使用して、Helm リリースとしてデプロイされたコンポーネントをカードにまとめるには、これをクリアします。

- Knative Services: 指定のコンポーネントの概要を使用して Knative Service コンポーネントをカードにまとめるには、これをクリアします。
- Operator Groupings: 指定のグループの概要を使用して Operator でデプロイされたコンポーネントをカードにまとめるには、これをクリアします。

- **Pod 数** または **ラベル**に基づく **Show** の要素
  - Pod Count: コンポーネントアイコンでコンポーネントの Pod 数を表示するためにこれを選択します。
  - Labels: コンポーネントラベルを表示または非表示にするためにこれを選択します。

**トポロジー** ビューには、アプリケーションを ZIP ファイル形式でダウンロードするための **アプリケーションのエクスポート** オプションも用意されています。その後、ダウンロードしたアプリケーションを別のプロジェクトまたはクラスターにインポートできます。詳細については、**追加リソース** セクションの **別のプロジェクトまたはクラスターへのアプリケーションのエクスポート** を参照してください。

## 4.3. アプリケーションおよびコンポーネントとの対話

Web コンソールの **Developer** パースペクティブの **Topology** ビューでは、**Graph view** に、アプリケーションおよびコンポーネントと対話するための次のオプションが提供されます。

- **Open URL** ( ) をクリックして、パブリック URL のルートで公開されるアプリケーションを表示します。
- **Edit Source code** をクリックして、ソースコードにアクセスし、これを変更します。

**注記**

この機能は、**From Git**、**From Catalog**、および **From Dockerfile** オプションを使用してアプリケーションを作成する場合にのみ利用できます。

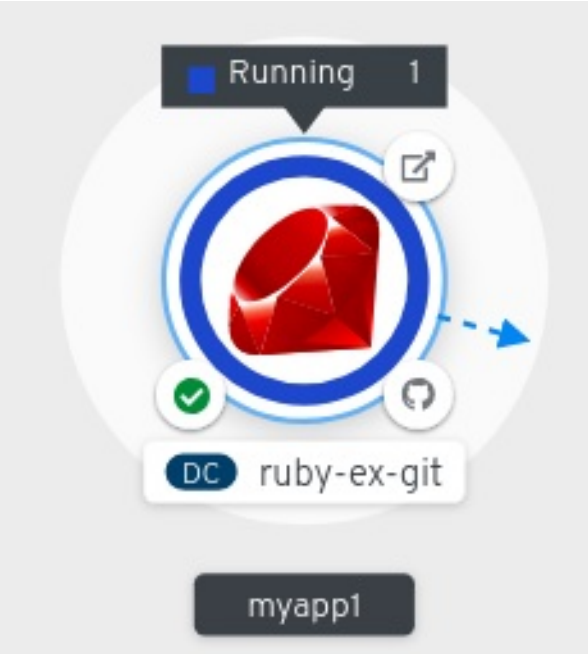
- カーソルを Pod の左下のアイコンの上に置き、最新ビルドおよびそのステータスを確認します。アプリケーションビルドのステータスは、**New** ( )、**Pending** ()、**Running** ( )、**Completed** ( )、**Failed** ( )、および **Canceled** ( ) と表示されます。
- Pod のステータスまたはフェーズは、色で区別され、ツールチップで次のように表示されます。
  - **Running** ( ): Pod はノードにバインドされ、すべてのコンテナーが作成されます。1 つ以上のコンテナーが実行中か、起動または再起動のプロセスが実行中です。
  - **Not Ready**( ): 複数のコンテナーを実行している Pod。すべてのコンテナーが準備状態にある訳ではありません。
  - **Warning**( ): Pod のコンテナーは終了されていますが、正常に終了しませんでした。一部のコンテナーは、他の状態にある場合があります。
  - **Failed**( ): Pod 内のすべてのコンテナーは終了しますが、少なくとも 1 つのコンテナーが終了に失敗しました。つまり、コンテナーはゼロ以外のステータスで終了するか、システムによって終了された状態であるかのいずれかになります。
  - **Pending**( ): Pod は Kubernetes クラスターによって受け入れられますが、1 つ以上のコンテナーが設定されておらず、実行される準備が整っていません。これには、Pod がスケ

ジュールされるのを待機する時間や、ネットワーク経由でコンテナーイメージのダウンロードに費やされた時間が含まれます。

- **Succeeded**( ): Pod のすべてのコンテナーが正常に終了し、再起動されません。
- **Terminating**( ): Pod が削除されている場合に、一部の kubectl コマンドによって **Terminating** と表示されます。**Terminating** ステータスは Pod フェーズのいずれにもありません。Pod には正常な終了期間が付与されます。これはデフォルトで 30 秒に設定されます。
- **Unknown**( ): Pod の状態を取得できませんでした。このフェーズは、通常、Pod が実行されているノードとの通信でエラーが発生するために生じます。

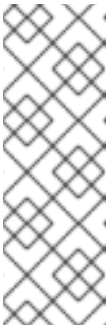
- アプリケーションを作成し、イメージがデプロイされると、ステータスは **Pending** と表示されます。アプリケーションをビルドすると、**Running**と表示されます。

**図4.1 Application トポロジー**




以下のように、異なるタイプのリソースオブジェクトのインジケーターと共に、アプリケーションリソース名が追加されます。

- CJ: **CronJob**
- D: **Deployment**
- DC: **DeploymentConfig**
- DS: **DaemonSet**
- J: **Job**
- P: **Pod**
- SS: **StatefulSet**
- (Knative): サーバーレスアプリケーション



**注記**

サーバーレスアプリケーションでは、**Graph view** での読み込みおよび表示にしばらく時間がかかります。サーバーレスアプリケーションをデプロイすると、これは最初にサービスリソースを作成し、次にリビジョンを作成します。続いて、これは **Graph view** にデプロイされ、表示されます。これが唯一のワークロードの場合には、**Add** ページにリダイレクトされる可能性があります。リビジョンがデプロイされると、サーバーレスアプリケーションは **Graph view** ビューに表示されます。

## 4.4. アプリケーション POD のスケーリングおよびビルドとルートの確認

**Topology** ビューは、**Overview** パネルでデプロイ済みのコンポーネントの詳細を提供します。次のように、**Overview** タブと **Details** タブを使用して、アプリケーション Pod をスケーリングし、ビルドステータス、サービス、およびルートを確認できます。

- コンポーネントノードをクリックし、右側の **Overview** パネルを確認します。**Details** タブを使用して以下を行います。
  - 上下の矢印を使用して Pod をスケーリングし、アプリケーションのインスタンス数の増減を手動で調整します。サーバーレスアプリケーションの場合、Pod は、チャネルのトラフィックに基づいてアイドルおよびスケールアップ時に自動的にゼロにスケーリングされます。
  - アプリケーションの **ラベル**、**アノテーション** および **ステータス** を確認します。
- **Resources** タブをクリックして、以下を実行します。
  - すべての Pod のリストを確認し、それらのステータスを表示し、ログにアクセスし、Pod をクリックして Pod の詳細を表示します。
  - ビルド、ステータスを確認し、ログにアクセスし、必要に応じて新規ビルドを開始します。
  - コンポーネントによって使用されるサービスとルートを確認します。

  サーバーレスアプリケーションの場合、**Resources** タブは、そのコンポーネントに使用されるリビジョン、ルート、および設定に関する情報を提供します。

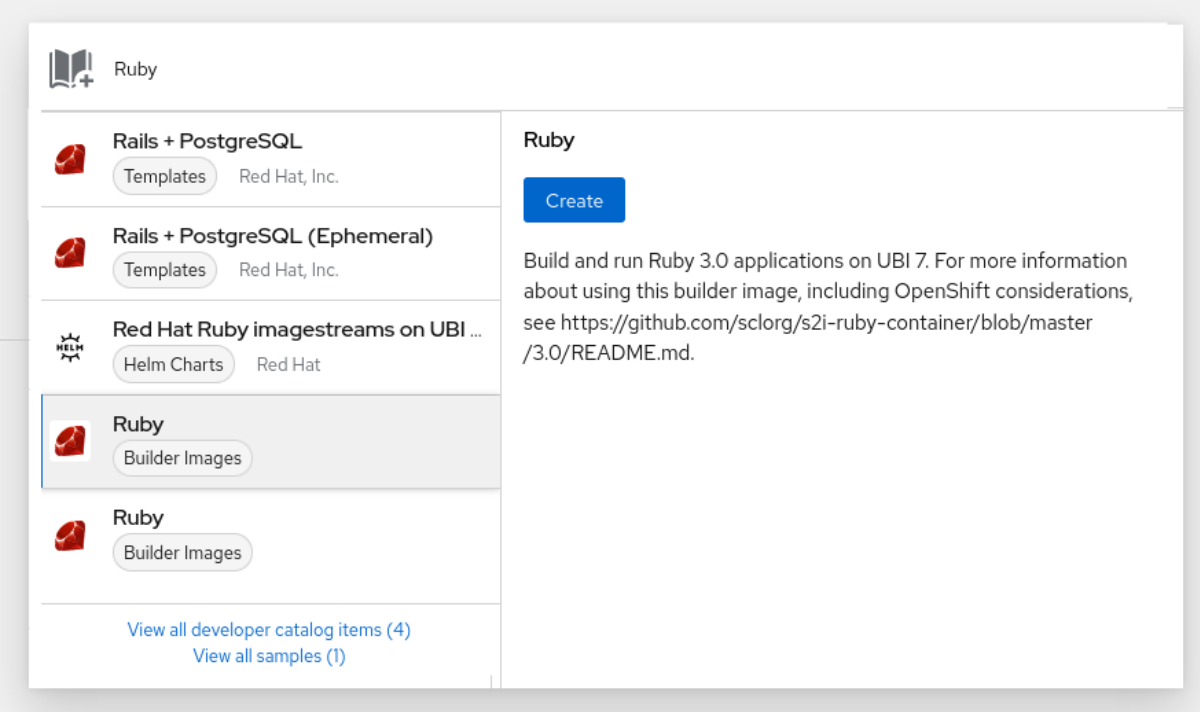
## 4.5. コンポーネントの既存プロジェクトへの追加

プロジェクトにコンポーネントを追加できます。

**手順**

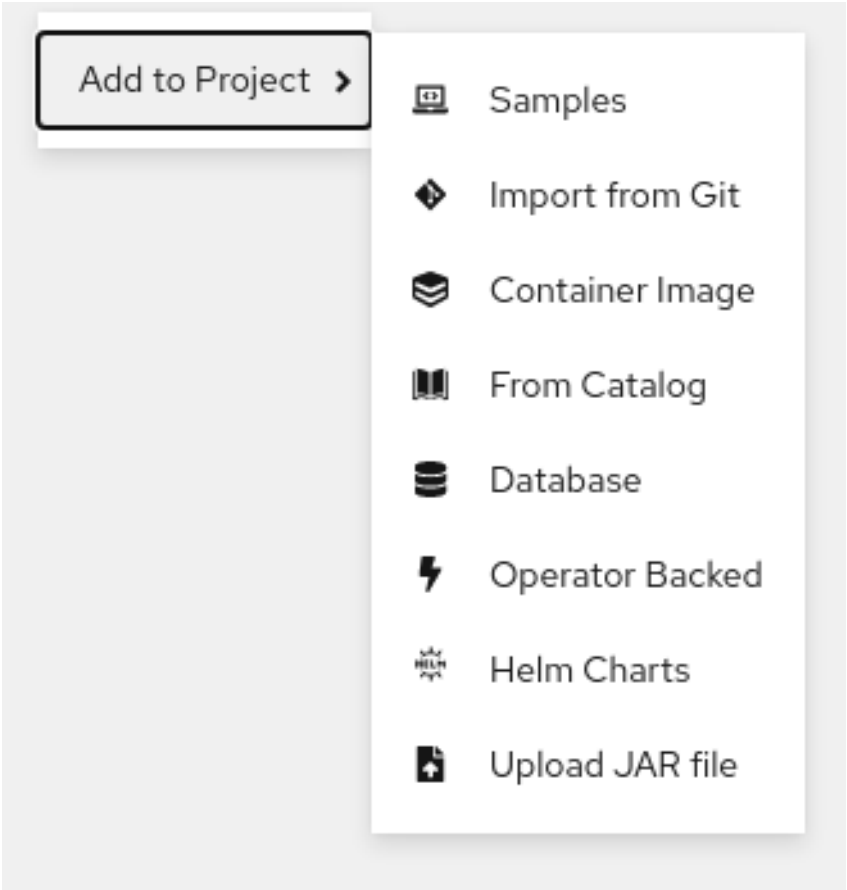
1. **+Add** ビューに移動します。
2. **Add to Project** (  ) をクリックし、左側のナビゲーションペインまたは **Ctrl**+**Space**を押します。
3. コンポーネントを検索し、**Start**/**Create**/**Install** ボタンをクリックするか、**Enter** をクリックしてコンポーネントをプロジェクトに追加し、トポロジー **Graph view** で確認します。

**図4.2 クイック検索を使用したコンポーネントの追加**




あるいは、トポロジーの **Graph view** を右クリックして、**Import from Git**、**Container Image**、**Database**、**From Catalog**、**Operator Backed**、**Helm Charts**、**Samples** または **Upload JAR file** などのコンテキストメニューの利用可能なオプションを使用して、プロジェクトにコンポーネントを追加することもできます。

**図4.3 サービスを追加するコンテキストメニュー**



## 4.6. アプリケーション内での複数コンポーネントのグループ化

+Add ビューを使用して、複数のコンポーネントまたはサービスをプロジェクトに追加し、**Topology** ビューを使用してアプリケーショングループ内のアプリケーションとリソースをグループ化できます。

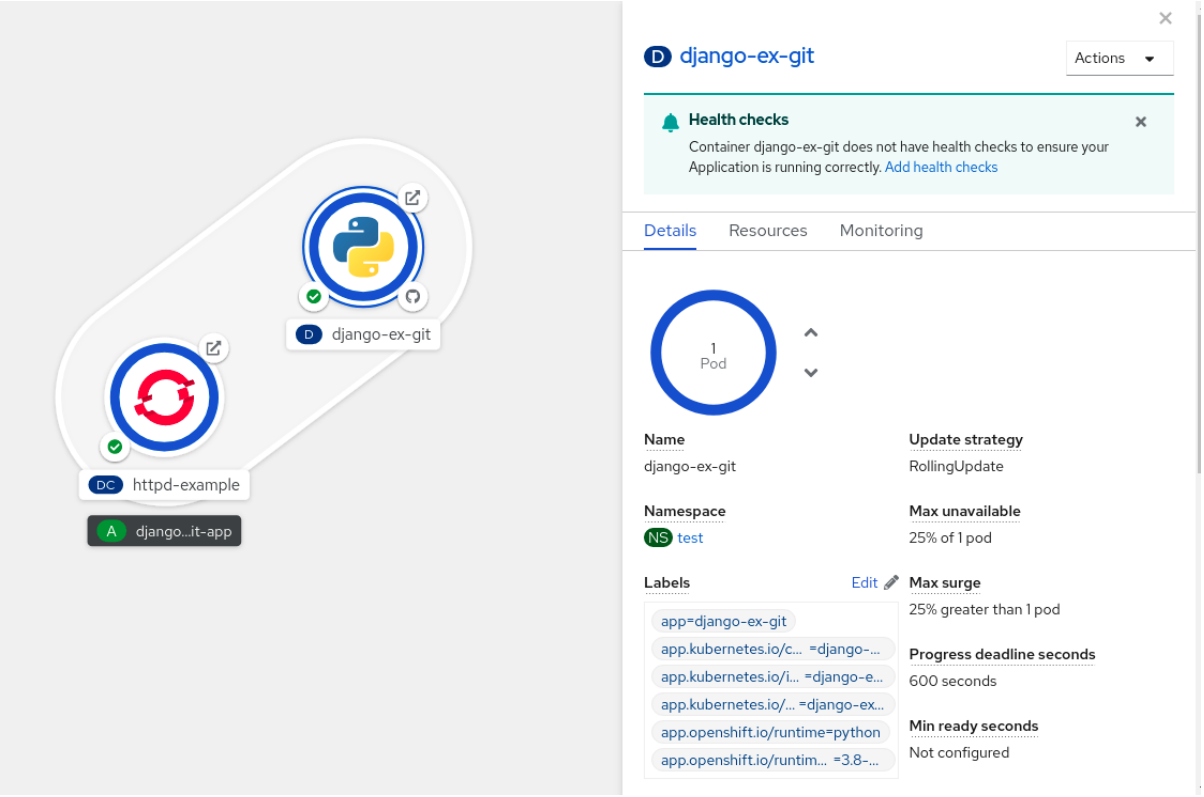
### 前提条件

- **Developer** パースペクティブを使用して OpenShift Container Platform に 2 つ以上のコンポーネントを作成し、デプロイしていること。

### 手順

- サービスを既存のアプリケーショングループに追加するには、**Shift**+ を既存のアプリケーショングループに追加します。コンポーネントをドラッグし、これをアプリケーショングループに追加すると、必要なラベルがコンポーネントに追加されます。

**図4.4 アプリケーションのグループ化**



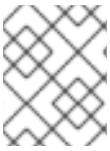

または、以下のようにコンポーネントをアプリケーションに追加することもできます。

1. サービス Pod をクリックし、右側の **Overview** パネルを確認します。
2. **Actions** ドロップダウンメニューをクリックし、**Edit Application Grouping** を選択します。
3. **Edit Application Grouping** ダイアログボックスで、**Application** ドロップダウンリストをクリックし、適切なアプリケーショングループを選択します。
4. **Save** をクリックしてサービスをアプリケーショングループに追加します。

アプリケーショングループからコンポーネントを削除するには、コンポーネントを選択し、**Shift**+ ドラッグでこれをアプリケーショングループからドラッグします。

## 4.7. サービスのアプリケーションへの追加

アプリケーションにサービスを追加するには、トポロジー **Graph view** のコンテキストメニューで**+Add** アクションを使用します。



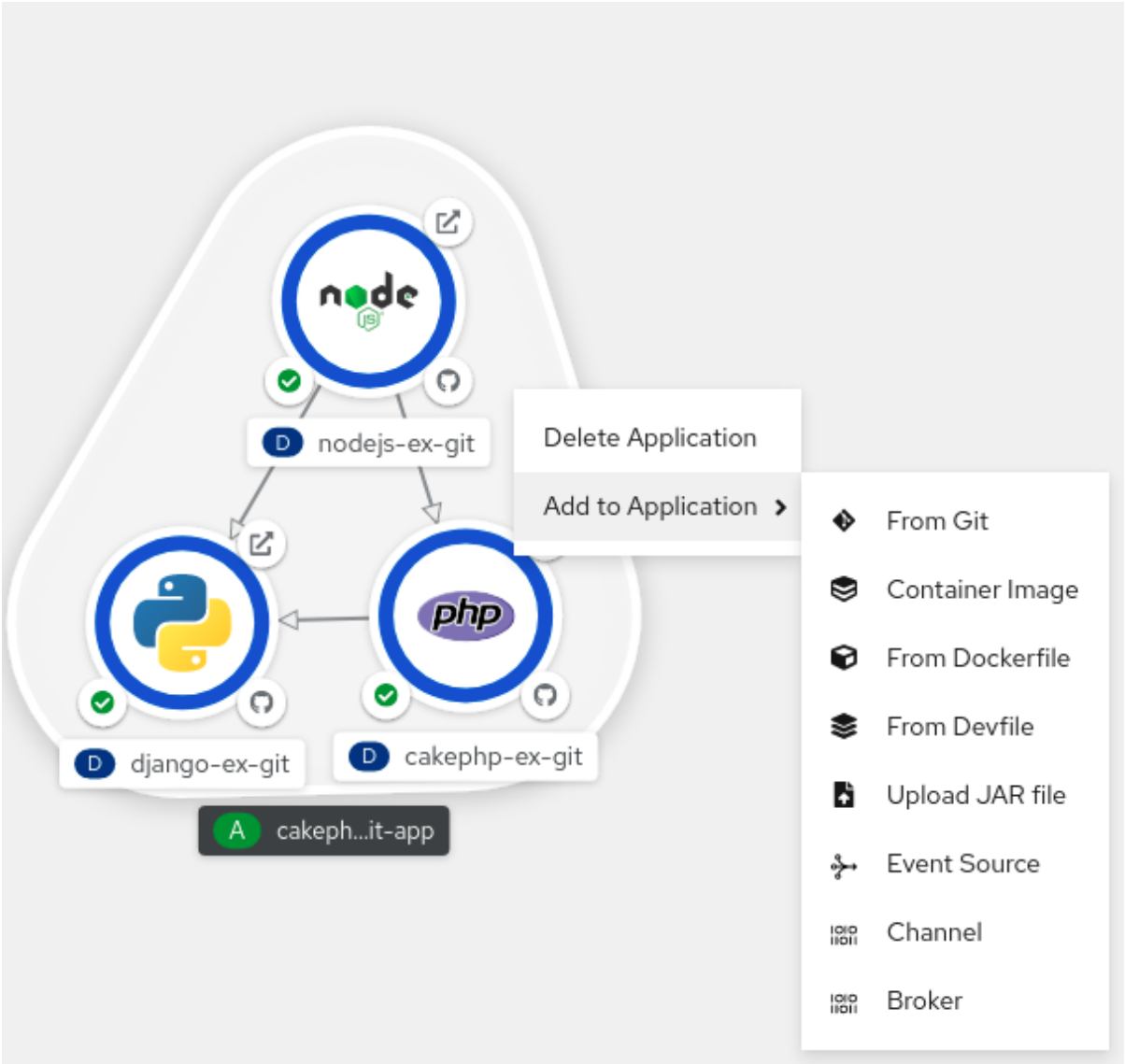
**注記**

コンテキストメニュー以外に、サイドバーを使用するか、アプリケーショングループから矢印の上にマウスをかざしてドラッグしてサービスを追加できます。

**手順**

1. トポロジー **Graph view** でアプリケーショングループを右クリックし、コンテキストメニューを表示します。

**図4.5 リソースコンテキストメニューの追加**




2. **Add to Application** を使用して、**From Git**、**Container Image**、**From Dockerfile**、**From Devfile**、**Upload JAR file**、**Event Source**、**Channel**、または **Broker** など、アプリケーショングループにサービスを追加する手法を選択します。

3. 選択した手法のフォームに入力して、**Create** をクリックします。たとえば、Git リポジトリーのソースコードに基づいてサービスを追加するには、**From Git** の手法を選択し、**Import from Git** フォームに入力して、**Create** をクリックします。

## 4.8. アプリケーションからのサービスの削除

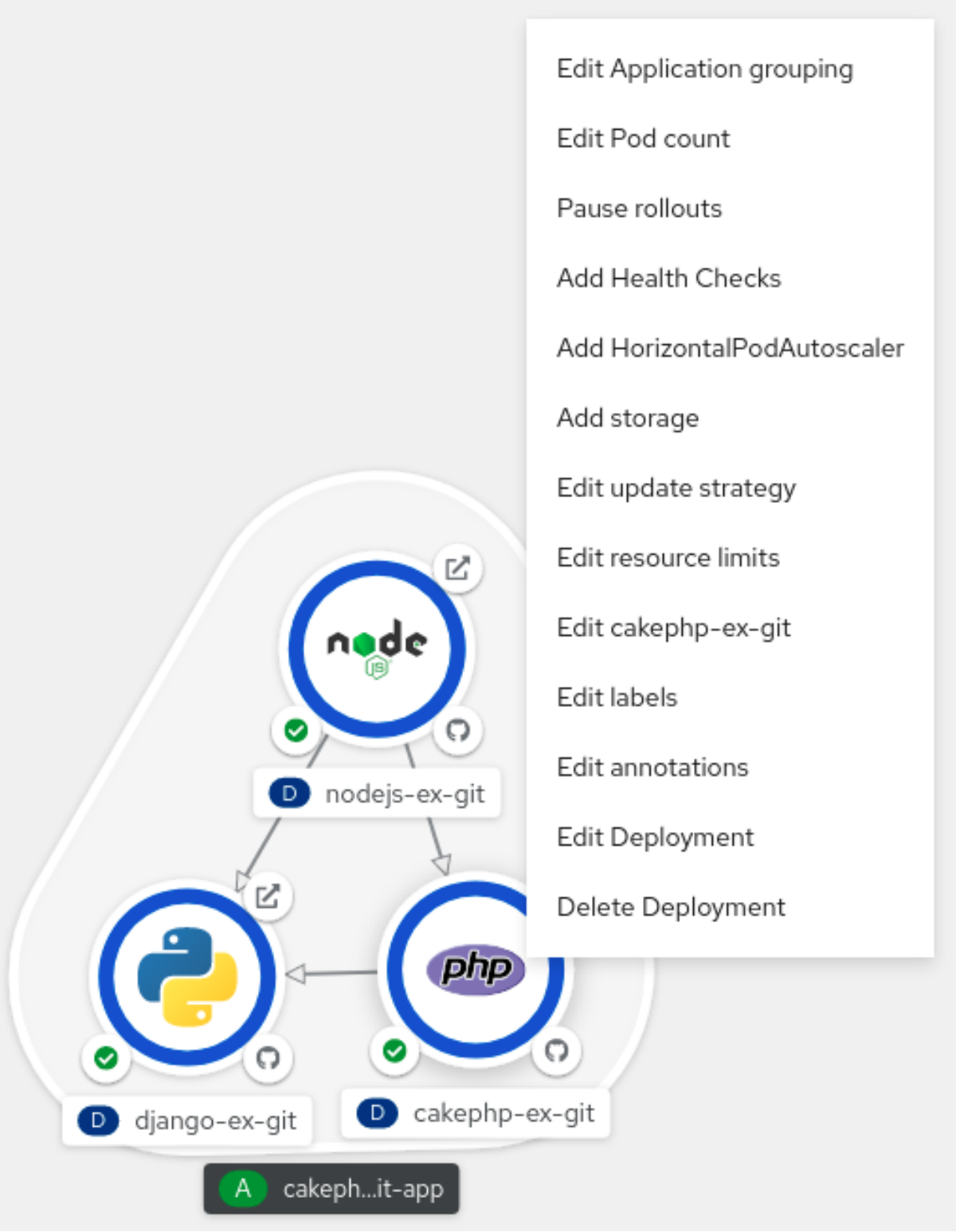
トポロジー **Graph view** のコンテキストメニューでアプリケーションからサービスを削除します。

**手順**

1. トポロジー **Graph view** でアプリケーショングループのサービスを右クリックし、コンテキストメニューを表示します。

2. **Delete Deployment** を選択してサービスを削除します。

図4.6 デプロイメントオプションの削除




## 4.9. TOPOLOGY ビューに使用するラベルとアノテーション

Topology ビューは、以下のラベルおよびアノテーションを使用します。

**ノードに表示されるアイコン**

ノードのアイコンは、最初に **app.openshift.io/runtime** ラベルを使用してから **app.kubernetes.io/name** ラベルを使用して一致するアイコンを検索して定義されます。このマッチングは、事前定義されたアイコンセットを使用して行われます。

**ソースコードエディターまたはソースへのリンク**

**app.openshift.io/vcs-uri** アノテーションは、ソースコードエディターへのリンクを作成するために使用されます。

**ノードコネクター**

**app.openshift.io/connects-to** アノテーションは、ノードに接続するために使用されます。

**アプリケーションのグループ化**

**app.kubernetes.io/part-of=<appname>** ラベルは、アプリケーション、サービス、およびコンポーネントをグループ化するために使用されます。

OpenShift Container Platform アプリケーションで使用する必要のあるラベルとアノテーションの詳細については、Guidelines for labels and annotations for OpenShift applications を参照してください。

## 4.10. 関連情報

- Git からアプリケーションを作成する方法は、Git のコードベースのインポートおよびアプリケーションの作成を参照してください。
- Developer パースペクティブを使用したアプリケーションのサービスへの接続を参照してください。
- アプリケーションのエクスポート を参照してください

# 第5章 アプリケーションのエクスポート

開発者は、アプリケーションを ZIP ファイル形式でエクスポートできます。必要に応じて、**+ Add** ビューの **YAML のインポート** オプションを使用して、エクスポートされたアプリケーションを同じクラスターまたは別のクラスター内の別のプロジェクトにインポートします。アプリケーションをエクスポートすると、アプリケーションリソースを再利用でき、時間を節約できます。

## 5.1. 前提条件

- Operator Hub から gitops-primer Operator をインストールしました。

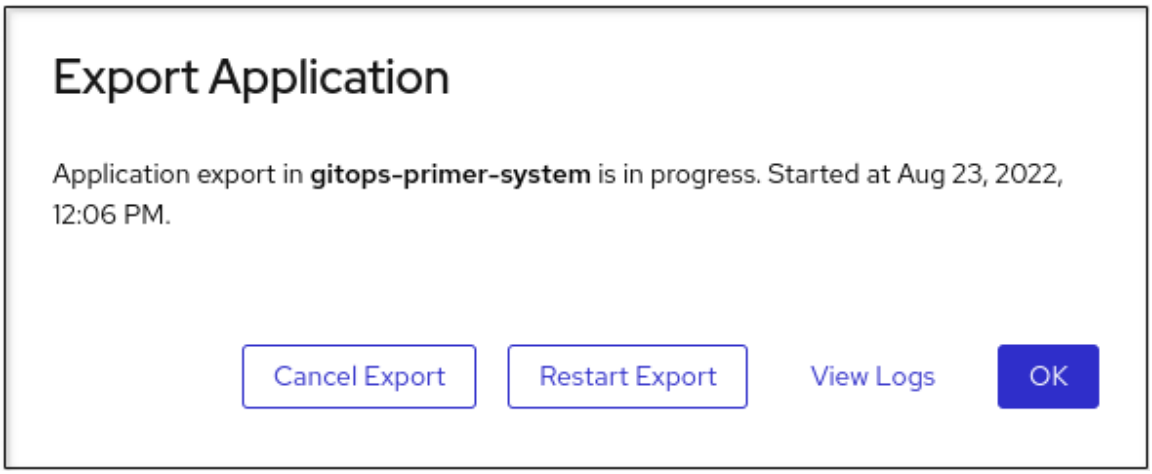
**注記**

gitops-primer Operator をインストールした後でも、**トポロジー** ビューで **アプリケーションのエクスポート** オプションが無効になります。

- **トポロジー** ビューでアプリケーションを作成し、アプリケーションの **エクスポート** を有効にしました。

## 5.2. 手順

1. 開発者パースペクティブで、次のいずれかの手順を実行します。
   a. **+ Add** ビューに移動し、**アプリケーションの移植性** タイルで **アプリケーションのエクスポート** をクリックします。
   b. **トポロジー** ビューに移動し、**アプリケーションのエクスポート** をクリックします。
2. **アプリケーションのエクスポート** ダイアログボックスで **OK** をクリックします。プロジェクトからのリソースのエクスポートが開始されたことを確認する通知が開きます。
3. 次のシナリオで実行する必要があるオプションの手順:
   - 不適切なアプリケーションのエクスポートを開始した場合は、**アプリケーションのエクスポート → エクスポート** の **キャンセル** をクリックします。
   - エクスポートがすでに進行中で、新たにエクスポートを開始する場合は、**アプリケーションのエクスポート → エクスポート** の **再開** をクリックします。
   - アプリケーションのエクスポートに関連するログを表示するには、**アプリケーションのエクスポート** をクリックし、**ログの表示** リンクをクリックします。



4. エクスポートが正常に完了したら、ダイアログボックスで **ダウンロード** をクリックして、アプリケーションリソースを ZIP 形式でマシンにダウンロードします。

# 第6章 アプリケーションのサービスへの接続

## 6.1. SERVICE BINDING OPERATOR のリリースノート

サービスバインディング Operator は、サービスバインディングのコントローラーおよび付随のカスタムリソース定義 (CRD) で設定されます。サービスバインディング Operator は、ワークロードおよびバッキングサービスのデータプレーンを管理します。サービスバインディングコントローラーは、バッキングサービスのコントロールプレーン提供のデータを読み取ります。次に、**ServiceBinding** リソースで指定されるルールに従って、このデータをワークロードに追加します。

サービスバインディング Operator を使用すると、以下を行うことができます。

- ワークロードを Operator 管理のバッキングサービスと共にバインドします。
- バインディングデータの設定を自動化します。
- サービスオペレーターは簡単にサービスへのアクセスのプロビジョニングや管理が行えます。
- クラスター環境の不一致をなくす一貫性がある宣言型サービスバインディングメソッドを使用し、開発ライフサイクルを充実させます。

サービスバインディング Operator のカスタムリソース定義 (CRD) は以下の API をサポートします。

- Service Binding: **binding.operators.coreos.com** API グループ
- **servicebinding.io** API グループを使用した **サービスバインディング (仕様 API)**。

### 6.1.1. サポート表

次の表の一部の機能は、テクノロジープレビュー 段階にあります。これらの実験的機能は、実稼働環境での使用を目的としていません。

以下の表では、機能は以下のステータスでマークされています。

- TP: **テクノロジープレビュー機能**
- GA: **一般公開機能**

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

**表6.1 サポート表**

| サービスバインディング Operator | API グループとサポート状況 | | OpenShift Versions |
|---|---|---|---|
| **バージョン** | **binding.operators.coreos.com** | **servicebinding.io** | |
| 1.3.3 | GA | GA | 4.9-4.12 |
| 1.3.1 | GA | GA | 4.9-4.11 |
| 1.3 | GA | GA | 4.9-4.11 |

| サービスバインディング Operator | API グループとサポート状況 | | OpenShift Versions |
|---|---|---|---|
| 1.2 | GA | GA | 4.7-4.11 |
| 1.1.1 | GA | TP | 4.7-4.10 |
| 1.1 | GA | TP | 4.7-4.10 |
| 1.0.1 | GA | TP | 4.7-4.9 |
| 1.0 | GA | TP | 4.7-4.9 |

### 6.1.2. 多様性を受け入れるオープンソースの強化

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

### 6.1.3. Service Binding Operator 1.3.3 のリリースノート

Service Binding Operator 1.3.3 は、OpenShift Container Platform 4.9、4.10、4.11、および 4.12 で利用できるようになりました。

#### 6.1.3.1. 修正された問題

- この更新の前に、Service Binding Operator のセキュリティー脆弱性 **CVE-2022-41717** が指摘されていました。この更新により、**CVE-2022-41717** エラーが修正され、**golang.org/x/net** パッケージが v0.0.0-20220906165146-f3363e06e74c から v0.4.0 に更新されます。APPSVC-1256

- この更新の前は、プロビジョニングされたサービスは、それぞれのリソースに servicebinding.io/provisioned-service: true アノテーションが設定されている場合にのみ検出され、他のプロビジョニングされたサービスは検出されませんでした。この更新により、検出メカニズムは status.binding.name 属性に基づいて、すべてのプロビジョニングされたサービスを正しく識別します。APPSVC-1204

### 6.1.4. Service Binding Operator 1.3.1 のリリースノート

Service Binding Operator 1.3.1 が OpenShift Container Platform 4.9、4.10、および 4.11 で利用できるようになりました。

#### 6.1.4.1. 修正された問題

- この更新以前に、Service Binding Operator におけるセキュリティーの脆弱性 **CVE-2022-32149** が指摘されていました。この更新により、**CVE-2022-32149** のエラーが修正され、**golang.org/x/text** パッケージが v0.3.7 から v0.3.8 に更新されます。 APPSVC-1220

### 6.1.5. Service Binding Operator 1.3 のリリースノート

Service Binding Operator 1.3 が OpenShift Container Platform 4.9、4.10、および 4.11 で利用できるようになりました。

#### 6.1.5.1. 削除された機能

- Service Binding Operator 1.3 では、リソースの使用率を向上させるために Operator Lifecycle Manager (OLM) 記述子機能が削除されました。OLM 記述子の代わりに、CRD アノテーションを使用してバインディングデータを宣言できます。

### 6.1.6. Service Binding Operator 1.2 のリリースノート

Service Binding Operator 1.2 が OpenShift Container Platform 4.7、4.8、4.9、4.10、および 4.11 で利用可能になりました。

#### 6.1.6.1. 新機能

このセクションでは、Service Binding Operator 1.2 の主な新機能について説明します。

- **optional** のフラグ値を **true** に設定して、Service Binding Operator がアノテーションのオプションフィールドを考慮できるようにします。
- **servicebinding.io/v1beta1** リソースのサポート。
- ワークロードの存在を必要とせずに関連するバインディングシークレットを公開することにより、バインド可能なサービスの検出可能性が向上します。

#### 6.1.6.2. 既知の問題

- 現在、OpenShift Container Platform 4.11 に Service Binding Operator をインストールすると、Service Binding Operator のメモリーフットプリントが予想される制限を超えて増加します。ただし、使用率が低い場合、メモリーフットプリントは環境またはシナリオの予想範囲内にとどまります。OpenShift Container Platform 4.10 と比較すると、負荷がかかると、平均および最大メモリーフットプリントの両方が大幅に増加します。この問題は、Service Binding Operator の以前のバージョンでも明らかです。現在、この問題に対する回避策はありません。APPSVC-1200
- デフォルトでは、展開されたファイルのアクセス許可は 0644 に設定されています。Service Binding Operator は、サービスが **0600** などの特定の権限を想定している場合に問題を引き起こす Kubernetes のバグにより、特定の権限を設定できません。回避策として、ワークロードリソース内で実行されているプログラムまたはアプリケーションのコードを変更して、ファイルを **/tmp** ディレクトリーにコピーし、適切な権限を設定することができます。APPSVC-1127
- 現時点で、Service Binding Operator を 1 つの namespace インストールモードでインストールする場合に発生する基地の問題があります。適切な namespace スコープのロールベースアクセス制御 (RBAC) ルールがないため、サービスバインディング Operator が自動的に検出およびバインドできる既知の Operator がサポートするいくつかのサービスへのアプリケーションのバインドが正常に行われません。これが発生すると、次の例のようなエラーメッセージが生成されます。

  **エラーメッセージの例**

  ```
  `postgresclusters.postgres-operator.crunchydata.com "hippo" is forbidden:
      User "system:serviceaccount:my-petclinic:service-binding-operator" cannot
      get resource "postgresclusters" in API group "postgres-operator.crunchydata.com"
      in the namespace "my-petclinic"`
  ```

回避策 1: **all namespaces** インストールモードでサービスバインディング Operator をインストールします。その結果、適切なクラスタースコープの RBAC ルールが存在し、バインディングが正常に実行されるようになります。

回避策 2: サービスバインディング Operator を **all namespaces** インストールモードでインストールできない場合は、サービスバインディング Operator がインストールされている namespace に以下のロールバインディングをインストールします。

**例:Crunchy Postgres Operator のロールバインディング**

```
kind: RoleBinding
apiVersion: rbac.authorization.k8s.io/v1
metadata:
  name: service-binding-crunchy-postgres-viewer
subjects:
  - kind: ServiceAccount
    name: service-binding-operator
roleRef:
  apiGroup: rbac.authorization.k8s.io
  kind: ClusterRole
  name: service-binding-crunchy-postgres-viewer-role
```

APPSVC-1062

- 仕様によると、**ClusterWorkloadResourceMapping** リソースを変更する場合、Service Binding Operator は以前のバージョンの **ClusterWorkloadResourceMapping** リソースを使用して、今まで反映されていたバインディングデータを削除する必要があります。現在、**ClusterWorkloadResourceMapping** リソースを変更すると、Service Binding Operator は **ClusterWorkloadResourceMapping** リソースの最新バージョンを使用してバインディングデータを削除します。その結果、{the servicebinding-title} はバインディングデータを誤って削除する可能性があります。回避策として、以下の手順を実施してください。

  1. 対応する **ClusterWorkloadResourceMapping** リソースを使用する **ServiceBinding** リソースをすべて削除します。
  2. **ClusterWorkloadResourceMapping** リソースを変更します。
  3. 手順 1 で削除した **ServiceBinding** リソースを再適用します。

  APPSVC-1102

### 6.1.7. Service Binding Operator 1.1.1 のリリースノート

Service Binding Operator 1.1.1 が OpenShift Container Platform 4.7、4.8、4.9、4.10 で利用可能になりました。

#### 6.1.7.1. 修正された問題

- この更新以前に、Service Binding Operator Helm チャートにおけるセキュリティーの脆弱性 **CVE-2021-38561** が指摘されていました。この更新により、**CVE-2021-38561** のエラーが修正され、**golang.org/x/text** パッケージが v0.3.6 から v0.3.7 に更新されます。 APPSVC-1124
- この更新以前は、Developer Sandbox のユーザーには、**ClusterWorkloadResourceMapping** リソースを読み取るための十分なパーミッションがありませんでした。その結果、Service Binding Operator はすべてのサービスバインディングの成功を妨げていました。今回の更新により、Service Binding Operator には、Developer Sandbox ユーザーを含め、認証されたサブ

ジェクトの適切なロールベースのアクセス制御 (RBAC) ルールが含まれるようになりました。これらの RBAC ルールにより、Service Binding Operator は Developer Sandbox ユーザーの **ClusterWorkloadResourceMapping** リソースを **get**、**list**、および **watch** して、サービスバインディングを正常に処理できます。APPSVC-1135

#### 6.1.7.2. 既知の問題

- 現時点で、Service Binding Operator を 1 つの namespace インストールモードでインストールする場合に発生する基地の問題があります。適切な namespace スコープのロールベースアクセス制御 (RBAC) ルールがないため、サービスバインディング Operator が自動的に検出およびバインドできる既知の Operator がサポートするいくつかのサービスへのアプリケーションのバインドが正常に行われません。これが発生すると、次の例のようなエラーメッセージが生成されます。

  **エラーメッセージの例**

  ```
  `postgresclusters.postgres-operator.crunchydata.com "hippo" is forbidden:
      User "system:serviceaccount:my-petclinic:service-binding-operator" cannot
      get resource "postgresclusters" in API group "postgres-operator.crunchydata.com"
      in the namespace "my-petclinic"`
  ```

  回避策 1: **all namespaces** インストールモードでサービスバインディング Operator をインストールします。その結果、適切なクラスタースコープの RBAC ルールが存在し、バインディングが正常に実行されるようになります。

  回避策 2: サービスバインディング Operator を **all namespaces** インストールモードでインストールできない場合は、サービスバインディング Operator がインストールされている namespace に以下のロールバインディングをインストールします。

  **例:Crunchy Postgres Operator のロールバインディング**

  ```
  kind: RoleBinding
  apiVersion: rbac.authorization.k8s.io/v1
  metadata:
    name: service-binding-crunchy-postgres-viewer
  subjects:
    - kind: ServiceAccount
      name: service-binding-operator
  roleRef:
    apiGroup: rbac.authorization.k8s.io
    kind: ClusterRole
    name: service-binding-crunchy-postgres-viewer-role
  ```

  APPSVC-1062

- 現在、**ClusterWorkloadResourceMapping** リソースを変更すると、Service Binding Operator は正しい動作を実装しません。回避策として、以下の手順を実施してください。

  1. 対応する **ClusterWorkloadResourceMapping** リソースを使用する **ServiceBinding** リソースをすべて削除します。
  2. **ClusterWorkloadResourceMapping** リソースを変更します。
  3. 手順 1 で削除した **ServiceBinding** リソースを再適用します。

APPSVC-1102

### 6.1.8. Service Binding Operator 1.1 のリリースノート

サービスバインディング Operator が OpenShift Container Platform 4.7、4.8、4.9、4.10 で利用可能になりました。

#### 6.1.8.1. 新機能

このセクションでは、Service Binding Operator 1.1 の主な新機能について説明します。

- サービスバインディングオプション
  - ワークロードリソースマッピング: セカンダリーワークロードに対してバインディングデータを投影する必要がある場所を正確に定義します。
  - ラベルセレクターを使用して新しいワークロードをバインドします。

#### 6.1.8.2. 修正された問題

- この更新以前は、ラベルセレクターを使用してワークロードを取得していたサービスバインディングは、指定されたラベルセレクターに一致する新しいワークロードにサービスバインディングデータを投影しませんでした。その結果、Service Binding Operator はそのような新しいワークロードを定期的にバインドできませんでした。今回の更新により、サービスバインディングは、指定されたラベルセレクターに一致する新しいワークロードにサービスバインディングデータを投影するようになりました。Service Binding Operator は、定期的に新しいワークロードを見つけてバインドを試みるようになりました。APPSVC-1083

#### 6.1.8.3. 既知の問題

- 現時点で、Service Binding Operator を 1 つの namespace インストールモードでインストールする場合に発生する基地の問題があります。適切な namespace スコープのロールベースアクセス制御 (RBAC) ルールがないため、サービスバインディング Operator が自動的に検出およびバインドできる既知の Operator がサポートするいくつかのサービスへのアプリケーションのバインドが正常に行われません。これが発生すると、次の例のようなエラーメッセージが生成されます。

  **エラーメッセージの例**

  ```
  `postgresclusters.postgres-operator.crunchydata.com "hippo" is forbidden:
      User "system:serviceaccount:my-petclinic:service-binding-operator" cannot
      get resource "postgresclusters" in API group "postgres-operator.crunchydata.com"
      in the namespace "my-petclinic"`
  ```

  回避策 1: **all namespaces** インストールモードでサービスバインディング Operator をインストールします。その結果、適切なクラスタースコープの RBAC ルールが存在し、バインディングが正常に実行されるようになります。

  回避策 2: サービスバインディング Operator を **all namespaces** インストールモードでインストールできない場合は、サービスバインディング Operator がインストールされている namespace に以下のロールバインディングをインストールします。

  **例:Crunchy Postgres Operator のロールバインディング**

```
kind: RoleBinding
apiVersion: rbac.authorization.k8s.io/v1
metadata:
  name: service-binding-crunchy-postgres-viewer
subjects:
  - kind: ServiceAccount
    name: service-binding-operator
roleRef:
  apiGroup: rbac.authorization.k8s.io
  kind: ClusterRole
  name: service-binding-crunchy-postgres-viewer-role
```

APPSVC-1062

- 現在、**ClusterWorkloadResourceMapping** リソースを変更すると、Service Binding Operator は正しい動作を実装しません。回避策として、以下の手順を実施してください。

  1. 対応する **ClusterWorkloadResourceMapping** リソースを使用する **ServiceBinding** リソースをすべて削除します。
  2. **ClusterWorkloadResourceMapping** リソースを変更します。
  3. 手順 1 で削除した **ServiceBinding** リソースを再適用します。

APPSVC-1102

### 6.1.9. Service Binding Operator 1.0.1 のリリースノート

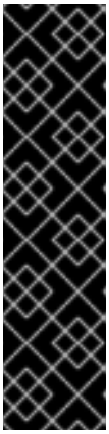
サービスバインディング Operator が OpenShift Container Platform 4.7、4.8 および 4.9 で利用可能になりました。

サービスバインディング Operator 1.0.1 は、以下で実行されている OpenShift Container Platform 4.9 以降をサポートします。

- IBM Power Systems
- IBM Z および LinuxONE

サービスバインディング Operator 1.0.1 のカスタムリソース定義 (CRD) は以下の API をサポートします。

- Service Binding: **binding.operators.coreos.com** API グループ
- Service Binding (Spec API テクノロジープレビュー) **servicebinding.io** API グループ



**重要**

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

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

#### 6.1.9.1. サポート表

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

テクノロジープレビュー機能のサポート範囲

以下の表では、機能は以下のステータスでマークされています。

- **TP: テクノロジープレビュー機能**
- **GA: 一般公開機能**

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

**表6.2 サポート表**

| 機能 | サービスバインディング Operator 1.0.1 |
|---|---|
| **binding.operators.coreos.com** API グループ | GA |
| **ServiceBinding.io** API グループ | TP |

#### 6.1.9.2. 修正された問題

- 今回の更新以前は、**postgresql.k8s.enterpriesedb.io/v1** API の **Cluster** カスタムリソース (CR) からデータ値をバインドすると、CR の **.metadata.name** フィールドから **host** バインディング値が収集されていました。収集されたバインディング値は間違ったホスト名であり、正しいホスト名は **.status.writeService** フィールドで確認できます。今回の更新により、サービスバインディング Operator がバッキングサービス CR からバインディングデータ値を公開するために使用するアノテーションが変更され、**.status.writeService** フィールドから **host** バインディング値を収集するようになりました。サービスバインディング Operator はこれらの変更されたアノテーションを使用して、**host** および **provider** のバインディングに正しいホスト名を反映します。APPSVC-1040

- 今回の更新以前は、**postgres-operator.crunchydata.com/v1beta1** API の **PostgresCluster** CR をバインドする際に、バインディングデータ値にデータベース証明書の値が含まれませんでした。その結果、アプリケーションはデータベースへの接続に失敗しました。今回の更新により、サービスバインディング Operator がバッキングサービス CR からバインディングデータを公開するために使用するアノテーションへの変更に、データベース証明書が含まれるようになりました。サービスバインディング Operator はこれらの変更されたアノテーションを使用して、正しい **ca.crt**、**tls.crt**、および **tls.key** 証明書ファイルを反映します。APPSVC-1045

- 今回の更新以前は、**pxc.percona.com** API の **PerconaXtraDBCluster** カスタムリソース (CR) をバインドする場合、バインディングデータ値に **port** および **database** の値が含まれませんでした。アプリケーションがデータベースサービスに正常に接続するには、これらのバインディング値とすでに反映されている他の値が必要です。今回の更新により、サービスバインディング Operator がバッキングサービス CR からバインディングデータ値を公開するために使用するアノテーションが変更され、追加の **por** および **database** バインディング値を反映するようになりました。サービスバインディング Operator はこれらの変更されたアノテーションを使用して、アプリケーションがデータベースサービスに正常に接続するために使用できるバインディング値の完全なセットを反映します。APPSVC-1073

#### 6.1.9.3. 既知の問題

- 現時点で、単一の namespace インストールモードでサービスバインディング Operator をインストールする際に、適切な namespace スコープのロールベースアクセス制御 (RBAC) ルールがないため、サービスバインディング Operator が自動的に検出およびバインドできる既知の Operator がサポートするいくつかのサービスへのアプリケーションのバインドが正常に行われません。さらに、以下のエラーメッセージが生成されます。

  **エラーメッセージの例**

  ```
  `postgresclusters.postgres-operator.crunchydata.com "hippo" is forbidden:
      User "system:serviceaccount:my-petclinic:service-binding-operator" cannot
      get resource "postgresclusters" in API group "postgres-operator.crunchydata.com"
      in the namespace "my-petclinic"`
  ```

  回避策 1: **all namespaces** インストールモードでサービスバインディング Operator をインストールします。その結果、適切なクラスタースコープの RBAC ルールが存在し、バインディングが正常に実行されるようになります。

  回避策 2: サービスバインディング Operator を **all namespaces** インストールモードでインストールできない場合は、サービスバインディング Operator がインストールされている namespace に以下のロールバインディングをインストールします。

  **例:Crunchy Postgres Operator のロールバインディング**

  ```
  kind: RoleBinding
  apiVersion: rbac.authorization.k8s.io/v1
  metadata:
    name: service-binding-crunchy-postgres-viewer
  subjects:
    - kind: ServiceAccount
      name: service-binding-operator
  roleRef:
    apiGroup: rbac.authorization.k8s.io
    kind: ClusterRole
    name: service-binding-crunchy-postgres-viewer-role
  ```

  APPSVC-1062

### 6.1.10. サービスバインディング Operator 1.0 のリリースノート

サービスバインディング Operator が OpenShift Container Platform 4.7、4.8 および 4.9 で利用可能になりました。

サービスバインディング Operator 1.0 のカスタムリソース定義 (CRD) は以下の API をサポートします。

- Service Binding: **binding.operators.coreos.com** API グループ
- Service Binding (Spec API テクノロジープレビュー) **servicebinding.io** API グループ

**重要**

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

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

#### 6.1.10.1. サポート表

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

テクノロジープレビュー機能のサポート範囲

以下の表では、機能は以下のステータスでマークされています。

- **TP**: **テクノロジープレビュー機能**
- **GA**: **一般公開機能**

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

**表6.3 サポート表**

| 機能 | サービスバインディング Operator 1.0 |
|---|---|
| **binding.operators.coreos.com** API グループ | GA |
| **ServiceBinding.io** API グループ | TP |

#### 6.1.10.2. 新機能

サービスバインディング Operator 1.0 は、以下で実行されている OpenShift Container Platform 4.9 以降をサポートします。

- IBM Power Systems
- IBM Z および LinuxONE

このセクションでは、サービスバインディング Operator 1.0 の主な新機能について説明します。

- サービスからのバインディングデータの公開

    - CRD、カスタムリソース (CR)、またはリソースに存在するアノテーションをベースにする。
    - Operator Lifecycle Manager(OLM) 記述子にある記述子をベースにする。
    - プロビジョニングされたサービスのサポート
- ワークロードのプロジェクション
    - ボリュームマウントを使用してバインディングデータをファイルとしてプロジェクションする。
    - バインディングデータを環境変数としてプロジェクションする。
- サービスバインディングオプション
    - ワークロード namespace とは異なる namespace でバッキングサービスをバインドする。
    - バインディングデータを特定のコンテナーワークロードにプロジェクションする。
    - バッキングサービス CR が所有するリソースからバインディングデータを自動的に検出する。
    - 公開されるバインディングデータからカスタムバインディングデータを作成する。
    - **PodSpec** 以外のワークロードリソースをサポートする。
- セキュリティー
    - ロールベースアクセス制御 (RBAC) をサポートする。

### 6.1.11. 関連情報

- サービスバインディング Operator

## 6.2. サービスバインディング OPERATOR

アプリケーション開発者は、ワークロードをビルドして接続するバッキングサービスへのアクセスが必要です。ワークロードをバッキングサービスに接続するのは、提案するシークレットにアクセスしてワークロードで消費する方法がサービスプロバイダーごとに異なるので、困難です。さらにワークロードのバインドおよびサービスのバッキングを手動で設定して保守する場合には、プロセスが煩雑で効率が悪く、エラーが発生しやすくなります。

サービスバインディング Operator を使用すると、アプリケーション開発者は、手作業でバインディング接続を設定する手順なしに、オペレーターが管理するバッキングサービスとワークロードを簡単にバインドできます。

### 6.2.1. サービスバインディングの用語

このセクションでは、サービスバインディングで使用される基本用語の概要を説明します。

| | |
|---|---|
| サービスバインディング | サービスに関する情報をワークロードに提供するアクションの表現。たとえば、Java アプリケーションと必要なデータベース間で認証情報の交換を確立することなどです。 |

| | |
|---|---|
| バッキングサービス | アプリケーションが通常の操作の一部としてネットワーク経由で使用するサービスまたはソフトウェア。たとえば、データベース、メッセージ、REST エンドポイント、イベントストリーム、アプリケーション、アプリケーションパフォーマンスモニター (APM)、またはハードウェアセキュリティーモジュール (HSM) が含まれます。 |
| ワークロード (アプリケーション) | コンテナー内で実行されているプロセス。たとえば、Sprsh Boot アプリケーション、NodeJS Express アプリケーション、Ruby on Rails アプリケーションなどが含まれます。 |
| バインディングデータ | クラスター内で他のリソースの動作を設定するのに使用するサービスに関する情報。たとえば、認証情報、接続の詳細、ボリュームマウント、またはシークレットが含まれます。 |
| バインディング接続 | バインド可能なバッキングサービスとそのバッキングサービスを必要とするアプリケーションなど、接続されたコンポーネント間の相互作用を確立する接続。 |

### 6.2.2. サービスバインディング Operator

サービスバインディング Operator は、サービスバインディングのコントローラーおよび付随のカスタムリソース定義 (CRD) で設定されます。サービスバインディング Operator は、ワークロードおよびバッキングサービスのデータプレーンを管理します。サービスバインディングコントローラーは、バッキングサービスのコントロールプレーン提供のデータを読み取ります。次に、**ServiceBinding** リソースで指定されるルールに従って、このデータをワークロードに追加します。

これにより、サービスバインディング Operator は、ワークロードとのバインディングデータを自動的に収集して共有することで、サービスはバッキングサービスまたは外部サービスを使用できます。このプロセスには、バッキングサービスをバインド可能にして、ワークロードとサービスをバインドすることが含まれます。

#### 6.2.2.1. Operator の管理するサービスをバインド可能にする

サービスをバインド可能にするには、Operator プロバイダーは、ワークロードに必要なバインドデータを公開して Operator が提供するサービスとバインドする必要があります。バインディングデータは、バッキングサービスを管理する Operator の CRD で、アノテーションか、記述子として指定できます。

#### 6.2.2.2. ワークロードをバッキングサービスとバインドする

サービスバインディング Operator を使用して、アプリケーション開発者はバインディング接続を確立する意思を宣言する必要があります。バッキングサービスを参照する **ServiceBinding** CR を作成する必要があります。このアクションにより、サービスバインディング Operator がトリガーされ、公開されたバインディングデータがワークロードにプロジェクションされます。サービスバインディング Operator は、宣言された意図を受けとり、バッキングサービスとワークロードをバインドします。

サービスバインディング Operator の CRD は以下の API をサポートします。

- Service Binding: **binding.operators.coreos.com** API グループ
- **servicebinding.io** API グループを使用した **サービスバインディング (仕様 API)**。

サービスバインディング Operator を使用すると、以下を行うことができます。

- ワークロードを Operator 管理のバッキングサービスとバインドします。
- バインディングデータの設定を自動化します。
- サービスへのアクセスをプロビジョニングおよび管理するためのロータッチな管理エクスペリエンスをサービス Operator に提供します。
- クラスター環境の不一致をなくす一貫性がある宣言型サービスバインディングメソッドを使用し、開発ライフサイクルを充実させます。

### 6.2.3. 主な特長

- サービスからのバインディングデータの公開
  - CRD、カスタムリソース (CR)、またはリソースに存在するアノテーションをベースにする。
- ワークロードのプロジェクション
  - ボリュームマウントを使用してバインディングデータをファイルとしてプロジェクションする。
  - バインディングデータを環境変数としてプロジェクションする。
- サービスバインディングオプション
  - ワークロード namespace とは異なる namespace でバッキングサービスをバインドする。
  - バインディングデータを特定のコンテナーワークロードにプロジェクションする。
  - バッキングサービス CR が所有するリソースからバインディングデータを自動的に検出する。
  - 公開されるバインディングデータからカスタムバインディングデータを作成する。
  - **PodSpec** 以外のワークロードリソースをサポートする。
- セキュリティー
  - ロールベースアクセス制御 (RBAC) をサポートする。

### 6.2.4. API の違い

サービスバインディング Operator の CRD は以下の API をサポートします。

- Service Binding: **binding.operators.coreos.com** API グループ
- **servicebinding.io** API グループを使用した **サービスバインディング (仕様 API)**。

これらの API グループは両方とも類似した機能を持っていますが、完全に同一ではありません。これらの API グループ間の相違点の完全なリストを次に示します。

| 機能 | **binding.operators.coreos.com** API グループによるサポート | **servicebinding.io** API グループによるサポート | 注意 |
| --- | --- | --- | --- |
| プロビジョニングされたサービスへのバインド | はい | はい | 該当なし (該当なし) |
| ダイレクトシークレットプロジェクション | はい | はい | 該当なし (該当なし) |
| ファイルとしてバインド | はい | はい | • **servicebinding.io** API グループのサービスバインディングのデフォルトの動作<br>• **binding.operators.coreos.com** API グループのサービスバインディングのオプトイン機能 |
| 環境変数としてバインド | はい | はい | • **binding.operators.coreos.com** API グループのサービスバインディングのデフォルトの動作。<br>• **servicebinding.io** API グループのサービスバインディングのオプトイン機能: 環境変数はファイルと共に作成されます。 |
| ラベルセレクターを使用したワークロードの選択 | はい | はい | 該当なし (該当なし) |
| Detecting binding resources (**.spec.detectBindingResources**) | はい | いいえ | **servicebinding.io** API グループには、同等の機能はありません。 |

| 機能 | **binding.operators.coreos.com** API グループによるサポート | **servicebinding.io** API グループによるサポート | 注意 |
| --- | --- | --- | --- |
| 命名ストラテジー | はい | いいえ | **servicebinding.io** API グループ内には、ネーミング戦略が使用するテンプレートを解釈する現在のメカニズムはありません。 |
| コンテナーパス | はい | 部分使用 | **binding.operators.coreos.com** API グループのサービスバインディングは、**ServiceBinding** リソース内のマッピング動作を指定できるため、**servicebinding.io** API グループは、ワークロードに関する詳細情報がないと、同等の動作を完全にサポートできません。 |
| コンテナー名のフィルタリング | いいえ | はい | **binding.operators.coreos.com** API グループには、同等の機能はありません。 |
| Secret path | はい | いいえ | **servicebinding.io** API グループには、同等の機能はありません。 |

| 機能 | binding.operators.coreos.com API グループによるサポート | servicebinding.io API グループによるサポート | 注意 |
| --- | --- | --- | --- |
| 代替バインディングソース (たとえば、アノテーションからのバインディングデータ) | はい | サービスバインディング Operator によって許可される | この仕様では、プロビジョニングされたサービスとシークレットからバインディングデータを取得するためのサポートが必要です。ただし、仕様を厳密に読むと、他のバインディングデータソースのサポートが許可されていることが示唆されます。この事実を利用して、Service Binding Operator はさまざまなソースからバインディングデータを取得できます (たとえば、アノテーションからバインディングデータを取得するなど)。Service Binding Operator は、両方の API グループでこれらのソースをサポートします。 |

### 6.2.5. 関連情報

- サービスバインディングの使用

## 6.3. サービスバインディング OPERATOR のインストール

以下では、クラスター管理者を対象に、サービスバインディング Operator を OpenShift Container Platform クラスターにインストールするプロセスについて説明します。

OpenShift Container Platform 4.7 以降では サービスバインディング Operator をインストールできます。

**前提条件**

- **cluster-admin** パーミッションを持つアカウントを使用して OpenShift Container Platform クラスターにアクセスできる。
- クラスターで Marketplace 機能 が有効になっているか、Red Hat Operator カタログソースが手動で設定されている。

### 6.3.1. Web コンソールを使用したサービスバインディング Operator のインストール

OpenShift Container Platform OperatorHub を使用してサービスバインディング Operator をインストールできます。サービスバインディング Operator をインストールする時に、サービスバインディングの設定に必要なカスタムリソース (CR) は Operator と共に自動的にインストールされます。

**手順**

1. Web コンソールの **Administrator** パースペクティブで、**Operators → OperatorHub** に移動します。
2. **Filter by keyword**ボックスを使用して、カタログで **Service Binding Operator** を検索します。**Service Binding Operator** タイルをクリックします。
3. **Service Binding Operator** ページで Operator についての簡単な説明を参照してください。**Install** をクリックします。
4. **Install Operator** ページで以下を行います。
   a. **Installation Mode** で **All namespaces on the cluster (default)**を選択します。このモードは、デフォルトの **openshift-operators** namespace に Operator をインストールします。これにより、Operator はクラスター内のすべての namespace を監視し、これらの namespace に対して利用可能になります。
   b. **Approval Strategy** で **Automatic** を選択します。これにより、Operator への今後のアップグレードは Operator Lifecycle Manager (OLM) によって自動的に処理されます。**Manual** 承認ストラテジーを選択すると、OLM は更新要求を作成します。クラスター管理者は、Operator を新規バージョンに更新できるように OLM 更新要求を手動で承認する必要があります。
   c. **Update Channel** を選択します。
      - デフォルトでは、**stable** チャネルでは、サービスバインディング Operator の安定した最新版のリリースをインストールできます。
5. **Install** をクリックします。

**注記**

Operator は **openshift-operators** namespace に自動的にインストールされます。

6. **Installed operator - ready for use**ペインで、**View Operator** をクリックします。Operator が **Installed Operators** ページに一覧表示されます。
7. **Status** が **Succeeded** に設定されており、サービスバインディング Operator のインストールが正常に行われたことを確認します。

### 6.3.2. 関連情報

- サービスバインディングの使用

## 6.4. サービスバインディングの使用

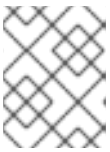
サービスバインディング Operator は、ワークロードおよびバッキングサービスのデータプレーンを管理します。本ガイドでは、データベースインスタンスの作成、アプリケーションのデプロイ、サービスバインディング Operator を使用してアプリケーションとデータベースサービス間のバインディング接続の作成に役立つ例を使用してその手順を説明します。

**前提条件**

- **cluster-admin** パーミッションを持つアカウントを使用して OpenShift Container Platform クラスターにアクセスできる。

ラスターにアクセスできる。

- **oc** CLI がインストールされている。
- OperatorHub からサービスバインディング Operator をインストールしている。
- **v5** Update チャネルを使用して、OperatorHub から Crunchy Postgres for Kubernetes Operator の 5.1.2 バージョンをインストールしました。また、インストールした Operator が、**my-petclinic** namespace など、適切な namespace で利用できる。



**注記**

**oc create namespace my-petclinic** コマンドを使用して namespace を作成できます。

### 6.4.1. PostgreSQL データベースインスタンスの作成

PostgreSQL データベースインスタンスを作成するには、**PostgresCluster** カスタムリソース (CR) を作成し、データベースを設定する必要があります。

**手順**

1. シェルで以下のコマンドを実行して、**my-petclinic** namespace に **PostgresCluster** CR を作成します。

```
$ oc apply -n my-petclinic -f - << EOD
---
apiVersion: postgres-operator.crunchydata.com/v1beta1
kind: PostgresCluster
metadata:
  name: hippo
spec:
  image: registry.developers.crunchydata.com/crunchydata/crunchy-postgres:ubi8-14.4-0
  postgresVersion: 14
  instances:
    - name: instance1
      dataVolumeClaimSpec:
        accessModes:
        - "ReadWriteOnce"
        resources:
          requests:
            storage: 1Gi
  backups:
    pgbackrest:
      image: registry.developers.crunchydata.com/crunchydata/crunchy-pgbackrest:ubi8-2.38-0
      repos:
      - name: repo1
        volume:
          volumeClaimSpec:
            accessModes:
            - "ReadWriteOnce"
            resources:
              requests:
                storage: 1Gi
EOD
```

この **PostgresCluster** CR に追加されたアノテーションは、サービスバインディング接続を有効にし、Operator の調整をトリガーします。

この出力では、データベースインスタンスが作成されていることを検証します。

**出力例**

```
postgrescluster.postgres-operator.crunchydata.com/hippo created
```

2. データベースインスタンスを作成したら、**my-petclinic** namespace のすべての Pod が実行されていることを確認します。

```
$ oc get pods -n my-petclinic
```

出力 (表示に数分かかる) で、データベースが作成され設定されていることを検証できます。

**出力例**

```
NAME                          READY   STATUS      RESTARTS   AGE
hippo-backup-9rxm-88rzq       0/1     Completed   0          2m2s
hippo-instance1-6psd-0        4/4     Running     0          3m28s
hippo-repo-host-0             2/2     Running     0          3m28s
```

データベースを設定したら、サンプルアプリケーションをデプロイしてデータベースサービスに接続できます。

### 6.4.2. Spring PetClinic サンプルアプリケーションのデプロイ

OpenShift Container Platform クラスターに、Spring PetClinic サンプルアプリケーションをデプロイするには、デプロイメント設定を使用し、アプリケーションをテストできるようにローカル環境を設定する必要があります。

**手順**

1. シェルで以下のコマンドを実行して、**spring-petclinic** アプリケーションを **PostgresCluster** カスタムリソース (CR) でデプロイします。

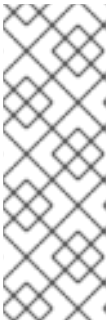
```
$ oc apply -n my-petclinic -f - << EOD
---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: spring-petclinic
  labels:
    app: spring-petclinic
spec:
  replicas: 1
  selector:
    matchLabels:
      app: spring-petclinic
  template:
    metadata:
      labels:
        app: spring-petclinic
```

```
    spec:
      containers:
        - name: app
          image: quay.io/service-binding/spring-petclinic:latest
          imagePullPolicy: Always
          env:
          - name: SPRING_PROFILES_ACTIVE
            value: postgres
          ports:
          - name: http
            containerPort: 8080
---
apiVersion: v1
kind: Service
metadata:
  labels:
    app: spring-petclinic
  name: spring-petclinic
spec:
  type: NodePort
  ports:
    - port: 80
      protocol: TCP
      targetPort: 8080
  selector:
    app: spring-petclinic
EOD
```

この出力では、Spring PetClinic サンプルアプリケーションが作成され、デプロイされていることを確認します。

**出力例**

```
deployment.apps/spring-petclinic created
service/spring-petclinic created
```



**注記**

Web コンソールの **Developer** パースペクティブで**コンテナーイメージ** を使用してアプリケーションをデプロイする場合は、**Advanced options** の **Deployment** セクションで以下の環境変数を入力する必要があります。

- Name: SPRING_PROFILES_ACTIVE
- Value: postgres

2. 以下のコマンドを実行して、アプリケーションがまだデータベースサービスに接続されていないことを確認します。

```
$ oc get pods -n my-petclinic
```

出力に**CrashLoopBackOff** ステータスが表示されるまで、数分かかります。

**出力例**

```
-
```

```
NAME                                READY   STATUS             RESTARTS      AGE
spring-petclinic-5b4c7999d4-wzdtz   0/1     CrashLoopBackOff   4 (13s ago)   2m25s
```

この段階では、Pod は起動に失敗します。アプリケーションとの対話を試みると、エラーが返されます。

3. サービスを公開して、アプリケーションのルートを作成します。

```
$ oc expose service spring-petclinic -n my-petclinic
```

出力は、**spring-petclinic** サービスが公開され、Spring PetClinic サンプルアプリケーションのルートが作成されたことを確認します。

**出力例**

```
route.route.openshift.io/spring-petclinic exposed
```

サービスバインディング Operator を使用すると、アプリケーションをデータベースサービスに接続できるようになります。

### 6.4.3. Spring PetClinic サンプルアプリケーションを PostgreSQL データベースサービスに接続します。

サンプルアプリ ks-本をデータベースサービスに接続するには、サービスバインディング Operator がバインディングデータをアプリケーションにプロジェクションするようにトリガーする **ServiceBinding** カスタムリソース (CR) を作成する必要があります。

**手順**

1. **ServiceBinding** CR を作成し、バインディングデータにパッチを適用します。

```
$ oc apply -n my-petclinic -f - << EOD
---
apiVersion: binding.operators.coreos.com/v1alpha1
kind: ServiceBinding
metadata:
  name: spring-petclinic-pgcluster
spec:
  services: 1
    - group: postgres-operator.crunchydata.com
      version: v1beta1
      kind: PostgresCluster 2
      name: hippo
  application: 3
    name: spring-petclinic
    group: apps
    version: v1
    resource: deployments
EOD
```

1 サービスリソースのリストを指定します。

2 データベースの CR。

3 Deployment または PodSpec が組み込まれた同様のリソースを参照するサンプルアプリケーション。

この出力では、バインディングデータをサンプルアプリケーションにプロジェクションする **ServiceBinding** CR が作成されていることを確認します。

**出力例**

```
servicebinding.binding.operators.coreos.com/spring-petclinic created
```

2. サービスバインディングのリクエストが正常に完了したことを確認します。

```
$ oc get servicebindings -n my-petclinic
```

**出力例**

```
NAME                        READY   REASON            AGE
spring-petclinic-pgcluster  True    ApplicationsBound  7s
```

デフォルトでは、データベースサービスのバインディングデータからの値は、サンプルアプリケーションを実行するワークロードコンテナーにファイルとしてプロジェクションされます。たとえば、Secret リソースからの値はすべて **bindings/spring-petclinic-pgcluster** ディレクトリーに反映されます。

**注記**

オプションとして、ディレクトリーの内容を出力して、アプリケーションのファイルに反映されたバインディングデータが含まれることを確認することもできます。

```
$ for i in username password host port type; do oc exec -it deploy/spring-petclinic -n my-petclinic -- /bin/bash -c 'cd /tmp; find /bindings/*/'$i' -exec echo -n {}:" " \; -exec cat {} \;'; echo; done
```

**出力例: シークレットリソースからのすべての値**

```
/bindings/spring-petclinic-pgcluster/username: <username>
/bindings/spring-petclinic-pgcluster/password: <password>
/bindings/spring-petclinic-pgcluster/host: hippo-primary.my-petclinic.svc
/bindings/spring-petclinic-pgcluster/port: 5432
/bindings/spring-petclinic-pgcluster/type: postgresql
```

3. アプリケーションポートからポート転送を設定し、ローカル環境からサンプルアプリケーションにアクセスします。

```
$ oc port-forward --address 0.0.0.0 svc/spring-petclinic 8080:80 -n my-petclinic
```

**出力例**

```
Forwarding from 0.0.0.0:8080 -> 8080
Handling connection for 8080
```

4. http://localhost:8080/petclinic にアクセスします。
localhost:8080 で Spring PetClinic サンプルアプリケーションにリモートでアクセスできるようになり、アプリケーションがデータベースサービスに接続されていることを確認できます。

### 6.4.4. 関連情報

- サービスバインディング Operator のインストール
- Developer パースペクティブを使用したアプリケーションの作成
- カスタムリソース定義からのリソース管理
- バインド可能な既知の Operator

## 6.5. IBM POWER、IBM Z、および IBM (R) LINUXONE でのサービスバインディングの使用

サービスバインディング Operator は、ワークロードおよびバッキングサービスのデータプレーンを管理します。本ガイドでは、データベースインスタンスの作成、アプリケーションのデプロイ、サービスバインディング Operator を使用してアプリケーションとデータベースサービス間のバインディング接続の作成に役立つ例を使用してその手順を説明します。

**前提条件**

- **cluster-admin** パーミッションを持つアカウントを使用して OpenShift Container Platform クラスターにアクセスできる。
- **oc** CLI がインストールされている。
- OperatorHub からサービスバインディング Operator をインストールしている。

### 6.5.1. PostgreSQL Operator のデプロイ

**手順**

1. **my-petclinic** namespace に Dev4Devs PostgreSQL Operator をデプロイするには、シェルで以下のコマンドを実行します。

```
$ oc apply -f - << EOD
---
apiVersion: v1
kind: Namespace
metadata:
  name: my-petclinic
---
apiVersion: operators.coreos.com/v1
kind: OperatorGroup
metadata:
  name: postgres-operator-group
  namespace: my-petclinic
---
apiVersion: operators.coreos.com/v1alpha1
kind: CatalogSource
metadata:
```

```
  name: ibm-multiarch-catalog
  namespace: openshift-marketplace
spec:
  sourceType: grpc
  image: quay.io/ibm/operator-registry-<architecture> 1
  imagePullPolicy: IfNotPresent
  displayName: ibm-multiarch-catalog
  updateStrategy:
    registryPoll:
      interval: 30m
---
apiVersion: operators.coreos.com/v1alpha1
kind: Subscription
metadata:
  name: postgresql-operator-dev4devs-com
  namespace: openshift-operators
spec:
  channel: alpha
  installPlanApproval: Automatic
  name: postgresql-operator-dev4devs-com
  source: ibm-multiarch-catalog
  sourceNamespace: openshift-marketplace
---
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRole
metadata:
  name: database-view
  labels:
    servicebinding.io/controller: "true"
rules:
  - apiGroups:
      - postgresql.dev4devs.com
    resources:
      - databases
    verbs:
      - get
      - list
EOD
```

1 Operator イメージ

- IBM Power:**quay.io/ibm/operator-registry-ppc64le:release-4.9**
- IBM Z および IBM® LinuxONE: **quay.io/ibm/operator-registry-s390x:release-4.8**

**検証**

1. Operator のインストール後に、**openshift-operators** namespace の Operator サブスクリプションを一覧表示します。

```
$ oc get subs -n openshift-operators
```

**出力例**

```
NAME                              PACKAGE                           SOURCE              CHANNEL
postgresql-operator-dev4devs-com  postgresql-operator-dev4devs-com  ibm-multiarch-
catalog  alpha
rh-service-binding-operator       rh-service-binding-operator       redhat-operators    stable
```

### 6.5.2. PostgreSQL データベースインスタンスの作成

PostgreSQL データベースインスタンスを作成するには、**Database** カスタムリソース (CR) を作成し、データベースを設定する必要があります。

**手順**

1. シェルで以下のコマンドを実行して、**my-petclinic** namespace に **Database** CR を作成します。

```
$ oc apply -f - << EOD
apiVersion: postgresql.dev4devs.com/v1alpha1
kind: Database
metadata:
  name: sampledatabase
  namespace: my-petclinic
  annotations:
    host: sampledatabase
    type: postgresql
    port: "5432"
    service.binding/database: 'path={.spec.databaseName}'
    service.binding/port: 'path={.metadata.annotations.port}'
    service.binding/password: 'path={.spec.databasePassword}'
    service.binding/username: 'path={.spec.databaseUser}'
    service.binding/type: 'path={.metadata.annotations.type}'
    service.binding/host: 'path={.metadata.annotations.host}'
spec:
  databaseCpu: 30m
  databaseCpuLimit: 60m
  databaseMemoryLimit: 512Mi
  databaseMemoryRequest: 128Mi
  databaseName: "sampledb"
  databaseNameKeyEnvVar: POSTGRESQL_DATABASE
  databasePassword: "samplepwd"
  databasePasswordKeyEnvVar: POSTGRESQL_PASSWORD
  databaseStorageRequest: 1Gi
  databaseUser: "sampleuser"
  databaseUserKeyEnvVar: POSTGRESQL_USER
  image: registry.redhat.io/rhel8/postgresql-13:latest
  databaseStorageClassName: nfs-storage-provisioner
  size: 1
EOD
```

この **Database** CR に追加されたアノテーションは、サービスバインディング接続を有効にし、Operator の調整をトリガーします。

この出力では、データベースインスタンスが作成されていることを検証します。

**出力例**

```
database.postgresql.dev4devs.com/sampledatabase created
```

2. データベースインスタンスを作成したら、**my-petclinic** namespace のすべての Pod が実行されていることを確認します。

```
$ oc get pods -n my-petclinic
```

出力 (表示に数分かかる) で、データベースが作成され設定されていることを検証できます。

**出力例**

```
NAME                                READY   STATUS    RESTARTS   AGE
sampledatabase-cbc655488-74kss      0/1     Running   0          32s
```

データベースを設定したら、サンプルアプリケーションをデプロイしてデータベースサービスに接続できます。

### 6.5.3. Spring PetClinic サンプルアプリケーションのデプロイ

OpenShift Container Platform クラスターに、Spring PetClinic サンプルアプリケーションをデプロイするには、デプロイメント設定を使用し、アプリケーションをテストできるようにローカル環境を設定する必要があります。

**手順**

1. シェルで以下のコマンドを実行して、**spring-petclinic** アプリケーションを **PostgresCluster** カスタムリソース (CR) でデプロイします。

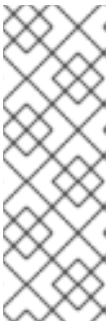
```
$ oc apply -n my-petclinic -f - << EOD
---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: spring-petclinic
  labels:
    app: spring-petclinic
spec:
  replicas: 1
  selector:
    matchLabels:
      app: spring-petclinic
  template:
    metadata:
      labels:
        app: spring-petclinic
    spec:
      containers:
        - name: app
          image: quay.io/service-binding/spring-petclinic:latest
          imagePullPolicy: Always
          env:
          - name: SPRING_PROFILES_ACTIVE
            value: postgres
          - name: org.springframework.cloud.bindings.boot.enable
```

```
          value: "true"
        ports:
        - name: http
          containerPort: 8080
---
apiVersion: v1
kind: Service
metadata:
  labels:
    app: spring-petclinic
  name: spring-petclinic
spec:
  type: NodePort
  ports:
    - port: 80
      protocol: TCP
      targetPort: 8080
  selector:
    app: spring-petclinic
EOD
```

この出力では、Spring PetClinic サンプルアプリケーションが作成され、デプロイされていることを確認します。

**出力例**

```
deployment.apps/spring-petclinic created
service/spring-petclinic created
```



**注記**

Web コンソールの **Developer** パースペクティブで**コンテナーイメージ** を使用してアプリケーションをデプロイする場合は、**Advanced options** の **Deployment** セクションで以下の環境変数を入力する必要があります。

- Name: SPRING_PROFILES_ACTIVE
- Value: postgres

2. 以下のコマンドを実行して、アプリケーションがまだデータベースサービスに接続されていないことを確認します。

```
$ oc get pods -n my-petclinic
```

**CrashLoopBackOff** ステータスが表示されるまで数分かかります。

**出力例**

```
NAME                              READY   STATUS             RESTARTS      AGE
spring-petclinic-5b4c7999d4-wzdtz   0/1     CrashLoopBackOff   4 (13s ago)   2m25s
```

この段階では、Pod は起動に失敗します。アプリケーションとの対話を試みると、エラーが返されます。

サービスバインディング Operator を使用すると、アプリケーションをデータベースサービスに接続できるようになります。

### 6.5.4. Spring PetClinic サンプルアプリケーションを PostgreSQL データベースサービスに接続します。

サンプルアプリ ks-本をデータベースサービスに接続するには、サービスバインディング Operator がバインディングデータをアプリケーションにプロジェクションするようにトリガーする **ServiceBinding** カスタムリソース (CR) を作成する必要があります。

**手順**

1. **ServiceBinding** CR を作成し、バインディングデータにパッチを適用します。

```
$ oc apply -n my-petclinic -f - << EOD
---
apiVersion: binding.operators.coreos.com/v1alpha1
kind: ServiceBinding
metadata:
   name: spring-petclinic-pgcluster
spec:
 services: 1
   - group: postgresql.dev4devs.com
     kind: Database 2
     name: sampledatabase
     version: v1alpha1
 application: 3
   name: spring-petclinic
   group: apps
   version: v1
   resource: deployments
EOD
```

1. サービスリソースのリストを指定します。
2. データベースの CR。
3. Deployment または PodSpec が組み込まれた同様のリソースを参照するサンプルアプリケーション。

この出力では、バインディングデータをサンプルアプリケーションにプロジェクションする **ServiceBinding** CR が作成されていることを確認します。

**出力例**

```
servicebinding.binding.operators.coreos.com/spring-petclinic created
```

2. サービスバインディングのリクエストが正常に完了したことを確認します。

```
$ oc get servicebindings -n my-petclinic
```

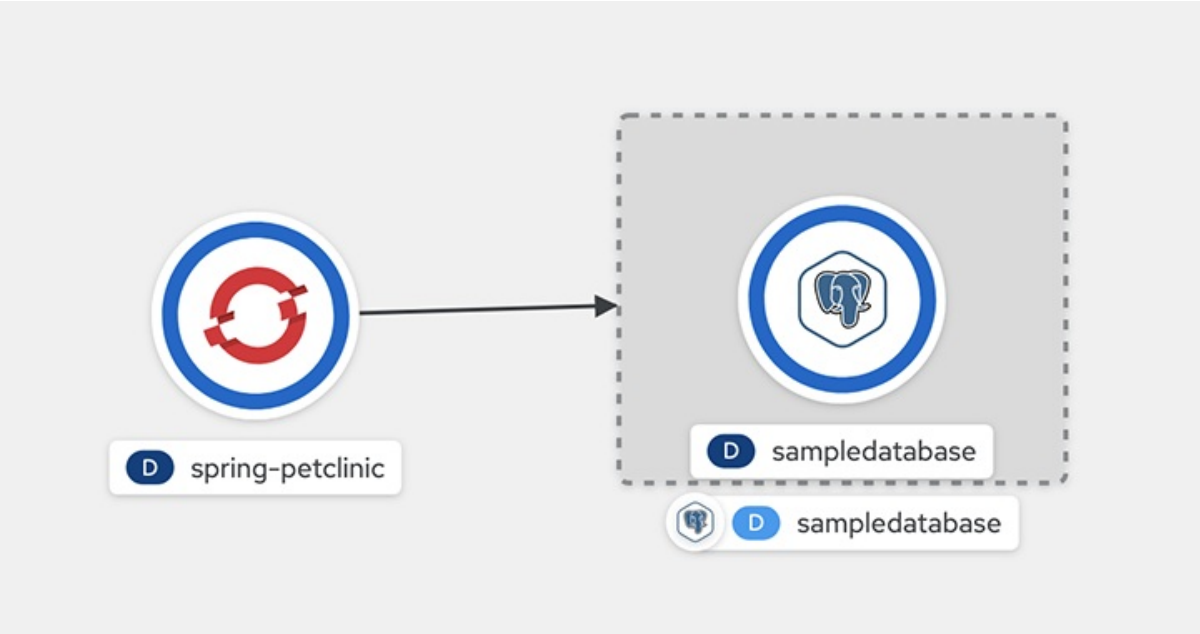
**出力例**

```
NAME                          READY   REASON            AGE
spring-petclinic-postgresql   True    ApplicationsBound   47m
```

デフォルトでは、データベースサービスのバインディングデータからの値は、サンプルアプリケーションを実行するワークロードコンテナーにファイルとしてプロジェクションされます。たとえば、Secret リソースからの値はすべて **bindings/spring-petclinic-pgcluster** ディレクトリーに反映されます。

3. これが作成されたら、トポロジーに移動し、接続を視覚的に確認できます。

**図6.1 spring-petclinic のサンプルデータベースへの接続**




4. アプリケーションポートからポート転送を設定し、ローカル環境からサンプルアプリケーションにアクセスします。

```
$ oc port-forward --address 0.0.0.0 svc/spring-petclinic 8080:80 -n my-petclinic
```

**出力例**

```
Forwarding from 0.0.0.0:8080 -> 8080
Handling connection for 8080
```

5. http://localhost:8080 にアクセスします。
localhost:8080 で Spring PetClinic サンプルアプリケーションにリモートでアクセスできるようになり、アプリケーションがデータベースサービスに接続されていることを確認できます。

### 6.5.5. 関連情報

- サービスバインディング Operator のインストール
- Developer パースペクティブを使用したアプリケーションの作成
- カスタムリソース定義からのリソースの管理

## 6.6. サービスからバインディングデータの公開

アプリケーション開発者は、ワークロードをビルドして接続するバッキングサービスへのアクセスが必要です。ワークロードをバッキングサービスに接続するのは、サービスプロバイダーごと、シークレットにアクセスしてワークロードで消費するのに必要となる方法が異なるので、困難です。

サービスバインディング Operator を使用すると、アプリケーション開発者は、手作業でバインディング接続を設定する手順なしに、オペレーターが管理するバッキングサービスとワークロードを簡単にバインドできます。サービスバインディングオペレーターがバインディングデータを提供するには、オペレータープロバイダーまたはバッキングサービスを作成するユーザーが、サービスバインディングオペレーターによって自動的に検出されるようにバインディングデータを公開する必要があります。次に、サービスバインディング Operator は、バッキングサービスからバインディングデータを自動的に収集し、ワークロードと共有して、一貫性のある、予測可能なエクスペリエンスを提供します。

### 6.6.1. バインディングデータを公開する方法

本セクションでは、バインディングデータの公開に使用できる方法について説明します。

ワークロードの要件や環境、および提供されるサービスとの連携方法を理解しておくようにしてください。

バインディングデータは以下の状況下で公開されます。

- バッキングサービスは、プロビジョニングされたサービスリソースとして利用できます。接続するサービスはサービスバインディング仕様に準拠するものになります。必要なバインディングデータ値すべてを使用して **Secret** リソースを作成し、バッキングサービスカスタムリソース (CR) で参照する必要があります。すべてのバインディングデータ値の検出は自動的に実行されます。
- バッキングサービスは、プロビジョニングされたサービスリソースとしては利用できません。バッキングサービスからバインディングデータを公開する必要があります。ワークロード要件および環境に応じて、以下のいずれかの方法でバインディングデータを公開することができます。
  - 直接のシークレット参照
  - カスタムリソース定義 (CRD) または CR アノテーションを使用したバインディングデータの宣言
  - 所有リソースによるバインディングデータの検出

#### 6.6.1.1. プロビジョニングされたサービス

プロビジョニングされたサービスは、バッキングサービス CR の **.status.binding.name** フィールドに配置された **Secret** リソースへの参照のあるバッキングサービス CR を表します。

Operator プロバイダーまたは、バッキングサービスを作成するユーザーが、**Secret** リソースを作成し、バッキングサービス CR の **status.binding.name** セクションでその CR を参照して、この方法を使用してサービスバインディング仕様に準拠できます。この **Secret** リソースは、バッキングサービスに接続するためにワークロードに必要なすべてのバインディングデータ値を指定する必要があります。

以下の例は、バッキングサービスおよび CR から参照される **Secret** リソースを表す **AccountService** CR を示しています。

**例: AccountService CR**

```
apiVersion: example.com/v1alpha1
kind: AccountService
```

```
name: prod-account-service
spec:
# ...
status:
  binding:
    name: hippo-pguser-hippo
```

**例: 参照された Secret リソース**

```
apiVersion: v1
kind: Secret
metadata:
  name: hippo-pguser-hippo
data:
  password: "<password>"
  user: "<username>"
# ...
```

サービスバインディングリソースを作成するとき、次のように **ServiceBinding仕様で AccountService** リソースの詳細を直接指定できます。

**ServiceBinding リソースの例**

```
apiVersion: binding.operators.coreos.com/v1alpha1
kind: ServiceBinding
metadata:
  name: account-service
spec:
# ...
  services:
  - group: "example.com"
    version: v1alpha1
    kind: AccountService
    name: prod-account-service
  application:
    name: spring-petclinic
    group: apps
    version: v1
    resource: deployments
```

**例: 仕様 API での ServiceBinding リソース**

```
apiVersion: servicebinding.io/v1beta1
kind: ServiceBinding
metadata:
  name: account-service
spec:
# ...
  service:
    apiVersion: example.com/v1alpha1
    kind: AccountService
    name: prod-account-service
  workload:
```

```
    apiVersion: apps/v1
    kind: Deployment
    name: spring-petclinic
```

この方法では、ワークロードにプロジェクションされるバインディングデータとして、**Secret** リソースを参照する **hippo-pguser-hippo** に、すべてのキーを公開します。

#### 6.6.1.2. 直接のシークレット参照

サービスバインディング定義で参照できる **Secret** リソースで、必要なバインディングデータ値すべてが利用できる場合にこの手法使用できます。この方法では、**ServiceBinding** リソースは **Secret** リソースを直接参照し、サービスに接続します。**Secret** リソースの全キーがバインディングデータとして公開されます。

**例: binding.operators.coreos.com API での仕様**

```
apiVersion: binding.operators.coreos.com/v1alpha1
kind: ServiceBinding
metadata:
  name: account-service
spec:
# ...
  services:
  - group: ""
    version: v1
    kind: Secret
    name: hippo-pguser-hippo
```

**例: servicebinding.io API に準拠した仕様**

```
apiVersion: servicebinding.io/v1beta1
kind: ServiceBinding
metadata:
  name: account-service
spec:
# ...
  service:
    apiVersion: v1
    kind: Secret
    name: hippo-pguser-hippo
```

#### 6.6.1.3. CRD または CR アノテーションによるバインディングデータを宣言する

この方法を使用して、バッキングサービスのリソースにアノテーションを付け、バインディングデータを特定のアノテーションで公開できます。**metadata** セクションにアノテーションを追加すると、バッキングサービスの CR および CRD が変更されます。サービスバインディング Operator は CR および CRD に追加されるアノテーションを検出し、アノテーションに基づいて抽出された値を使用して **Secret** リソースを作成します。

以下の例は、**metadata** セクションに追加されるアノテーションと、リソースから参照される **ConfigMap** オブジェクトを示しています。

**例:CR アノテーションで定義される Secret オブジェクトからのバインディングデータの公開**

```
apiVersion: postgres-operator.crunchydata.com/v1beta1
kind: PostgresCluster
metadata:
  name: hippo
  namespace: my-petclinic
  annotations:
    service.binding: 'path={.metadata.name}-pguser-{.metadata.name},objectType=Secret'
# ...
```

上記の例では、**hippo-pguser-hippo** に解決する **{.metadata.name}-pguser-{.metadata.name}** テンプレートにシークレット名の名前を配置します。テンプレートには複数の JSONPath 表現を含めることができます。

**例: リソースからの参照された Secret オブジェクト**

```
apiVersion: v1
kind: Secret
metadata:
  name: hippo-pguser-hippo
data:
  password: "<password>"
  user: "<username>"
```

**例:CR アノテーションで定義される ConfigMap オブジェクトからのバインディングデータの公開**

```
apiVersion: postgres-operator.crunchydata.com/v1beta1
kind: PostgresCluster
metadata:
  name: hippo
  namespace: my-petclinic
  annotations:
    service.binding: 'path={.metadata.name}-config,objectType=ConfigMap'
# ...
```

上記の例では、**hippo-config** に解決する **{.metadata.name}-config** テンプレートに設定マップの名前を配置します。テンプレートには複数の JSONPath 表現を含めることができます。

**例: リソースからの参照された ConfigMap オブジェクト**

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: hippo-config
data:
  db_timeout: "10s"
  user: "hippo"
```

#### 6.6.1.4. 所有リソースによるバインディングデータの検出

バッキングサービスが、バインディングデータの検出に使用できるルート、サービス、設定マップ、シークレットなど、1つ以上の Kubernetes リソースを所有している場合は、このメソッドを使用できます。この方法では、Service Binding Operator は、バッキングサービス CR が所有するリソースからバ

インディングデータを検出します。

次の例では、**detectBindingResourcesAPI** オプションが **ServiceBindingCR** で **true** に設定されています。

**例**

```
apiVersion: binding.operators.coreos.com/v1alpha1
kind: ServiceBinding
metadata:
  name: spring-petclinic-detect-all
  namespace: my-petclinic
spec:
  detectBindingResources: true
  services:
    - group: postgres-operator.crunchydata.com
      version: v1beta1
      kind: PostgresCluster
      name: hippo
  application:
    name: spring-petclinic
    group: apps
    version: v1
    resource: deployments
```

直前の例では、**PostgresCluster** カスタムリソースはルート、サービス、設定マップ、またはシークレットなどの 1 つ以上の Kubernetes リソースを所有します。

サービスバインディング Operator は、所有リソースごとに公開されるバインディングデータを自動的に検出します。

### 6.6.2. データモデル

アノテーションで使用されるデータモデルは、特定の規則に従います。

サービスバインディングアノテーションは、以下の規則を使用する必要があります。

```
service.binding(/<NAME>)?:
    "<VALUE>|(path=<JSONPATH_TEMPLATE>(,objectType=<OBJECT_TYPE>)?(,elementType=
<ELEMENT_TYPE>)?(,sourceKey=<SOURCE_KEY>)?(,sourceValue=<SOURCE_VALUE>)?)"
```

ここでは、以下のようになります。

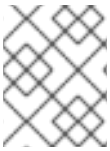
| | |
|---|---|
| **<NAME>** | バインディング値を公開する名前を指定します。**objectType** パラメーターが **Secret** または **ConfigMap** に設定されている場合にのみ除外できます。 |
| **<VALUE>** | **path** が設定されていない場合に公開される定数値を指定します。 |

データモデルは、**path**、**elementType**、**objectType**、**sourceKey**、および **sourceValue** パラメーターの許可される値とセマンティックの詳細を提供します。

**表6.4 パラメーターおよびその説明**

| パラメーター | 説明 | デフォルト値 |
| --- | --- | --- |
| **path** | 中かっこ {} で囲まれた JSONPath 表現で設定される JSONPath テンプレート。 | 該当なし |
| **elementType** | **path** パラメーターで参照される要素の値が以下のいずれかのタイプに準拠するかどうかを指定します。<br>● **string**<br>● **sliceOfStrings**<br>● **sliceOfMaps** | **string** |
| **objectType** | **path** パラメーターで示される要素の値が、現在の namespace の **ConfigMap**、**Secret**、または平文の文字列を参照するかどうかを指定します。 | **secret**(**elementType** が文字列以外の場合) |
| **sourceKey** | バインディングデータを収集する際にバインディングシークレットに追加される **ConfigMap** または **Secret** リソースのキーを指定します。<br>注記:<br>● **elementType**=**sliceOfMaps** と併用される場合、**sourceKey** パラメーターは、値がバインディングシークレットのキーとして使用される、マップのスライスのキーを指定します。<br>● このオプションパラメーターを使用して、参照される **Secret** または **ConfigMap** リソースの特定のエントリーをバインディングデータとして公開します。<br>● 指定されていない場合、**Secret** または **ConfigMap** リソースからのすべてのキーと値が公開され、バインディングシークレットに追加されます。 | 該当なし |

| パラメーター | 説明 | デフォルト値 |
| --- | --- | --- |
| **sourceValue** | マップのスライスのキーを指定します。<br><br>注記:<br><br>● このキーの値は、バインディングシークレットに追加されるキーと値のペアのエントリーの値を生成するベースとして使用されます。<br><br>● さらに、**sourceKey** の値は、バインディングシークレットに追加されるキーと値のペアのエントリーのキーとして使用されます。<br><br>● **elementType=sliceOfMaps** の場合のみ必須です。 | 該当なし |



**注記**

**sourceKey** および **sourceValue** パラメーターは、**path** パラメーターで指定された要素が **ConfigMap** または **Secret** リソースを参照する場合にのみ適用されます。

### 6.6.3. アノテーションマッピングをオプションに設定する

アノテーションにはオプションのフィールドを含めることができます。たとえば、サービスエンドポイントが認証を必要としない場合、資格情報へのパスが存在しない可能性があります。このような場合、アノテーションのターゲットパスにフィールドが存在しない可能性があります。その結果、Service Binding Operator はデフォルトでエラーを生成します。

サービスプロバイダーは、アノテーションマッピングが必要かどうかを示すために、サービスを有効にするときにアノテーションに **optional** フラグの値を設定できます。Service Binding Operator は、ターゲットパスが使用可能な場合にのみ、アノテーションマッピングを提供します。ターゲットパスが利用できない場合、Service Binding Operator はオプションのマッピングをスキップし、エラーを出力することなく既存のマッピングの展開を続行します。

**手順**

- アノテーションのフィールドをオプションにするには、**optional** フラグ値を **true** に設定します。

  **例**

```
apiVersion: apps.example.org/v1beta1
kind: Database
metadata:
  name: my-db
  namespace: my-petclinic
  annotations:
    service.binding/username: path={.spec.name},optional=true
# ...
```

注記

- **optional** フラグ値を **false** に設定し、Service Binding Operator がターゲットパスを見つけることができない場合、Operator はアノテーションマッピングに失敗します。
- **optional** のフラグに値が設定されていない場合、サービスバインディング Operator はデフォルトで値を **false** と見なし、アノテーションマッピングに失敗します。

### 6.6.4. RBAC 要件

サービスバインディング Operator を使用してバッキングサービスバインディングデータを公開するには、特定のロールベースアクセス制御 (RBAC) パーミッションが必要になります。**ClusterRole** リソースの **rules** フィールドに特定の動詞を指定し、バッキングサービスリソースの RBAC パーミッションを付与します。これらの **rules** を定義すると、サービスバインディング Operator はクラスター全体でバッキングサービスリソースのバインディングデータを読み取ることができます。ユーザーにバインディングデータの読み取りまたはアプリケーションリソースの変更のパーミッションがない場合、サービスバインディング Operator はこのようなユーザーがサービスをアプリケーションにバインドできないようにします。RBAC 要件を順守することで、ユーザーの不要なパーミッション昇格を回避し、承認されていないサービスまたはアプリケーションへのアクセスを防ぎます。

サービスバインディング Operator は、専用のサービスアカウントを使用して Kubernetes API に対してリクエストを実行します。デフォルトでは、このアカウントはサービスをワークロードにバインドするためのパーミッションを持ち、共に以下の標準の Kubernetes または OpenShift オブジェクトで表されます。

- **デプロイメント**
- **DaemonSets**
- **ReplicaSet**
- **StatefulSets**
- **DeploymentConfig**

Operator サービスアカウントは集約されたクラスターロールにバインドされ、Operator プロバイダーまたはクラスター管理者はワークロードへのカスタムサービスリソースのバインドを有効にできます。**ClusterRole** 内の必要なパーミッションを付与するには、これに **servicebinding.io/controller** フラグでラベルを付け、フラグの値を **true** に設定します。以下の例は、サービスバインディング Operator が Crunchy PostgreSQL Operator のカスタムリソース (CR) を **取得**、**監視**、および **一覧表示** するのを許可する方法を示しています。

**例:Crunchy PostgreSQL Operator によってプロビジョニングされる PostgreSQL データベースインスタンスへのバインディングの有効化**

```
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRole
metadata:
  name: postgrescluster-reader
  labels:
    servicebinding.io/controller: "true"
rules:
- apiGroups:
  - postgres-operator.crunchydata.com
```

```
  resources:
    - postgresclusters
  verbs:
    - get
    - watch
    - list
  ...
```

このクラスターロールは、バッキングサービス Operator のインストール時にデプロイできます。

### 6.6.5. 公開可能なバインディングデータのカテゴリー

サービスバインディング Operator を使用すると、バッキングサービスリソースおよびカスタムリソース定義 (CRD) からバインディングデータ値を公開できます。

本セクションでは、さまざまな公開可能なバインディングデータのカテゴリーを使用する方法を例とともに紹介します。これらのサンプルは、実際の環境と要件に合わせて変更する必要があります。

#### 6.6.5.1. リソースからの文字列の公開

以下の例は、**PostgresCluster** カスタムリソース (CR) の **metadata.name** フィールドから文字列を公開する方法を示しています。

**例**

```
apiVersion: postgres-operator.crunchydata.com/v1beta1
kind: PostgresCluster
metadata:
  name: hippo
  namespace: my-petclinic
  annotations:
    service.binding/username: path={.metadata.name}
# ...
```

#### 6.6.5.2. 定数値のバインディング項目としての公開

以下の例は、**PostgresCluster** カスタムリソース (CR) から定数値を公開する方法を示しています。

**例: 定数値の公開**

```
apiVersion: postgres-operator.crunchydata.com/v1beta1
kind: PostgresCluster
metadata:
  name: hippo
  namespace: my-petclinic
  annotations:
    "service.binding/type": "postgresql" (1)
```

(1) **postgresql** 値で公開されるバインディング **タイプ**。

#### 6.6.5.3. リソースから参照される設定マップまたはシークレット全体を公開する

以下の例では、シークレット全体をアノテーションにより公開する方法を説明します。

**例: アノテーションによるシークレット全体の公開**

```
apiVersion: postgres-operator.crunchydata.com/v1beta1
kind: PostgresCluster
metadata:
  name: hippo
  namespace: my-petclinic
  annotations:
    service.binding: 'path={.metadata.name}-pguser-{.metadata.name},objectType=Secret'
```

**例: バッキングサービスリソースから参照されるシークレット**

```
apiVersion: v1
kind: Secret
metadata:
  name: hippo-pguser-hippo
data:
  password: "<password>"
  user: "<username>"
```

#### 6.6.5.4. リソースから参照される設定マップまたはシークレットから特定のエントリーを公開する

以下の例では、アノテーションにより設定マップから特定のエントリーを公開する方法を説明します。

**例: アノテーションを使用した設定マップからのエントリーの公開**

```
apiVersion: postgres-operator.crunchydata.com/v1beta1
kind: PostgresCluster
metadata:
  name: hippo
  namespace: my-petclinic
  annotations:
    service.binding: 'path={.metadata.name}-config,objectType=ConfigMap,sourceKey=user'
```

**例: バッキングサービスリソースから参照される設定マップ**

バインディングデータには、名前が **db_timeout**、値が **10s** のキーが必要です。

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: hippo-config
data:
  db_timeout: "10s"
  user: "hippo"
```

#### 6.6.5.5. リソース定義値の公開

以下の例は、リソース定義の値をアノテーションを使用して公開する方法を説明します。

**例: アノテーションによるリソース定義値の公開**

```
apiVersion: postgres-operator.crunchydata.com/v1beta1
kind: PostgresCluster
metadata:
  name: hippo
  namespace: my-petclinic
  annotations:
    service.binding/username: path={.metadata.name}
    ...
```

#### 6.6.5.6. コレクションのエントリーを、各エントリーのキーと値で公開する

以下の例は、アノテーションを使用して各エントリーのキーと値を持つコレクションのエントリーを公開する方法を示しています。

**例: アノテーションによるコレクションのエントリーの公開**

```
apiVersion: postgres-operator.crunchydata.com/v1beta1
kind: PostgresCluster
metadata:
  name: hippo
  namespace: my-petclinic
  annotations:
    "service.binding/uri": "path=
{.status.connections},elementType=sliceOfMaps,sourceKey=type,sourceValue=url"
spec:
# ...
status:
  connections:
    - type: primary
      url: primary.example.com
    - type: secondary
      url: secondary.example.com
    - type: '404'
      url: black-hole.example.com
```

以下の例では、1つ前のアノテーションでのコレクションエントリーが、バインドされたアプリケーションにどのようにプロジェクションされるかを紹介します。

**例: データファイルのバインディング**

```
/bindings/<binding-name>/uri_primary => primary.example.com
/bindings/<binding-name>/uri_secondary => secondary.example.com
/bindings/<binding-name>/uri_404 => black-hole.example.com
```

**例: バッキングサービスリソースの設定**

```
status:
  connections:
    - type: primary
      url: primary.example.com
    - type: secondary
```

```
      url: secondary.example.com
    - type: '404'
      url: black-hole.example.com
```

上記の例では、**primary**、**secondary** などのキーを使用したすべての値をプロジェクションできるようにします。

#### 6.6.5.7. コレクションのアイテムをアイテムごとに1つのキーで公開する

以下の例は、アノテーションを使用して項目ごとに1つのキーを持つコレクションの項目を公開する方法を示しています。

**例: アノテーションによるコレクションの項目の公開**

```
apiVersion: postgres-operator.crunchydata.com/v1beta1
kind: PostgresCluster
metadata:
  name: hippo
  namespace: my-petclinic
  annotations:
    "service.binding/tags": "path={.spec.tags},elementType=sliceOfStrings"
spec:
    tags:
      - knowledge
      - is
      - power
```

以下の例では、1つ前のアノテーションでのコレクションアイテムが、バインドされたアプリケーションにどのようにプロジェクションされるかを紹介します。

**例: データファイルのバインディング**

```
/bindings/<binding-name>/tags_0 => knowledge
/bindings/<binding-name>/tags_1 => is
/bindings/<binding-name>/tags_2 => power
```

**例: バッキングサービスリソースの設定**

```
spec:
  tags:
  - knowledge
  - is
  - power
```

#### 6.6.5.8. エントリー値ごとに1つのキーを使用してコレクションエントリーの値を公開する

以下の例は、アノテーションを使用してエントリー値ごとに1つのキーを持つコレクションエントリーの値を公開する方法を示しています。

**例: アノテーションを使用したコレクションエントリーの値の公開**

```
apiVersion: postgres-operator.crunchydata.com/v1beta1
kind: PostgresCluster
```

```
metadata:
  name: hippo
  namespace: my-petclinic
  annotations:
    "service.binding/url": "path={.spec.connections},elementType=sliceOfStrings,sourceValue=url"
spec:
  connections:
    - type: primary
      url: primary.example.com
    - type: secondary
      url: secondary.example.com
    - type: '404'
      url: black-hole.example.com
```

以下の例では、1 つ前のアノテーションでのコレクション値が、バインドされたアプリケーションにどのようにプロジェクションされるかを紹介します。

**例: データファイルのバインディング**

```
/bindings/<binding-name>/url_0 => primary.example.com
/bindings/<binding-name>/url_1 => secondary.example.com
/bindings/<binding-name>/url_2 => black-hole.example.com
```

### 6.6.6. 関連情報

- クラスターサービスバージョン (CSV) の定義
- バインドデータのプロジェクション

## 6.7. バインディングデータのプロジェクション

本セクションでは、バインディングデータを使用する方法について説明します。

### 6.7.1. バインディングデータの使用

バッキングサービスがバインディングデータを公開した後、ワークロードがこのデータにアクセスして消費するには、バッキングサービスからワークロードにデータをプロジェクションする必要があります。サービスバインディング Operator は、以下のいずれかの方法でデータセットをワークロードに自動的にプロジェクションします。

1. ファイルとして (デフォルト)。
2. 環境変数として。(**ServiceBinding** リソースから **.spec.bindAsFiles** パラメーターを設定した後)。

### 6.7.2. ワークロードコンテナー内にバインディングデータをプロジェうションするディレクトリーパスの設定

デフォルトでは、サービスバインディング Operator は、バインディングデータをファイルとしてワークロードリソースの特定のディレクトリーにマウントします。ワークロードが実行されるコンテナーで設定された **SERVICE_BINDING_ROOT** 環境変数を使用してディレクトリーパスを設定できます。

**例: ファイルとしてマウントされるバインディングデータ**

```
$SERVICE_BINDING_ROOT 1
├── account-database 2
│   ├── type 3
│   ├── provider 4
│   ├── uri
│   ├── username
│   └── password
└── transaction-event-stream 5
    ├── type
    ├── connection-count
    ├── uri
    ├── certificates
    └── private-key
```

1 ルートディレクトリー。

2 5 バインディングデータを保存するディレクトリー。

3 対応するディレクトリーにプロジェクションされるバインディングデータのタイプを識別する必須の ID。

4 オプション: アプリケーションが接続できるバッキングサービスのタイプを識別できるように、プロバイダーを識別するための ID。

バインディングデータを環境変数として使用するには、環境変数の読み取りに使用できる任意のプログラミング言語の組み込み言語機能を使用します。

**例: Python クライアントの使用**

```
import os
username = os.getenv("USERNAME")
password = os.getenv("PASSWORD")
```

**警告**

**バインディングデータのディレクトリー名を使用してバインディングデータを検索する場合**

Service Binding Operator は、**ServiceBinding** リソース名 (**.metadata.name**) をバインディングデータディレクトリー名として使用します。この仕様は、**.spec.name** フィールドを介してその名前をオーバーライドする方法も提供します。その結果、namespace に複数の **ServiceBinding** リソースがある場合、バインディングデータ名の競合が発生する可能性があります。ただし、Kubernetes でのボリュームマウントの性質上、バインディングデータディレクトリーには **シークレット** リソースの 1 つのみからの値が含まれます。

### 6.7.2.1. バインディングデータをファイルとしてプロジェクションするための最終パスの計算

以下の表は、ファイルが指定のディレクトリーにマウントされるときに、バインディクデータプロジェクションの最終パスを計算する方法に関する設定をまとめています。

**表6.5 最終パスの計算の概要**

| SERVICE_BINDING_ROOT | 最終パス |
|---|---|
| 利用不可 | **/bindings/<ServiceBinding_ResourceName>** |
| **dir/path/root** | **dir/path/root/<ServiceBinding_ResourceName>** |

1 つ前の表の **<ServiceBinding_ResourceName>** エントリーは、カスタムリソース (CR) の **.metadata.name** セクションで設定する **ServiceBinding** リソースの名前を指定します。

**注記**

デフォルトでは、展開されたファイルのアクセス許可は 0644 に設定されています。Service Binding Operator は、サービスが **0600** などの特定権限を想定する場合に問題を引き起こす Kubernetes のバグにより、特定の権限を設定できません。回避策として、ワークロードリソース内で実行されているプログラムまたはアプリケーションのコードを変更して、ファイルを **/tmp** ディレクトリーにコピーし、適切な権限を設定することができます。

既存の **SERVICE_BINDING_ROOT** 環境変数内のバインディングデータにアクセスして使用するには、環境変数を読み取れる任意のプログラミング言語の組み込み言語機能を使用します。

**例: Python クライアントの使用**

```
from pyservicebinding import binding
try:
    sb = binding.ServiceBinding()
except binding.ServiceBindingRootMissingError as msg:
    # log the error message and retry/exit
    print("SERVICE_BINDING_ROOT env var not set")
sb = binding.ServiceBinding()
bindings_list = sb.bindings("postgresql")
```

直前の例では、**bindings_list** 変数には、**postgresql** データベースサービスタイプのバインディングデータが含まれます。

### 6.7.3. バインディングデータのプロジェクション

ワークロード要件および環境に応じて、ファイルまたは環境変数としてバインディングデータをプロジェクションすることができます。

**前提条件**

- 以下の概念について理解しておく。
  - ワークロードの環境および要件、指定のサービスと連携する方法。
  - ワークロードリソースでのバインディングデータ消費量。

- デフォルトの方法でデータプロジェクションの最終パスを計算する方法の設定。

- バインディングデータがバッキングサービスから公開されている。

**手順**

1. ファイルとしてバインディングデータをプロジェクションするには、既存の **SERVICE_BINDING_ROOT** 環境変数がワークロードが実行されるコンテナーで存在することを確認して、宛先フォルダーを決定します。

2. バインドデータを環境変数としてプロジェクションするには、カスタムリソース (CR) の **ServiceBinding** リソースから、 **.spec.bindAsFiles** パラメーターの値を **false** に設定します。

### 6.7.4. 関連情報

- サービスからバインディングデータの公開

- アプリケーションのソースコードでの反映されたバインディングデータの使用

## 6.8. サービスバインディング OPERATOR を使用したワークロードのバインド

アプリケーション開発者は、バインディングシークレットを使用して、ワークロードを 1 つまたは複数のバッキングサービスにバインドする必要があります。このシークレットは、ワークロードによって使用される情報を保存するために生成されます。

たとえば、接続するサービスがすでにバインディングデータを公開しているとします。この場合、**ServiceBinding** カスタムリソース (CR) と共に、使用されるワークロードの必要になります。この **ServiceBinding** CR を使用することで、ワークロードはバインドするサービスの詳細と共にバインディング要求を送信します。

**ServiceBinding CR の例**

```
apiVersion: binding.operators.coreos.com/v1alpha1
kind: ServiceBinding
metadata:
  name: spring-petclinic-pgcluster
  namespace: my-petclinic
spec:
  services: 1
  - group: postgres-operator.crunchydata.com
    version: v1beta1
    kind: PostgresCluster
    name: hippo
  application: 2
    name: spring-petclinic
    group: apps
    version: v1
    resource: deployments
```

1 サービスリソースの一覧を指定します。

2 Deployment または PodSpec が組み込まれた同様のリソースを参照するサンプルアプリケーション。

上記の例で示されるように、**ConfigMap** または **Secret** 自体を、バインディングデータのソースとして使用されるサービスリソースとして直接使用することもできます。

### 6.8.1. 命名ストラテジー

命名ストラテジーは、**binding.operators.coreos.com** API グループでのみ利用できます。

命名ストラテジーは Go テンプレートを使用して、サービスバインディングリクエストでカスタムバインディング名を定義するのに役立ちます。命名ストラテジーは、**ServiceBinding** カスタムリソース (CR) のマッピングを含むすべての属性に適用されます。

バッキングサービスは、バインディング名をファイルまたは環境変数としてワークロードに反映します。ワークロードが特定の形式で反映されるバインディング名を要求し、バッキングサービスから反映されるバインディング名がその形式で利用できない場合、命名ストラテジーを使用してバインディング名を変更できます。

#### 定義済みの後処理関数

命名ストラテジーを使用する一方、ワークロードの要求や要件によっては、任意の組み合わせで以下の定義済みの後処理関数を使用して、文字列を変換できます。

- **upper**: 文字列を大文字に変換します。
- **lower**: 文字列を小文字に変換します。
- **title**: 特定の一部の語句を除いて、各語句の最初の文字が大文字になるように文字列を変換します。

#### 事前に定義された命名ストラテジー

アノテーションで宣言されたバインディング名は、以下の事前に定義された命名ストラテジーに従って、ワークロードへの反映前に名前の変更に対して処理されます。

- **none**: これが適用されると、バインディング名は変更されません。

  **例**

  テンプレートのコンパイル後、バインディング名は **{{ .name }}** の形式を取ります。

  ```
  host: hippo-pgbouncer
  port: 5432
  ```

- **upper**: **namingStrategy** が定義されていない場合に適用されます。これが適用されると、バインディング名キーのすべての文字列を大文字に変換します。

  **例**

  テンプレートのコンパイル後、バインディング名は **{{ .service.kind | upper}}_{{ .name | upper }}** の形式を取ります。

  ```
  DATABASE_HOST: hippo-pgbouncer
  DATABASE_PORT: 5432
  ```

  ワークロードが別の形式を要求する場合は、カスタム命名ストラテジーを定義し、接頭辞とセパレーターを使用してバインディング名を変更できます (例:**PORT_DATABASE**)。

**注記**

- バインディング名がファイルとして反映される場合、デフォルトでは、事前定義された**none**命名ストラテジーが適用され、バインディング名は変更されません。
- バインディング名が環境変数として反映され、**namingStrategy** が定義されていない場合には、デフォルトでは事前定義された **uppercase** 命名ストラテジーが適用されます。
- カスタムバインディング名と事前定義済みの後処理関数の別の組み合わせを使用して、カスタム命名ストラテジーを定義することで、事前に定義された命名ストラテジーを上書きできます。

### 6.8.2. 高度なバインディングオプション

**ServiceBinding** カスタムリソース (CR) を定義して、次の高度なバインディングオプションを使用できます。

- バインディング名の変更: このオプションは、**binding.operators.coreos.com** API グループでのみ使用できます。
- カスタムバインディングデータの作成: このオプションは、**binding.operators.coreos.com** API グループでのみ使用できます。
- ラベルセレクターを使用したワークロードのバインド: このオプションは、**binding.operators.coreos.com** および **servicebinding.io** API グループの両方で使用できます。

#### 6.8.2.1. ワークロードへの反映前のバインディング名の変更

**ServiceBinding** CR の **.spec.namingStrategy** 属性で、バインディング名を変更するルールを指定できます。たとえば、PostgreSQL データベースに接続する Spring PetClinic サンプルアプリケーションについて考えてみましょう。この場合、PostgreSQL データベースサービスは、バインディングに使用するデータベースの **host** および **port** フィールドを公開します。Spring PetClinic サンプルアプリケーションは、バインディング名を使用してこの公開されたバインディングデータにアクセスできます。

**例:ServiceBinding CR の Spring PetClinic サンプルアプリケーション**

```
# ...
  application:
    name: spring-petclinic
    group: apps
    version: v1
    resource: deployments
# ...
```

**例:ServiceBinding CR の PostgreSQL データベースサービス**

```
# ...
  services:
  - group: postgres-operator.crunchydata.com
    version: v1beta1
```

```
    kind: PostgresCluster
    name: hippo
# ...
```

**namingStrategy** が定義されておらず、バインディング名が環境変数として反映される場合、バッキングサービスの **host: hippo-pgbouncer** 値および反映される環境変数は以下の例のように表示されます。

**例**

```
DATABASE_HOST: hippo-pgbouncer
```

ここでは、以下のようになります。

| | |
|---|---|
| **DATABASE** | **kind** バックエンドサービスを指定します。 |
| **HOST** | バインディング名を指定します。 |

**POSTGRESQL_{{ .service.kind | upper }}_{{ .name | upper }}_ENV** 命名ストラテジーを適用すると、サービスバインディングリクエストで準備したカスタムバインディング名の一覧が以下の例のように表示されます。

**例**

```
POSTGRESQL_DATABASE_HOST_ENV: hippo-pgbouncer
POSTGRESQL_DATABASE_PORT_ENV: 5432
```

以下の項目は、**POSTGRESQL_{{ .service.kind | upper }}_{{ .name | upper }}_ENV** 命名ストラテジーで定義される表現について説明しています。

- **.name**: バッキングサービスが公開するバインディング名を参照します。上記の例では、バインディング名は **HOST** および **PORT** です。
- **.service.kind**: バインディング名が命名ストラテジーで変更されるサービスリソースの種類を参照します。
- **upper**: Go テンプレート文字列をコンパイルする際に文字列を後処理するために使用する文字列関数。
- **POSTGRESQL**: カスタムバインディング名の接頭辞。
- **ENV**: カスタムバインディング名の接尾辞。

前述の例と同様に、**namingStrategy** で文字列テンプレートを定義し、バインディング名のそれぞれのキーがサービスバインディングリクエストによってどのように準備されるかを定義できます。

#### 6.8.2.2. カスタムバインディングデータの作成

アプリケーション開発者は、以下の状況でカスタムバインディングデータを作成できます。

- バッキングサービスがバインディングデータを公開しない。

- 公開される値が、ワークロードによって要求される形式では利用できません。

たとえば、バッキングサービス CR がホスト、ポート、およびデータベースユーザーをバインディングデータとして公開するが、ワークロードはバインディングデータを接続文字列として使用することを要求するケースを考えてみます。バッキングサービスを表す Kubernetes リソースの属性を使用して、カスタムバインディングデータを作成できます。

**例**

```
apiVersion: binding.operators.coreos.com/v1alpha1
kind: ServiceBinding
metadata:
   name: spring-petclinic-pgcluster
   namespace: my-petclinic
spec:
   services:
   - group: postgres-operator.crunchydata.com
     version: v1beta1
     kind: PostgresCluster
     name: hippo (1)
     id: postgresDB (2)
   - group: ""
     version: v1
     kind: Secret
     name: hippo-pguser-hippo
     id: postgresSecret
   application:
     name: spring-petclinic
     group: apps
     version: v1
     resource: deployments
   mappings:
     ## From the database service
     - name: JDBC_URL
       value: 'jdbc:postgresql://{{ .postgresDB.metadata.annotations.proxy }}:{{ .postgresDB.spec.port }}/{{ .postgresDB.metadata.name }}'
     ## From both the services!
     - name: CREDENTIALS
       value: '{{ .postgresDB.metadata.name }}{{ translationService.postgresSecret.data.password }}'
     ## Generate JSON
     - name: DB_JSON (3)
       value: {{ json .postgresDB.status }} (4)
```

(1) バッキングサービスリソースの名前。

(2) オプションの識別子。

(3) Service Binding Operator が生成する JSON 名。Service Binding Operator は、この JSON 名をファイルまたは環境変数の名前として投影します。

(4) Service Binding Operator が生成する JSON 値。Service Binding Operator は、この JSON 値をファイルまたは環境変数として投影します。JSON 値には、バッキングサービスカスタムリソースの指定したフィールドの属性が含まれます。

#### 6.8.2.3. ラベルセレクターを使用したワークロードのバインド

ラベルセレクターを使用して、バインドするワークロードを指定できます。ラベルセレクターを使用してワークロードを取得するサービスバインディングを宣言すると、Service Binding Operator は、指定されたラベルセレクターに一致する新しいワークロードを定期的に見つけてバインドしようとします。

たとえば、クラスター管理者は、**ServiceBinding** CR で適切な **labelSelector** フィールドを設定することにより、**environment: production** ラベルを持つ namespace 内のすべての **Deployment** にサービスをバインドできます。これにより、Service Binding Operator はこれらの各ワークロードを 1 つの **ServiceBinding** CR にバインドできます。

**binding.operators.coreos.com/v1alpha1 API の ServiceBinding CR の例**

```
apiVersion: binding.operators.coreos.com/v1alpha1
kind: ServiceBinding
metadata:
  name: multi-application-binding
  namespace: service-binding-demo
spec:
  application:
    labelSelector: 1
      matchLabels:
        environment: production
    group: apps
    version: v1
    resource: deployments
  services:
    group: ""
    version: v1
    kind: Secret
    name: super-secret-data
```

1 バインドされるワークロードを指定します。

**servicebinding.io API の ServiceBinding CR の例**

```
apiVersion: servicebindings.io/v1beta1
kind: ServiceBinding
metadata:
  name: multi-application-binding
  namespace: service-binding-demo
spec:
  workload:
    selector: 1
      matchLabels:
        environment: production
    apiVersion: app/v1
    kind: Deployment
  service:
    apiVersion: v1
    kind: Secret
    name: super-secret-data
```

1 バインドされるワークロードを指定します。

**重要**

次のフィールドのペアを定義すると、Service Binding Operator はバインディング操作を拒否し、エラーを生成します。

- **binding.operators.coreos.com/v1alpha1** API の **name** フィールドと **labelSelector** フィールド。
- **servicebinding.io** API (Spec API) の **name** フィールドと **selector** フィールド。

**再バインドの動作を理解する**

バインドが成功した後、**name** フィールドを使用してワークロードを識別する場合を考えてみましょう。そのワークロードを削除して再作成すると、**ServiceBinding** リコンサイラーはワークロードを再バインドせず、Operator はバインディングデータをワークロードに投影できません。ただし、**labelSelector** フィールドを使用してワークロードを識別する場合、**ServiceBinding** リコンサイラーはワークロードを再バインドし、Operator はバインディングデータを反映します。

### 6.8.3. PodSpec に準拠していないセカンダリーワークロードのバインド

サービスバインディングの一般的なシナリオでは、バッキングサービス、ワークロード (デプロイメント)、およびサービスバインディング Operator を設定する必要があります。PodSpec に準拠しておらず、プライマリーワークロード (デプロイメント) とサービスバインディング Operator の間にあるセカンダリーワークロード (アプリケーション Operator の場合もあります) が関与するシナリオについて考えてみます。

このようなセカンダリーワークロードリソースの場合、コンテナーパスのロケーションは任意です。サービスバインディングの場合、CR のセカンダリーワークロードが PodSpec に準拠していない場合、コンテナーパスのロケーションを指定する必要があります。これにより、バインディングデータが**ServiceBinding** カスタムリソース (CR) のセカンダリーワークロードで指定されたコンテナーパスに反映されます (たとえば、Pod 内にバインディングデータを配置したくない場合)。

Service Binding Operator では、コンテナーまたはシークレットがワークロード内に存在する場所のパスを設定し、これらのパスをカスタムの場所にバインドできます。

#### 6.8.3.1. コンテナーパスのカスタムロケーションの設定

Service Binding Operator がバインディングデータを環境変数として投影する場合、このカスタムの場所は **binding.operators.coreos.com** API グループで使用できます。

PodSpec に準拠しておらず、**spec.containers** パスに置かれているコンテナーを持つセカンダリーワークロード CR について考えてみます。

**例: セカンダリーワークロード CR**

```
apiVersion: "operator.sbo.com/v1"
kind: SecondaryWorkload
metadata:
  name: secondary-workload
spec:
  containers:
  - name: hello-world
    image: quay.io/baijum/secondary-workload:latest
    ports:
    - containerPort: 8080
```

**手順**

- **ServiceBinding** CR で値を指定して **spec.containers** パスを設定し、このパスを **spec.application.bindingPath.containersPath** カスタムロケーションにバインドします。

  **例:ServiceBinding CR とカスタムロケーションの spec.containers パス**

```
apiVersion: binding.operators.coreos.com/v1alpha1
kind: ServiceBinding
metadata:
   name: spring-petclinic-pgcluster
spec:
   services:
   - group: postgres-operator.crunchydata.com
     version: v1beta1
     kind: PostgresCluster
     name: hippo
     id: postgresDB
   - group: ""
     version: v1
     kind: Secret
     name: hippo-pguser-hippo
     id: postgresSecret
   application: 1
     name: spring-petclinic
     group: apps
     version: v1
     resource: deployments
   application: 2
     name: secondary-workload
     group: operator.sbo.com
     version: v1
     resource: secondaryworkloads
     bindingPath:
       containersPath: spec.containers 3
```

1. Deployment または PodSpec が組み込まれた同様のリソースを参照するサンプルアプリケーション。
2. PodSpec に準拠していないセカンダリーワークロード。
3. コンテナーパスのカスタムロケーション。

コンテナーパスのロケーションを指定した後に、サービスバインディング Operator はバインディングデータを生成します。これは、**ServiceBinding** CR のセカンダリーワークロードで指定されるコンテナーパスで利用できます。

以下の例は、 **envFrom**フィールドと**secretRef**フィールドを持つ**spec.containers**パスを示しています。

**例:envFrom および secretRef フィールドのあるセカンダリーワークロード CR**

```
apiVersion: "operator.sbo.com/v1"
kind: SecondaryWorkload
```

```
metadata:
   name: secondary-workload
spec:
   containers:
   - env: (1)
     - name: ServiceBindingOperatorChangeTriggerEnvVar
       value: "31793"
     envFrom:
     - secretRef:
         name: secret-resource-name (2)
     image: quay.io/baijum/secondary-workload:latest
     name: hello-world
     ports:
     - containerPort: 8080
     resources: {}
```

(1) サービスバインディング Operator で生成される値を持つコンテナーの一意の配列。これらの値はバッキングサービス CR に基づいています。

(2) サービスバインディング Operator によって生成される **Secret** リソースの名前。

#### 6.8.3.2. シークレットパスのカスタムロケーションの設定

Service Binding Operator がバインディングデータを環境変数として投影する場合、このカスタムの場所は **binding.operators.coreos.com** API グループで使用できます。

PodSpec に準拠しておらず、**spec.secret** パスに置かれているシークレットのみを持つセカンダリーワークロード CR を考えてみます。

**例: セカンダリーワークロード CR**

```
apiVersion: "operator.sbo.com/v1"
kind: SecondaryWorkload
metadata:
   name: secondary-workload
spec:
   secret: ""
```

**手順**

- **ServiceBinding** CR で値を指定して **spec.secret** パスを設定し、このパスを **spec.application.bindingPath.secretPath** カスタムロケーションにバインドします。

  **例:ServiceBinding CR とカスタムロケーションの spec.secret パス**

  ```
  apiVersion: binding.operators.coreos.com/v1alpha1
  kind: ServiceBinding
  metadata:
     name: spring-petclinic-pgcluster
  spec:
  ...
     application: (1)
       name: secondary-workload
  ```

```
      group: operator.sbo.com
      version: v1
      resource: secondaryworkloads
      bindingPath:
        secretPath: spec.secret (2)
...
```

(1) PodSpec に準拠していないセカンダリーワークロード。

(2) **Secret** リソースの名前が含まれるシークレットパスのカスタムロケーション。

シークレットパスのロケーションを指定した後に、サービスバインディング Operator はバインディングデータを生成します。これは、**ServiceBinding** CR のセカンダリーワークロードで指定されるシークレットパスで利用できます。

以下の例は、**binding-request** 値による **spec.secret** パスを示しています。

**例:binding-request 値が設定されたセカンダリーワークロード CR**

```
...
apiVersion: "operator.sbo.com/v1"
kind: SecondaryWorkload
metadata:
   name: secondary-workload
spec:
   secret: binding-request-72ddc0c540ab3a290e138726940591debf14c581 (1)
...
```

(1) Service Binding Operator が生成する **Secret** リソースの一意の名前。

#### 6.8.3.3. ワークロードリソースマッピング

**注記**

- ワークロードリソースマッピングは、両方の API グループ (**binding.operators.coreos.com** および **servicebinding.io**) の **ServiceBinding** カスタムリソース (CR) のセカンダリーワークロードで使用できます。
- **servicebinding.io** API グループの下での み、**ClusterWorkloadResourceMapping** リソースを定義する必要があります。ただし、**ClusterWorkloadResourceMapping** リソース は、**binding.operators.coreos.com** および **servicebinding.io** の両方の API グループで **ServiceBinding** リソースと対話します。

コンテナーパスの設定方法を使用してカスタムパスの場所を設定できない場合は、バインディングデータを投影する必要がある場所を正確に定義できます。**servicebinding.io** API グループで **ClusterWorkloadResourceMapping** リソースを定義して、特定のワークロードの種類のバインディングデータを投影する場所を指定します。

次の例は、**CronJob.batch/v1** リソースのマッピングを定義する方法を示しています。

**例: CronJob.batch/v1 リソースのマッピング**

```
apiVersion: servicebinding.io/v1beta1
kind: ClusterWorkloadResourceMapping
metadata:
 name: cronjobs.batch (1)
spec:
  versions:
  - version: "v1" (2)
    annotations: .spec.jobTemplate.spec.template.metadata.annotations (3)
    containers:
    - path: .spec.jobTemplate.spec.template.spec.containers[*] (4)
    - path: .spec.jobTemplate.spec.template.spec.initContainers[*]
      name: .name (5)
      env: .env (6)
      volumeMounts: .volumeMounts (7)
    volumes: .spec.jobTemplate.spec.template.spec.volumes (8)
```

(1) **ClusterWorkloadResourceMapping** リソースの名前。マップされたワークロードリソースの **plural.group** として修飾する必要があります。

(2) マップされているリソースのバージョン。指定されていないバージョンは、*ワイルドカードと一致させることができます。

(3) オプション: Pod 内の **.annotations** フィールドの識別子。固定 JSONPath で指定されます。デフォルト値は **.spec.template.spec.annotations** です。

(4) JSONPath で指定された、Pod 内の **.containers** および **.initContainers** フィールドの識別子。**containers** フィールドの下にエントリーが定義されていない場合、Service Binding Operator のデフォルトは **.spec.template.spec.containers[*]** および **.spec.template.spec.initContainers[*]** の 2 つのパスになり、他のすべてのフィールドはデフォルトとして次のように設定されます。ただし、エントリーを指定する場合は、**.path** フィールドを定義する必要があります。

(5) オプション: コンテナー内の **.name** フィールドの識別子。固定 JSONPath で指定されます。デフォルト値は **.name** です。

(6) オプション: コンテナー内の **.env** フィールドの識別子。固定 JSONPath で指定されます。デフォルト値は **.env** です。

(7) オプション: コンテナー内の **.volumeMounts** フィールドの識別子。固定 JSONPath で指定されます。デフォルト値は **.volumeMounts** です。

(8) オプション: Pod 内の **.volumes** フィールドの識別子。固定 JSONPath で指定されます。デフォルト値は **.spec.template.spec.volumes** です。

**重要**

- このコンテキストでは、固定 JSONPath は、次の操作のみを受け入れる JSONPath 文法のサブセットです。
  - フィールド検索: **.spec.template**
  - 配列のインデックス: **.spec['template']**

  その他の操作は受け付けません。
- これらのフィールドのほとんどはオプションです。指定されていない場合、Service Binding Operator は **PodSpec** リソースと互換性のあるデフォルトを想定します。
- Service Binding Operator では、これらの各フィールドが Pod デプロイメントの対応するフィールドと構造的に同等である必要があります。たとえば、ワークロードリソースの **.env** フィールドの内容は、Pod リソースの **.env** フィールドが受け入れるのと同じデータ構造を受け入れる必要があります。それができない場合、そのようなワークロードにバインディングデータを投影すると、Service Binding Operator で予期しない動作が発生する可能性があります。

**binding.operators.coreos.com API グループに固有の動作**

**ClusterWorkloadResourceMapping** リソースが **binding.operators.coreos.com** API グループの下の **ServiceBinding** リソースと対話する場合、次の動作が予想されます。

- **bindAsFiles: false** フラグ値を持つ **ServiceBinding** リソースがこれらのマッピングのいずれかと一緒に作成される場合、環境変数は、対応する **ClusterWorkloadResourceMapping** リソースで指定された各 **path** フィールドの下の **.envFrom** フィールドに投影されます。
- クラスター管理者は、バインド目的で **ServiceBinding.bindings.coreos.com** リソースの **ClusterWorkloadResourceMapping** リソースと **.spec.application.bindingPath.containersPath** フィールドの両方を指定できます。Service Binding Operator は、**ClusterWorkloadResourceMapping** リソースと **.spec.application.bindingPath.containersPath** フィールドの両方で指定された場所にバインディングデータを投影しようとします。この動作は、**path: $containersPath** 属性を持つ対応する **ClusterWorkloadResourceMapping** リソースにコンテナーエントリーを追加することと同じです。他のすべての値はデフォルト値を取ります。

### 6.8.4. バッキングサービスからのワークロードのバインド解除

**oc** ツールを使用して、バッキングサービスからワークロードのバインドを解除できます。

- バッキングサービスからワークロードのバインドを解除するには、これにリンクされている **ServiceBinding** カスタムリソース (CR) を削除します。

```
$ oc delete ServiceBinding <.metadata.name>
```

**例**

```
$ oc delete ServiceBinding spring-petclinic-pgcluster
```

ここでは、以下のようになります。

| | |
|---|---|
| **spring-petclinic-pgcluster** | **ServiceBinding** CR の名前を指定します。 |

### 6.8.5. 関連情報

- ワークロードをバッキングサービスとバインドする
- Spring PetClinic サンプルアプリケーションを PostgreSQL データベースサービスに接続します。
- ファイルからのカスタムリソースの作成
- ClusterWorkloadResourceMapping リソースのサンプルスキーマ。

## 6.9. 開発者パースペクティブを使用したアプリケーションのサービスへの接続

**Topology** ビューは、次の目的で使用します。

- アプリケーション内での複数コンポーネントのグループ化
- コンポーネントを相互に接続します。
- ラベルを使用して複数のリソースをサービスに接続します。

バインディングまたはビジュアルコネクターを使用して、コンポーネントを接続できます。

コンポーネント間のバインディング接続は、ターゲットノードが Operator がサポートするサービスである場合にのみ確立できます。これは、矢印をこのようなターゲットノードにドラッグする際に表示される **Create a binding connector** ツールチップによって示されます。アプリケーションがバインディングコネクターを使用してサービスに接続されると、**ServiceBinding** が作成されます。その後、サービスバインディング Operator コントローラーは必要なバインディングデータをアプリケーションデプロイメントにプロジェクションします。要求が正常に行われると、アプリケーションが再デプロイされ、接続されたコンポーネント間の対話が確立されます。

ビジュアルコネクターは、接続先となるコンポーネント間の視覚的な接続のみを表示します。コンポーネント間の対話は確立されません。ターゲットノードが Operator がサポートするサービスではない場合、**Create a visual connector** ツールチップは矢印をターゲットノードにドラッグすると表示されます。

### 6.9.1. Operator が支援するバインド可能なサービスの検出と識別

ユーザーは、バインド可能なサービスを作成する場合は、どのサービスがバインド可能かを知っている必要があります。バインド可能なサービスは、クレデンシャル、接続の詳細、ボリュームマウント、シークレット、およびその他のバインドデータなどのバインドデータを標準的な方法で公開するため、アプリケーションが簡単に使用できるサービスです。**Developer** パースペクティブは、そのようなバインド可能なサービスを発見して識別するのに役立ちます。

**手順**

- Operator が支援するバインド可能なサービスを検出して識別する場合、次の代替アプローチを検討してください。

- **+Add → Developer Catalog → Operator Backed** をクリックして、Operator-backed タイルを表示します。サービスバインディング機能をサポートする Operator が支援するサービスの場合、タイルに **Bindable** バッジが表示されます。
- **Operator Backed** ページの左側のペインで、**Bindable** を選択します。

  **ヒント**

  **Service binding** の横にあるヘルプアイコンをクリックして、バインド可能なサービスの詳細を表示します。

- **+Add → Add** をクリックして、Operator が支援するサービスを検索します。バインド可能なサービスをクリックすると、サイドパネルに **Bindable** バッジが表示されます。

### 6.9.2. コンポーネント間のビジュアル接続の作成

ビジュアルコネクターを使用してアプリケーションコンポーネントに接続する意図を示すことができます。

この手順では、PostgreSQL データベースサービスと Spring PetClinic のサンプルアプリケーション間の視覚的な接続の作成例を説明します。

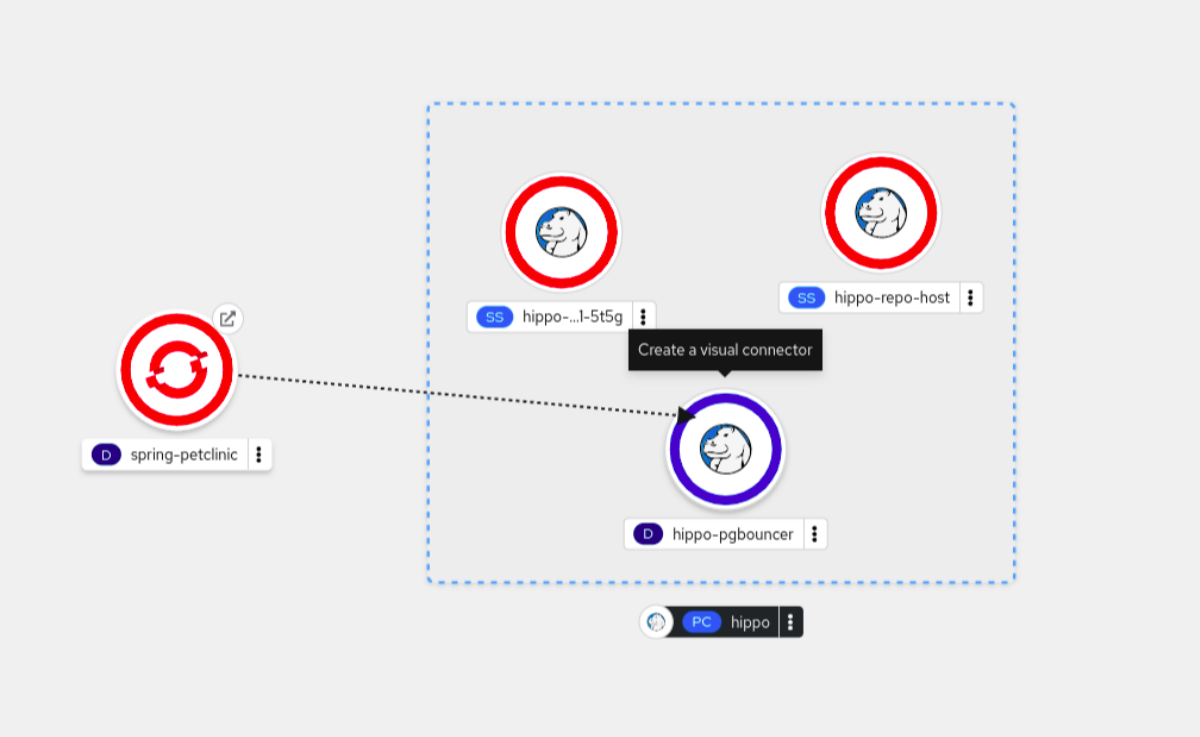
**前提条件**

- **Developer** パースペクティブを使用して Spring PetClinic のサンプルアプリケーションを作成し、デプロイしている。
- **Developer** パースペクティブを使用して Crunchy PostgreSQL データベースインスタンスを作成し、デプロイしている。このインスタンスには、**hippo-backup**、**hippo-instance**、**hippo-repo-host**、**hippo-pgbouncer** の 4 つのコンポーネントがあります。

**手順**

1. **Developer** パースペクティブで、関連するプロジェクト (**my-petclinic** など) に切り替えます。
2. Spring PetClinic サンプルアプリケーションにカーソルを合わせ、ノード上の矢印を確認します。

図6.2 ビジュアルコネクター

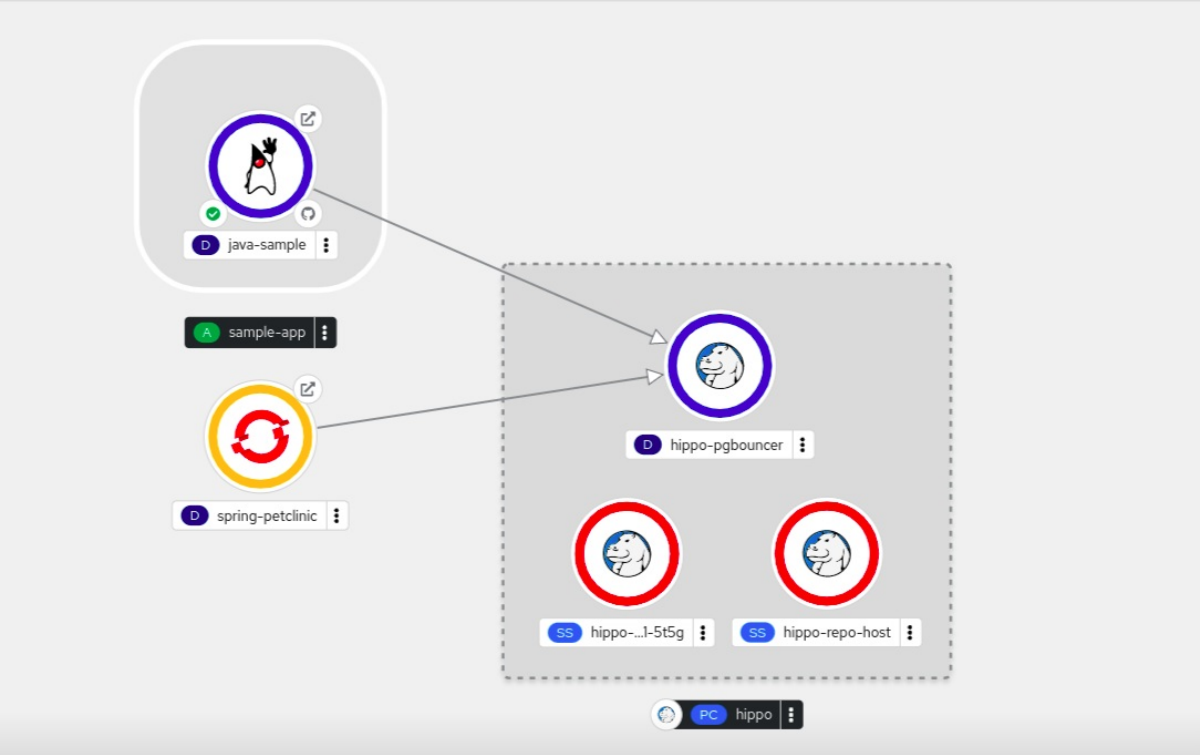



3. 矢印をクリックして **hippo-pgbouncer** デプロイメントに向かってドラッグし、Spring PetClinic サンプルアプリケーションを接続します。

4. **spring-petclinic** デプロイメントをクリックし、**Overview** パネルを表示します。**Details** タブで Annotations セクションの編集アイコンをクリックして、**Key = app.openshift.io/connects-to** と Value = **[{"apiVersion":"apps/v1","kind":"Deployment","name":"hippo-pgbouncer"}]** アノテーションがデプロイメントに追加されていることを確認します。

5. オプション: これらの手順を繰り返して、作成した他のアプリケーションとコンポーネントの間に視覚的な接続を確立できます。

図6.3 複数アプリケーションへの接続




## 6.9.3. コンポーネント間のバインディング接続の作成

次の例に示すように、Operator がサポートするコンポーネントを使用してバインディング接続を作成できます。この例では、PostgreSQL データベースサービスと Spring PetClinic サンプルアプリケーションを使用します。PostgreSQL Database Operator がサポートするサービスとのバインディング接続を作成するには、最初に Red Hat 提供の PostgreSQL Database Operator を Operator に追加してから、**Operator** をインストールする必要があります。次に、PostreSQL Database Operator は、シークレット、設定マップ、ステータス、および仕様属性でバインディングデータを公開するデータベースリソースを作成および管理します。

**前提条件**

- **Developer** パースペクティブで Spring PetClinic サンプルアプリケーションを作成してデプロイしました。
- **OperatorHub** から Service Binding Operator をインストールしました。
- **v5 Update** チャネルの OperatorHub から **Crunchy Postgres for Kubernetes**Operator をインストールしました。
- **Developer** パースペクティブで **PostgresCluster** リソースを作成しました。これにより、**hippo-backup**、**hippo-instance**、**hippo-repo-host**、**hippo-pgbouncer** というコンポーネント を持つ Crunchy PostgreSQL データベースインスタンスが作成されました。

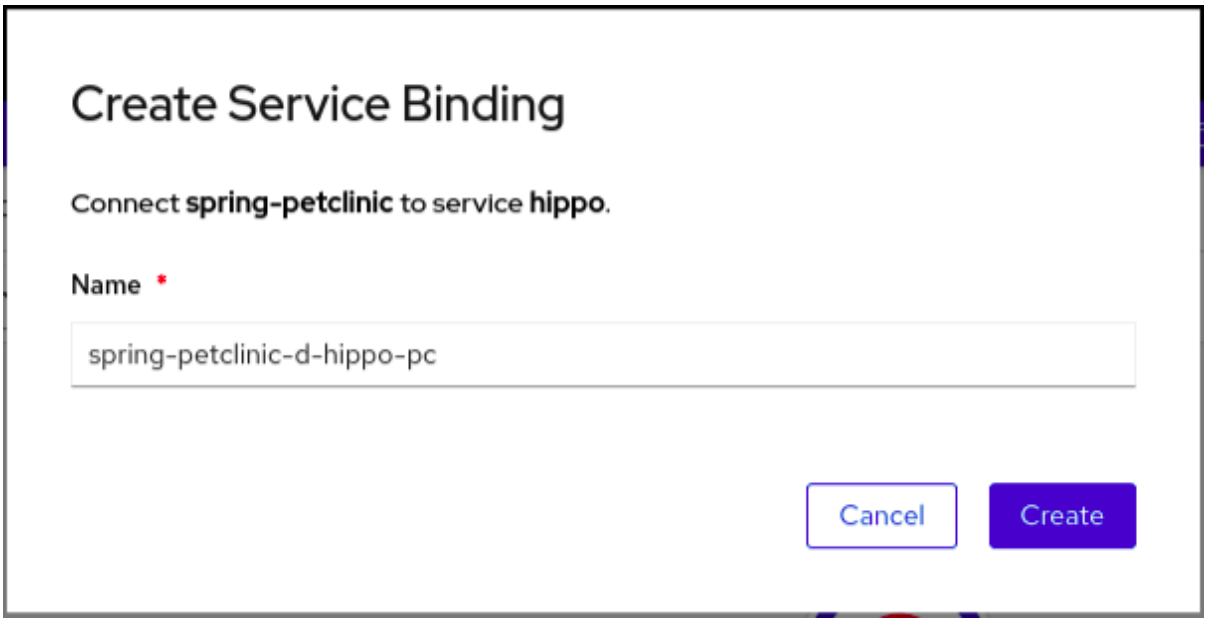
**手順**

1. **Developer** パースペクティブで、関連するプロジェクト (**my-petclinic** など) に切り替えます。
2. **Topology** ビューで、Spring PetClinic サンプルアプリケーションにカーソルを合わせてノードの矢印を確認します。

3. 矢印を Postgres クラスターの **hippo** データベースアイコンにドラッグアンドドロップして、Spring PetClinic サンプルアプリケーションとのバインディング接続を作成します。

4. **サービスバインドの作成** ダイアログで、サービスバインドのデフォルトの名前をそのまま使用するか、別の名前を追加して、**作成** をクリックします。

**図6.4 Service Binding ダイアログ**



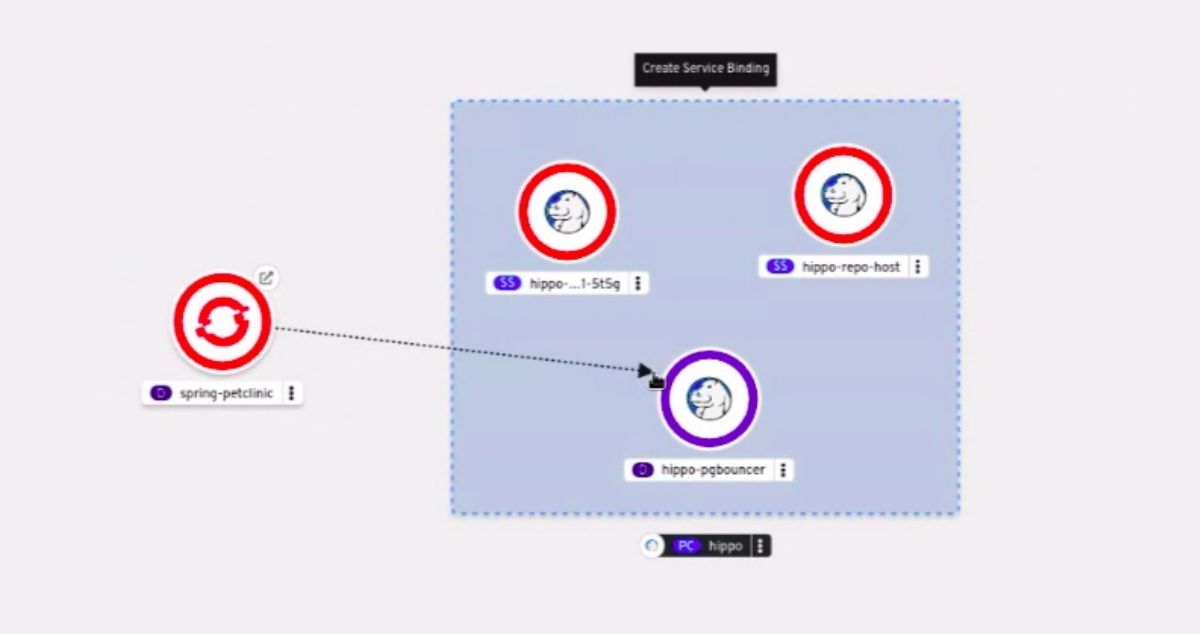

5. オプション: Topology ビューを使用してバインド接続を作成するのが難しい場合は、**+Add → YAML → Import YAML** に移動します。

6. オプション: YAML エディターで、**ServiceBinding** リソースを追加します。

```
apiVersion: binding.operators.coreos.com/v1alpha1
kind: ServiceBinding
metadata:
   name: spring-petclinic-pgcluster
   namespace: my-petclinic
spec:
   services:
   - group: postgres-operator.crunchydata.com
     version: v1beta1
     kind: PostgresCluster
     name: hippo
   application:
     name: spring-petclinic
     group: apps
     version: v1
     resource: deployments
```

サービスバインディングリクエストが作成され、**ServiceBinding** リソースを通じてバインディング接続が作成されます。データベースサービス接続要求が成功すると、アプリケーションが再デプロイされ、接続が確立されます。

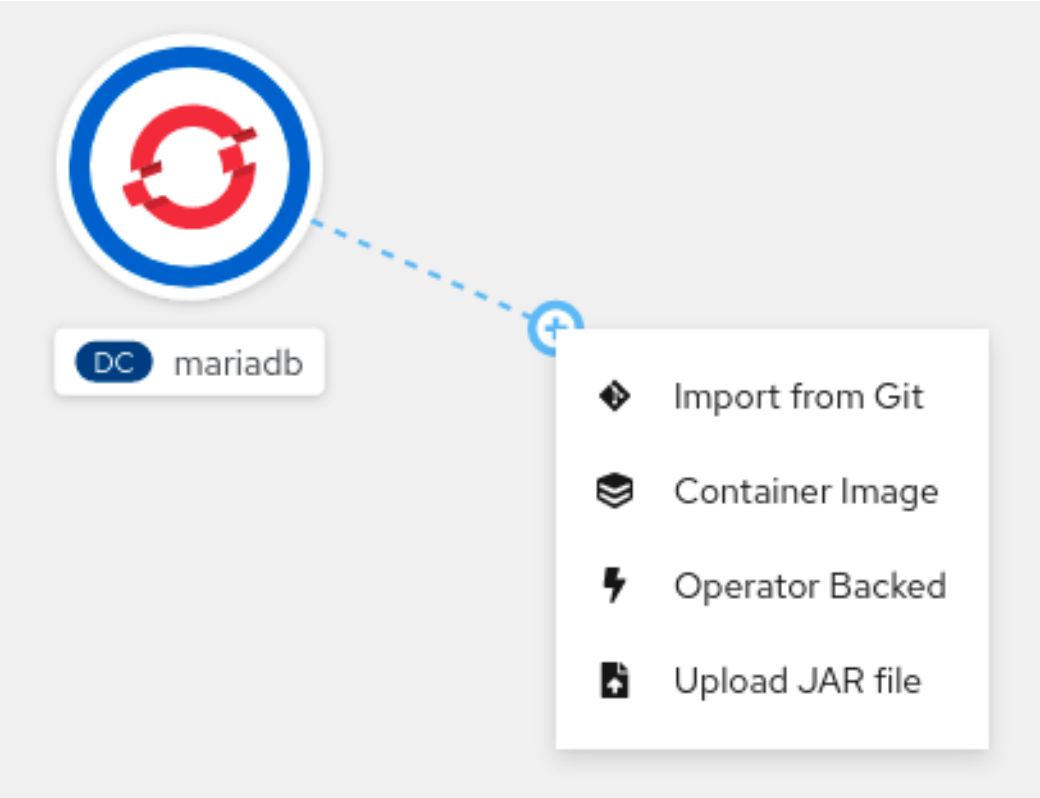
**図6.5 バインディングコネクター**




**ヒント**

矢印をドラッグしてコンテキストメニューを使用し、Operator がサポートするサービスへのバインディング接続を追加して作成できます。

**図6.6 バインディング接続を作成するためのコンテキストメニュー**




7. ナビゲーションメニューで、**トポロジー** をクリックします。トポロジービューの spring-petclinic 展開には、その Web ページを表示するための 開く URL リンクが含まれています。

8. **URL を開く** リンクをクリックします。

Spring PetClinic サンプルアプリケーションをリモートで表示して、アプリケーションがデータベースサービスに接続され、データが Crunchy PostgreSQL データベースサービスからアプリケーションに正常に投影されたことを確認できます。

Service Binding Operator は、アプリケーションとデータベースサービスの間に有効な接続を正常に作成しました。

### 6.9.4. Topology ビューからのサービスバインディングのステータス確認

**Developer** パースペクティブは、**Topology** ビューを通じてサービスバインディングのステータスを確認するのに役立ちます。

**手順**

1. サービスのバインドが成功したら、バインドコネクターをクリックします。サイドパネルが表示され、**Details** タブの下に **Connected** ステータスが表示されます。
   必要に応じて、次のページで **Developer** パースペクティブから **Connected** ステータスを表示できます。

   - **ServiceBindings** ページ。
   - **ServiceBinding details** ページ。さらに、ページタイトルには **Connected** バッジが表示されます。

2. サービスバインディングに失敗した場合、バインディングコネクターの接続の中央に赤い矢印と赤い十字が表示されます。このコネクターをクリックすると、サイドパネルの **Details** タブに **Error** ステータスが表示されます。必要に応じて、**Error** ステータスをクリックして、根本的な問題に関する特定の情報を表示します。
   次のページで、**Developer** パースペクティブから **Error** ステータスとツールチップを表示することもできます。

   - **ServiceBindings** ページ。
   - **ServiceBinding details** ページ。さらに、ページタイトルには **Error** バッジが表示されます。

**ヒント**

**ServiceBindings** ページで、**Filter** ドロップダウンを使用して、ステータスに基づいてサービスバインディングをリスト表示します。

### 6.9.5. 関連情報

- サービスバインディングの使用
- バインド可能な既知の Operator

# 第7章 HELM チャートの使用

## 7.1. HELM について

Helm は、アプリケーションやサービスの OpenShift Container Platform クラスターへのデプロイメントを単純化するソフトウェアパッケージマネージャーです。

Helm は **charts** というパッケージ形式を使用します。Helm チャートは、OpenShift Container Platform リソースを記述するファイルのコレクションです。

クラスター内のチャートの実行中のインスタンスは、**リリース** と呼ばれます。チャートがクラスターにインストールされているたびに、新規のリリースが作成されます。

チャートのインストール時、またはリリースがアップグレードまたはロールバックされるたびに、増分リビジョンが作成されます。

### 7.1.1. 主な特長

Helm は以下を行う機能を提供します。

- チャートリポジトリーに保存したチャートの大規模なコレクションの検索。
- 既存のチャートの変更。
- OpenShift Container Platform または Kubernetes リソースの使用による独自のチャートの作成。
- アプリケーションのチャートとしてのパッケージ化および共有。

### 7.1.2. OpenShift の Helm チャートの Red Hat 認定

Red Hat OpenShift Container Platform にデプロイする全コンポーネントに対して、Red Hat による Helm チャートの検証と認定を受けることができます。チャートは、自動化の Red Hat OpenShift 認定ワークフローを経て、セキュリティーコンプライアンスを確保し、プラットフォームとの統合とサービス全般が最適であることを保証します。認定はチャートの整合性を確保し、Helm チャートが Red Hat OpenShift クラスターでシームレスに機能することを確認します。

### 7.1.3. 関連情報

- Red Hat パートナーとしての Helm チャートの認定方法は、OpenShift の Helm チャートの Red Hat 認定 を参照してください。
- Red Hat パートナー向けの OpenShift および Container 認定に関する情報は、Partner Guide for OpenShift and Container Certification を参照してください。
- チャートのリストについては、Red Hat **Helm インデックス** ファイルを参照してください。
- Red Hat Marketplace で利用可能なチャートを確認できます。詳細は、Red Hat Marketplace の使用 を参照してください。

## 7.2. HELM のインストール

以下のセクションでは、CLI を使用して各種の異なるプラットフォームに Helm をインストールする方法を説明します。

また、OpenShift Container Platform Web コンソールから最新のバイナリーへの URL を見つけるには、右上隅の？アイコンをクリックし、**Command Line Tools** を選択します。

**前提条件**

- Go バージョン 1.13 以降がインストールされている。

### 7.2.1. Linux の場合

1. Helm バイナリーをダウンロードし、これをパスに追加します。
   - Linux (x86_64, amd64)

     ```
     # curl -L https://mirror.openshift.com/pub/openshift-v4/clients/helm/latest/helm-linux-amd64 -o /usr/local/bin/helm
     ```

   - Linux on IBM Z および IBM® LinuxONE (s390x)

     ```
     # curl -L https://mirror.openshift.com/pub/openshift-v4/clients/helm/latest/helm-linux-s390x -o /usr/local/bin/helm
     ```

   - Linux on IBM Power (ppc64le)

     ```
     # curl -L https://mirror.openshift.com/pub/openshift-v4/clients/helm/latest/helm-linux-ppc64le -o /usr/local/bin/helm
     ```

2. バイナリーファイルを実行可能にします。

   ```
   # chmod +x /usr/local/bin/helm
   ```

3. インストールされたバージョンを確認します。

   ```
   $ helm version
   ```

   **出力例**

   ```
   version.BuildInfo{Version:"v3.0",
   GitCommit:"b31719aab7963acf4887a1c1e6d5e53378e34d93", GitTreeState:"clean",
   GoVersion:"go1.13.4"}
   ```

### 7.2.2. Windows 7/8 の場合

1. 最新の **.exe** ファイル をダウンロードし、希望のディレクトリーに配置します。
2. **Start** を右クリックし、**Control Panel** をクリックします。
3. **System and Security** を選択してから **System** をクリックします。
4. 左側のメニューから、**Advanced systems settings** を選択し、下部にある **Environment Variables** をクリックします。
5. **Variable** セクションから **Path** を選択し、**Edit** をクリックします。

6. **New** をクリックして、 **.exe** ファイルのあるフォルダーへのパスをフィールドに入力するか、**Browse** をクリックし、ディレクトリーを選択して **OK** をクリックします。

### 7.2.3. Windows 10 の場合

1. 最新の **.exe** ファイル をダウンロードし、希望のディレクトリーに配置します。
2. **Search** クリックして、**env** または **environment** を入力します。
3. **Edit environment variables for your account**を選択します。
4. **Variable** セクションから **Path** を選択し、**Edit** をクリックします。
5. **New** をクリックし、exe ファイルのあるディレクトリーへのパスをフィールドに入力するか、**Browse** をクリックし、ディレクトリーを選択して **OK** をクリックします。

### 7.2.4. MacOS の場合

1. Helm バイナリーをダウンロードし、これをパスに追加します。

```
# curl -L https://mirror.openshift.com/pub/openshift-v4/clients/helm/latest/helm-darwin-amd64
-o /usr/local/bin/helm
```

2. バイナリーファイルを実行可能にします。

```
# chmod +x /usr/local/bin/helm
```

3. インストールされたバージョンを確認します。

```
$ helm version
```

**出力例**

```
version.BuildInfo{Version:"v3.0",
GitCommit:"b31719aab7963acf4887a1c1e6d5e53378e34d93", GitTreeState:"clean",
GoVersion:"go1.13.4"}
```

## 7.3. カスタム HELM チャートリポジトリーの設定

以下の方法のいずれかを使用して、OpenShift Container Platform クラスターに Helm チャートをインストールできます。

- CLI
- Web コンソールの **Developer** パースペクティブ。

Web コンソールの **Developer** パースペクティブの **Developer Catalog** には、クラスターで利用可能な Helm チャートが表示されます。デフォルトで、これは Red Hat Helm チャートリポジトリーの OpenShift Helm チャートのリストを表示します。チャートの一覧については、Red Hat **Helm インデックス ファイル**を参照してください。

クラスター管理者は、デフォルトのクラススコープの Helm リポジトリーとは別に、複数のクラススコープおよび namespace スコープの Helm チャートリポジトリーを追加し、**Developer Catalog** でこれらのリポジトリーから Helm チャートを表示できます。

適切なロールベースアクセス制御 (RBAC) パーミッションを持つ通常のユーザーまたはプロジェクトメンバーとして、デフォルトのクラススコープの Helm リポジトリーとは別に、複数の namespace スコープの Helm チャートリポジトリーを追加し、**Developer Catalog** でこれらのリポジトリーから Helm チャートを表示できます。

Web コンソールの **Developer** パースペクティブでは、**Helm** ページを使用して次のことができます。

- **作成** ボタンを使用して、Helm リリースとリポジトリーを作成します。
- クラススコープまたは namespace スコープの Helm チャートリポジトリーを作成、更新、または削除します。
- リポジトリータブで既存の Helm チャートリポジトリーのリストを表示します。これも、クラススコープまたは namespace スコープのいずれかとして簡単に区別できます。

### 7.3.1. OpenShift Container Platform クラスターでの Helm チャートのインストール

**前提条件**

- 実行中の OpenShift Container Platform クラスターがあり、ログインしている。
- Helm がインストールされている。

**手順**

1. 新規プロジェクトを作成します。

```
$ oc new-project vault
```

2. Helm チャートのリポジトリーをローカルの Helm クライアントに追加します。

```
$ helm repo add openshift-helm-charts https://charts.openshift.io/
```

**出力例**

```
"openshift-helm-charts" has been added to your repositories
```

3. リポジトリーを更新します。

```
$ helm repo update
```

4. サンプルの HashiCorp Vault をインストールします。

```
$ helm install example-vault openshift-helm-charts/hashicorp-vault
```

**出力例**

```
NAME: example-vault
LAST DEPLOYED: Fri Mar 11 12:02:12 2022
```

```
NAMESPACE: vault
STATUS: deployed
REVISION: 1
NOTES:
Thank you for installing HashiCorp Vault!
```

5. チャートが正常にインストールされたことを確認します。

```
$ helm list
```

**出力例**

```
NAME          NAMESPACE REVISION UPDATED                                  STATUS   CHART
APP VERSION
example-vault vault     1        2022-03-11 12:02:12.296226673 +0530 IST deployed vault-
0.19.0 1.9.2
```

### 7.3.2. 開発者パースペクティブを使用した Helm チャートのインストール

Web コンソールまたは CLI コンソールの **Developer** パースペクティブを使用して、**Developer Catalog** にリスト表示されている Helm チャートからチャートを選択し、インストールできます。Helm チャートをインストールして Helm リリースを作成し、Web コンソールの **Developer** パースペクティブに表示できます。

**前提条件**

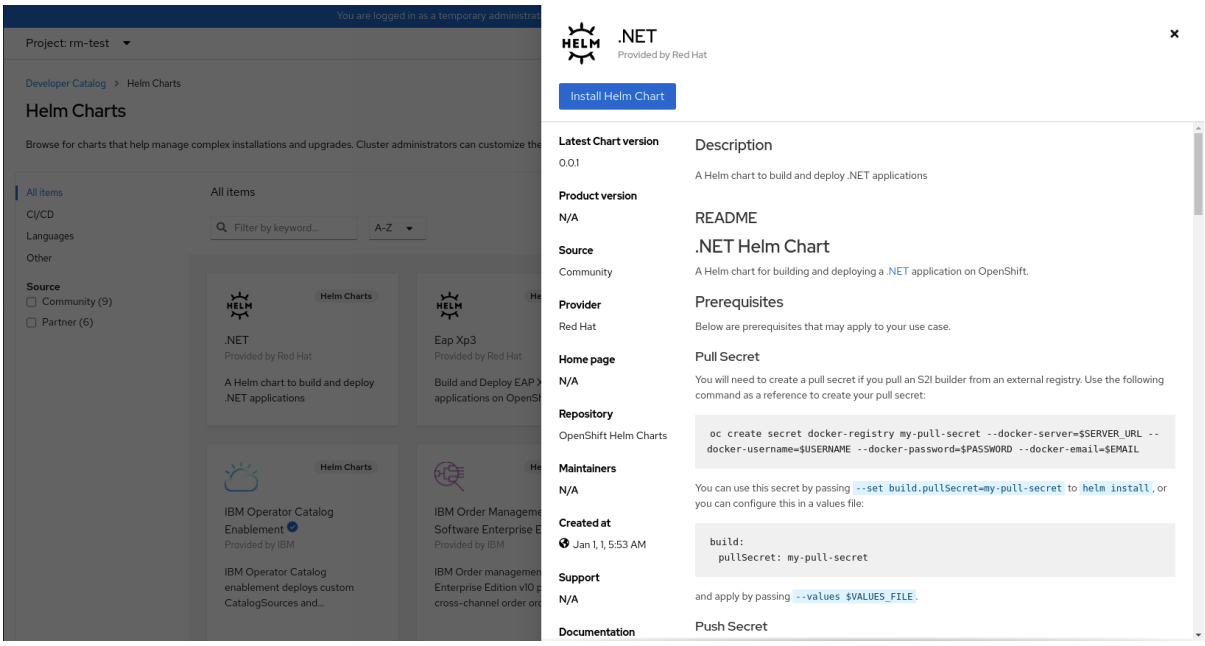
- Web コンソールにログイン し、**Developer パースペクティブ** に切り替えている。

**手順**

**Developer Catalog** で提供される Helm チャートから Helm リリースを作成するには、以下を実行します。

1. **Developer** パースペクティブで、**+Add** ビューに移動し、プロジェクトを選択します。次に、**Helm Chart** オプションをクリックし、**Developer Catalog** にすべての Helm チャートを表示します。
2. チャートを選択し、チャートの説明、README、チャートについてのその他の詳細を確認します。
3. **Install Helm Chart** をクリックします。

図7.1 Developer カタログの Helm チャート

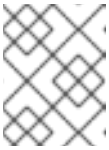



4. **Install Helm Chart** ページで以下を行います。

   a. リリースの固有の名前を **Release Name** フィールドに入力します。

   b. **Chart Version** ドロップダウンリストから必要なチャートのバージョンを選択します。

   c. **Form View** または **YAML View** を使用して Helm チャートを設定します。



**注記**

利用可能な場合は、**YAML View** と **Form View** 間で切り替えることができます。ビューの切り替え時に、データは永続化されます。

   d. **Install** をクリックして Helm リリースを作成します。リリースが表示される **Topology** ビューにリダイレクトされます。Helm チャートにリリースノートがある場合、チャートは事前に選択され、右側のパネルにそのリリースのリリースノートが表示されます。

   e. **Helm Releases** ページで、新しく作成された Helm リリースを表示します。

サイドパネルで **Actions** ボタンを使用するか、Helm リリースを右クリックして Helm リリースのアップグレード、ロールバック、またはアンインストールを実行できます。

### 7.3.3. Web 端末での Helm の使用

Web コンソールの **Developer** パースペクティブで Web ターミナルにアクセスする と、Helm を使用できます。

### 7.3.4. OpenShift Container Platform でのカスタム Helm チャートの作成

**手順**

1. 新規プロジェクトを作成します。

```
$ oc new-project nodejs-ex-k
```

2. OpenShift Container Platform オブジェクトが含まれる Node.js チャートのサンプルをダウンロードします。

```
$ git clone https://github.com/redhat-developer/redhat-helm-charts
```

3. サンプルチャートを含むディレクトリーに移動します。

```
$ cd redhat-helm-charts/alpha/nodejs-ex-k/
```

4. **Chart.yaml** ファイルを編集し、チャートの説明を追加します。

```
apiVersion: v2 (1)
name: nodejs-ex-k (2)
description: A Helm chart for OpenShift (3)
icon: https://static.redhat.com/libs/redhat/brand-assets/latest/corp/logo.svg (4)
version: 0.2.1 (5)
```

(1) チャート API バージョン。これは、Helm 3 以上を必要とする Helm チャートの場合は **v2** である必要があります。

(2) チャートの名前。

(3) チャートの説明。

(4) アイコンとして使用するイメージへの URL。

(5) Semantic Versioning (SemVer) 2.0.0 仕様に準拠したチャートのバージョン。

5. チャートが適切にフォーマットされていることを確認します。

```
$ helm lint
```

**出力例**

```
[INFO] Chart.yaml: icon is recommended

1 chart(s) linted, 0 chart(s) failed
```

6. 直前のディレクトリーレベルに移動します。

```
$ cd ..
```

7. チャートをインストールします。

```
$ helm install nodejs-chart nodejs-ex-k
```

8. チャートが正常にインストールされたことを確認します。

```
$ helm list
```

**出力例**

```
NAME NAMESPACE REVISION UPDATED STATUS CHART APP VERSION
nodejs-chart nodejs-ex-k 1 2019-12-05 15:06:51.379134163 -0500 EST deployed nodejs-
0.1.0  1.16.0
```

### 7.3.5. カスタム Helm チャートリポジトリーの追加

クラスター管理者は、カスタムの Helm チャートリポジトリーをクラスターに追加し、**Developer Catalog** のこれらのリポジトリーから Helm チャートへのアクセスを有効にできます。

**手順**

1. 新規の Helm Chart リポジトリーを追加するには、Helm Chart Repository カスタムリソース (CR) をクラスターに追加する必要があります。

   **Helm チャートリポジトリー CR のサンプル**

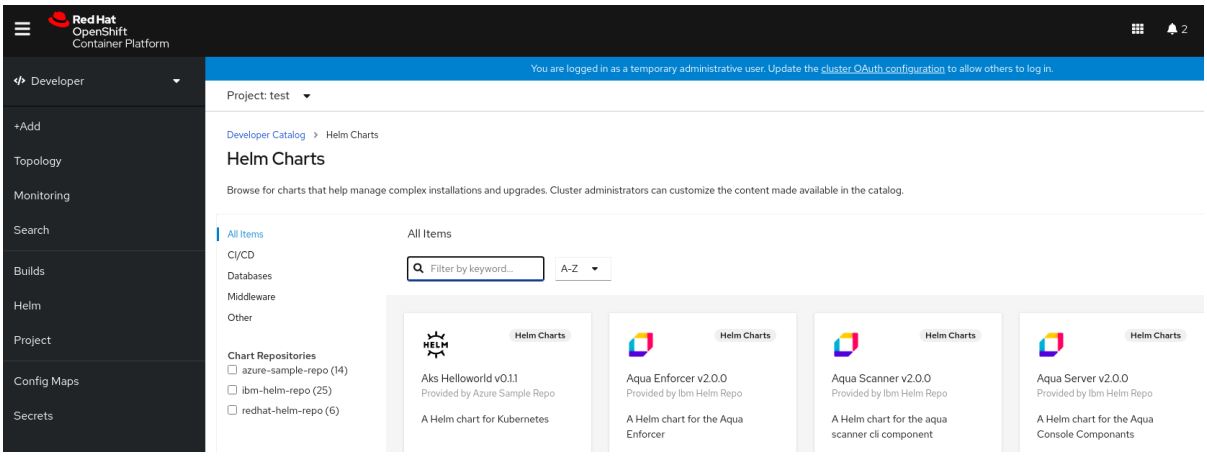
   ```
   apiVersion: helm.openshift.io/v1beta1
   kind: HelmChartRepository
   metadata:
     name: <name>
   spec:
    # optional name that might be used by console
    # name: <chart-display-name>
     connectionConfig:
       url: <helm-chart-repository-url>
   ```

   たとえば、Azure サンプルチャートリポジトリーを追加するには、以下を実行します。

   ```
   $ cat <<EOF | oc apply -f -
   apiVersion: helm.openshift.io/v1beta1
   kind: HelmChartRepository
   metadata:
     name: azure-sample-repo
   spec:
     name: azure-sample-repo
     connectionConfig:
       url: https://raw.githubusercontent.com/Azure-Samples/helm-charts/master/docs
   EOF
   ```

2. Web コンソールで **Developer Catalog** に移動し、チャートリポジトリーの Helm チャートが表示されることを確認します。
   たとえば、**Chart リポジトリー** フィルターを使用して、リポジトリーから Helm チャートを検索します。

図7.2 チャートリポジトリーのフィルター




**注記**

クラスター管理者がすべてのチャートリポジトリーを削除する場合は、**+Add** ビュー、**Developer Catalog**、および左側のナビゲーションパネルで Helm オプションを表示できません。

## 7.3.6. namespace スコープのカスタム Helm チャートリポジトリーの追加

Helm リポジトリーのクラススコープの **HelmChartRepository** カスタムリソース定義 (CRD) は、管理者が Helm リポジトリーをカスタムリソースとして追加できるようにします。namespace スコープの **ProjectHelmChartRepository** CRD により、適切なロールベースアクセス制御 (RBAC) パーミッションのあるプロジェクトメンバーは、任意の、ただし固有の namespace スコープの Helm リポジトリーリソースを作成できます。このようなプロジェクトメンバーは、クラススコープと namespace スコープ両方の Helm リポジトリーリソースからチャートを表示できます。

**注記**

- 管理者は、ユーザーが namespace スコープの Helm リポジトリーリソースを作成するのを制限できます。ユーザーを制限することで、管理者はクラスターロールではなく namespace ロールを使用して RBAC を柔軟に制御できます。これにより、ユーザーの不要なパーミッション昇格を回避し、承認されていないサービスまたはアプリケーションへのアクセスを防ぎます。
- namespace スコープの Helm リポジトリーを追加しても、既存のクラススコープの Helm リポジトリーの動作には影響を及ぼしません。

適切な RBAC パーミッションを持つ通常のユーザーまたはプロジェクトメンバーとして、カスタムの namespace スコープの Helm チャートリポジトリーをクラスターに追加し、**Developer Catalog** でこれらのリポジトリーから Helm チャートへのアクセスを有効にできます。

**手順**

1. 新規の namespace スコープの Helm Chart Repository を追加するには、Helm Chart Repository カスタムリソース (CR) を namespace に追加する必要があります。

**namespace スコープの Helm Chart Repository CR のサンプル**

```
apiVersion: helm.openshift.io/v1beta1
kind: ProjectHelmChartRepository
```

```
metadata:
  name: <name>
spec:
  url: https://my.chart-repo.org/stable

  # optional name that might be used by console
  name: <chart-repo-display-name>

  # optional and only needed for UI purposes
  description: <My private chart repo>

  # required: chart repository URL
  connectionConfig:
    url: <helm-chart-repository-url>
```

たとえば、**my-namespace** namespace スコープの Azure サンプルチャートリポジトリーを追加するには、以下を実行します。

```
$ cat <<EOF | oc apply --namespace my-namespace -f -
apiVersion: helm.openshift.io/v1beta1
kind: ProjectHelmChartRepository
metadata:
  name: azure-sample-repo
spec:
  name: azure-sample-repo
  connectionConfig:
    url: https://raw.githubusercontent.com/Azure-Samples/helm-charts/master/docs
EOF
```

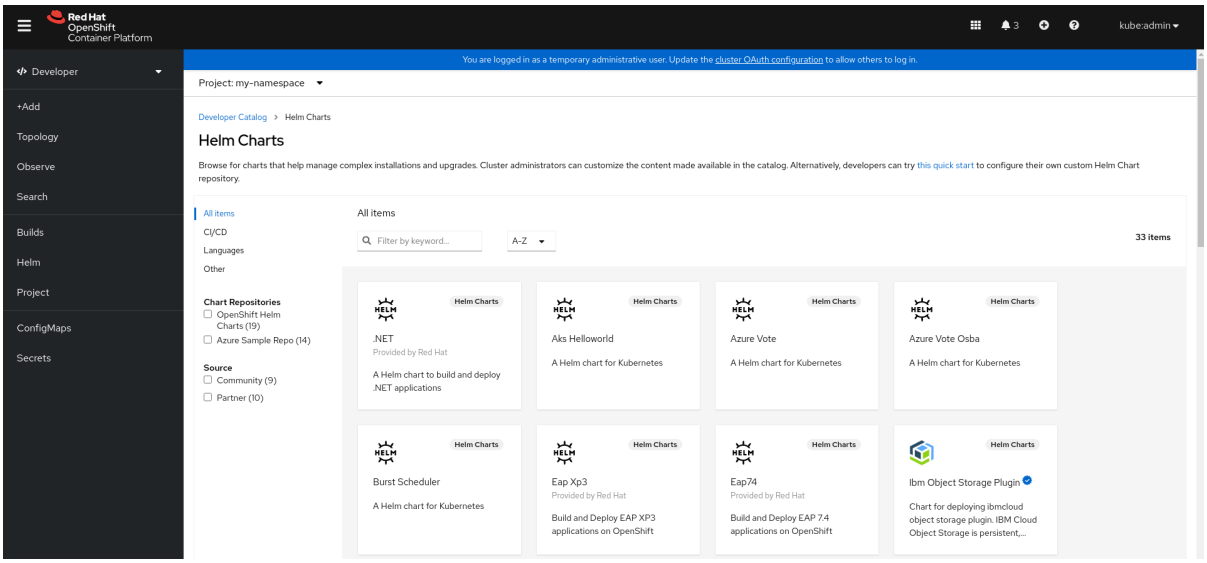
この出力から、namespace スコープの Helm Chart Repository CR が作成されていることが分かります。

**出力例**

```
projecthelmchartrepository.helm.openshift.io/azure-sample-repo created
```

2. Web コンソールで **Developer Catalog** に移動し、チャートリポジトリーの Helm チャートが **my-namespace** namespace に表示されることを確認します。
たとえば、**Chart リポジトリー** フィルターを使用して、リポジトリーから Helm チャートを検索します。

図7.3 namespace のチャートリポジトリーフィルター




あるいは、以下のコマンドを実行します。

```
$ oc get projecthelmchartrepositories --namespace my-namespace
```

**出力例**

```
NAME                    AGE
azure-sample-repo       1m
```

**注記**

クラスター管理者または適切な RBAC パーミッションを持つ通常ユーザーが特定の namespace のすべてのチャートリポジトリーを削除すると、その特定の namespace の **+Add** ビュー、**Developer Catalog**、および左側のナビゲーションパネルで Helm オプションを表示することはできません。

## 7.3.7. Helm チャートリポジトリーを追加するための認証情報および CA 証明書の作成

一部の Helm チャートリポジトリーに接続するには、認証情報とカスタム認証局 (CA) 証明書が必要です。Web コンソールと CLI を使用して認証情報と証明書を追加することができます。

**手順**

認証情報と証明書を設定し、CLI を使用して Helm チャートリポジトリーを追加します。

1. **openshift-config** namespace で、PEM でエンコードされた形式のカスタム CA 証明書で **ConfigMap** を作成し、これを設定マップ内の **ca-bundle.crt** キーに保存します。

```
$ oc create configmap helm-ca-cert \
--from-file=ca-bundle.crt=/path/to/certs/ca.crt \
-n openshift-config
```

2. **openshift-config** namespace で、クライアント TLS 設定を追加するために **Secret** オブジェクトを作成します。

```
$ oc create secret tls helm-tls-configs \
```

```
--cert=/path/to/certs/client.crt \
--key=/path/to/certs/client.key \
-n openshift-config
```

クライアント証明書とキーは PEM でエンコードされた形式であり、それぞれ **tls.crt** および **tls.key** キーに保存される必要があります。

3. 以下のように Helm リポジトリーを追加します。

```
$ cat <<EOF | oc apply -f -
apiVersion: helm.openshift.io/v1beta1
kind: HelmChartRepository
metadata:
  name: <helm-repository>
spec:
  name: <helm-repository>
  connectionConfig:
    url: <URL for the Helm repository>
    tlsConfig:
        name: helm-tls-configs
    ca:
 name: helm-ca-cert
EOF
```

**ConfigMap** および **Secret** は、**tlsConfig** および **ca** フィールドを使用して HelmChartRepository CR で使用されます。これらの証明書は、Helm リポジトリー URL への接続に使用されます。

4. デフォルトでは、認証されたユーザーはすべて設定済みのチャートにアクセスできます。ただし、証明書が必要なチャートリポジトリーの場合は、以下のように **openshift-config** namespace で **helm-ca-cert** 設定マップおよび **helm-tls-configs** シークレットへの読み取りアクセスを提供する必要があります。

```
$ cat <<EOF | kubectl apply -f -
apiVersion: rbac.authorization.k8s.io/v1
kind: Role
metadata:
  namespace: openshift-config
  name: helm-chartrepos-tls-conf-viewer
rules:
- apiGroups: [""]
  resources: ["configmaps"]
  resourceNames: ["helm-ca-cert"]
  verbs: ["get"]
- apiGroups: [""]
  resources: ["secrets"]
  resourceNames: ["helm-tls-configs"]
  verbs: ["get"]
---
kind: RoleBinding
apiVersion: rbac.authorization.k8s.io/v1
metadata:
  namespace: openshift-config
  name: helm-chartrepos-tls-conf-viewer
subjects:
  - kind: Group
```

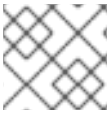
```
    apiGroup: rbac.authorization.k8s.io
    name: 'system:authenticated'
roleRef:
  apiGroup: rbac.authorization.k8s.io
  kind: Role
  name: helm-chartrepos-tls-conf-viewer
EOF
```

### 7.3.8. 証明書レベルでの Helm チャートのフィルタリング

**Developer Catalog** の認定レベルに基づいて Helm チャートをフィルターできます。

**手順**

1. **Developer** パースペクティブで、**+Add** ビューに移動し、プロジェクトを選択します。
2. **Developer Catalog** タイルから、**Helm Chart** オプションを選択して **Developer Catalog** ですべての Helm チャートを表示します。
3. Helm チャートのリストの左側にあるフィルターを使用して、必要なチャートをフィルターします。
   - **Chart Repositories** フィルターを使用して、**Red Hat Certification Charts** または **OpenShift Helm Charts** が提供したチャートをフィルターします。
   - **Source** フィルターを使用して、**Partners**、**Community** または **Red Hat** から提供されるチャートをフィルターします。認定チャートはアイコン ( ) で表示されます。



**注記**

プロバイダータイプが 1 つしかない場合は、**Source** フィルターは表示されません。

必要なチャートを選択してインストールできるようになりました。

### 7.3.9. Helm チャートリポジトリーの無効化

**HelmChartRepository** の **disabled** プロパティーを **true** に設定して、カタログにある特定の Helm チャートリポジトリーからの Helm チャートを無効にできます。

**手順**

- CLI を使用して Helm チャートリポジトリーを無効にするには、**disabled: true** フラグをカスタムリソースに追加します。たとえば、Azure サンプルチャートリポジトリーを削除するには、以下を実行します。

```
$ cat <<EOF | oc apply -f -
apiVersion: helm.openshift.io/v1beta1
kind: HelmChartRepository
metadata:
  name: azure-sample-repo
spec:
  connectionConfig:
```

```
  url:https://raw.githubusercontent.com/Azure-Samples/helm-charts/master/docs
 disabled: true
EOF
```

- Web コンソールを使用して、最近追加された Helm チャートリポジトリーを無効にするには、以下を実行します。

  1. **Custom Resource Definitions** に移動し、 **HelmChartRepository** カスタムリソースを検索します。
  2. **Instances** に移動し、無効にするリポジトリーを見つけ、その名前をクリックします。
  3. **YAML** タブに移動し、**spec** セクションに **disabled: true** フラグを追加し、**Save** をクリックします。

     **例**

     ```
     spec:
       connectionConfig:
         url: <url-of-the-repositoru-to-be-disabled>
       disabled: true
     ```

     リポジトリーは無効にされ、カタログには表示されなくなります。

## 7.4. HELM リリースの使用

Web コンソールの **Developer** パースペクティブを使用し、Helm リリースの更新、ロールバック、またはアンインストールを実行できます。

### 7.4.1. 前提条件

- Web コンソールにログイン し、**Developer パースペクティブ** に切り替えている。

### 7.4.2. Helm リリースのアップグレード

Helm リリースをアップグレードして、新規チャートバージョンにアップグレードしたり、リリース設定を更新したりできます。

**手順**

1. **Topology** ビューで Helm リリースを選択し、サイドパネルを表示します。
2. **Actions → Upgrade Helm Release** をクリックします。
3. **Upgrade Helm Release** ページで、アップグレード先とする **Chart Version** を選択してから **Upgrade** をクリックし、別の Helm リリースを作成します。**Helm Releases** ページには 2 つのリビジョンが表示されます。

### 7.4.3. Helm リリースのロールバック

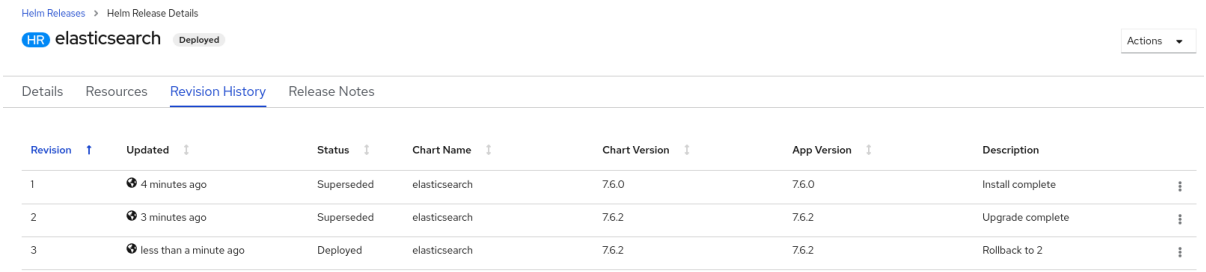
リリースに失敗する場合、Helm リリースを直前のバージョンにロールバックできます。

**手順**

Helm ビューを使用してリリースをロールバックするには、以下を実行します。

1. **Developer** パースペクティブで **Helm** ビューに移動し、namespace の **Helm Releases** を表示します。
2. リスト表示されているリソースに隣接する **Options** メニュー をクリックし、**Rollback** を選択します。
3. **Rollback Helm Release** ページで、ロールバックする **Revision** を選択し、 **Rollback** をクリックします。
4. **Helm Releases** ページで、チャートをクリックし、リリースの詳細およびリソースを表示します。
5. **Revision History** タブに移動し、チャートのすべてのリビジョンを表示します。

**図7.4 Helm リビジョン履歴**




6. 必要な場合は、さらに特定のリビジョンに隣接する **Options** メニュー を使用して、ロールバックするリビジョンを選択します。

## 7.4.4. Helm リリースのアンインストール

**手順**

1. **Topology** ビューで、 Helm リリースを右クリックし、 **Uninstall Helm Release** を選択します。
2. 確認プロンプトでチャートの名前を入力し、**Uninstall** をクリックします。

# 第8章 デプロイメント

## 8.1. DEPLOYMENT および DEPLOYMENTCONFIG オブジェクトについて

OpenShift Container Platform の **Deployment** および **DeploymentConfig** API オブジェクトは、一般的なユーザーアプリケーションに対する詳細な管理を行うためのよく似ているものの、異なる 2 つの方法を提供します。これらは、以下の個別の API オブジェクトで設定されています。

- アプリケーションの特定のコンポーネントの必要な状態を記述する、Pod テンプレートとしての **Deployment** または **DeploymentConfig**。
- **Deployment** オブジェクトには、1 つ以上の **レプリカセット** が使用され、これには Pod テンプレートとしてのデプロイメントの特定の時点の状態のレコードが含まれます。同様に、**DeploymentConfig** オブジェクトには、1 つ以上の **レプリケーションコントローラー** (以前はレプリカセットでした) 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。

**DeploymentConfig** オブジェクトで特定の機能または動作を指定する必要がない場合、**Deployment** オブジェクトを使用します。

### 8.1.1. デプロイメントのビルディングブロック

デプロイメントおよびデプロイメント設定は、それぞれビルディングブロックとして、ネイティブ Kubernetes API オブジェクトの **ReplicaSet** および **ReplicationController** の使用によって有効にされます。

ユーザーは、**Deployment** または **DeploymentConfig** オブジェクトによって所有されるレプリカセット、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は変更を適切に伝播します。

**ヒント**

既存のデプロイメントストラテジーが特定のユースケースに適さない場合で、デプロイメントのライフサイクル期間中に複数の手順を手動で実行する必要がある場合は、カスタムデプロイメントストラテジーを作成することを検討してください。

以下のセクションでは、これらのオブジェクトの詳細情報を提供します。

#### 8.1.1.1. レプリカセット

**ReplicaSet** は、指定された数の Pod レプリカが特定の時点で実行されるようにするネイティブの Kubernetes API オブジェクトです。

**注記**

カスタム更新のオーケストレーションが必要な場合や、更新が全く必要のない場合にのみレプリカセットを使用します。それ以外はデプロイメントを使用します。レプリカセットは個別に使用できますが、Pod 作成/削除/更新のオーケストレーションにはデプロイメントでレプリカセットを使用します。デプロイメントは、自動的にレプリカセットを管理し、Pod に宣言的更新を加えるので、作成するレプリカセットを手動で管理する必要はありません。

以下は、**ReplicaSet** 定義の例になります。

```
apiVersion: apps/v1
kind: ReplicaSet
metadata:
  name: frontend-1
  labels:
    tier: frontend
spec:
  replicas: 3
  selector: 1
    matchLabels: 2
      tier: frontend
    matchExpressions: 3
      - {key: tier, operator: In, values: [frontend]}
  template:
    metadata:
      labels:
        tier: frontend
    spec:
      containers:
      - image: openshift/hello-openshift
        name: helloworld
        ports:
        - containerPort: 8080
          protocol: TCP
      restartPolicy: Always
```

1. 一連のリソースに対するラベルのクエリー。**matchLabels** と **matchExpressions** の結果は論理的に結合されます。
2. セレクターに一致するラベルでリソースを指定する等価ベースのセレクター
3. キーをフィルターするセットベースのセレクター。これは、**tier** と同等のキー、**frontend** と同等の値のリソースをすべて選択します。

#### 8.1.1.2. レプリケーションコントローラー

レプリカセットと同様に、レプリケーションコントローラーは、Pod の指定された数のレプリカが常に実行されるようにします。Pod が終了または削除された場合に、レプリケーションコントローラーは定義した数になるまでインスタンス化する数を増やします。同様に、必要以上の数の Pod が実行されている場合には、定義された数に一致させるために必要な数の Pod を削除します。レプリカセットとレプリケーションコントローラーの相違点は、レプリカセットではセットベースのセレクター要件をサポートし、レプリケーションコントローラーは等価ベースのセレクター要件のみをサポートする点です。

レプリケーションコントローラー設定は以下で設定されています。

- 必要なレプリカ数 (これはランタイム時に調整可能)。
- レプリケートされた Pod の作成時に使用する **Pod** 定義。
- 管理された Pod を識別するためのセレクター。

セレクターは、レプリケーションコントローラーが管理する Pod に割り当てられるラベルセットで

す。これらのラベルは、**Pod** 定義に組み込まれ、レプリケーションコントローラーがインスタンス化します。レプリケーションコントローラーは、必要に応じて調節するために、セレクターを使用して、すでに実行中の Pod 数を判断します。

レプリケーションコントローラーは、追跡もしませんが、負荷またはトラフィックに基づいて自動スケーリングを実行することもありません。この場合は、レプリカ数を外部の自動スケーラーで調整する必要があります。

**注記**

レプリケーションコントローラーを直接作成するのではなく、**DeploymentConfig** を使用してレプリケーションコントローラーを作成します。

カスタムオーケストレーションが必要な場合や、更新が必要ない場合は、レプリケーションコントローラーの代わりにレプリカセットを使用します。

以下は、レプリケーションコントローラー定義の例です。

```
apiVersion: v1
kind: ReplicationController
metadata:
  name: frontend-1
spec:
  replicas: 1 (1)
  selector: (2)
    name: frontend
  template: (3)
    metadata:
      labels: (4)
        name: frontend (5)
    spec:
      containers:
      - image: openshift/hello-openshift
        name: helloworld
        ports:
        - containerPort: 8080
          protocol: TCP
      restartPolicy: Always
```

1. 実行する Pod のコピー数です。
2. 実行する Pod のラベルセレクターです。
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 のラベルにはラベルセレクターからのものが含まれている必要があります。
5. パラメーター拡張後の名前の最大長さは 63 文字です。

### 8.1.2. Deployments

Kubernetes は、**Deployment** という OpenShift Container Platform のファーストクラスのネイティブ API オブジェクトを提供します。**Deployment** オブジェクトは、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で希望する状態を記述します。デプロイメントは、Pod のライフサイクルをオーケストレーションするレプリカセットを作成します。

たとえば、以下のデプロイメント定義はレプリカセットを作成し、1 つの **hello-openshift** Pod を起動します。

**デプロイメントの定義**

```
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
```

### 8.1.3. DeploymentConfig オブジェクト

レプリケーションコントローラーでビルドする OpenShift Container Platform は **DeploymentConfig** オブジェクトの概念を使用したソフトウェアの開発およびデプロイメントライフサイクルの拡張サポートを追加します。最も単純な場合に、**DeploymentConfig** オブジェクトは新規アプリケーションコントローラーのみを作成し、それに Pod を起動させます。

ただし、**DeploymentConfig** オブジェクトの OpenShift Container Platform デプロイメントは、イメージの既存デプロイメントから新規デプロイメントに移行する機能を提供し、レプリケーションコントローラーの作成前後に実行されるフックも定義します。

**DeploymentConfig** デプロイメントシステムは以下の機能を提供します。

- アプリケーションを実行するためのテンプレートである **DeploymentConfig** オブジェクト。
- 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。
- 直前のバージョンから新規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デプロイメントストラテジー。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異なる時点でカスタム動作を実行するためのフックのセット (ライフサイクルフック)。
- 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。

**DeploymentConfig** オブジェクトを作成すると、レプリケーションコントローラーが、**DeploymentConfig** オブジェクトの Pod テンプレートとして作成されます。デプロイメントが変更されると、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て以前のレプリケーションコントローラーのスケールダウン、および新規レプリケーションコントーラーのスケールアップが行われます。

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。

OpenShift Container Platform **DeploymentConfig** オブジェクトは以下の詳細を定義します。

1. **ReplicationController** 定義の要素。
2. 新規デプロイメントの自動作成のトリガー。
3. デプロイメント間の移行ストラテジー。
4. ライフサイクルフック。

デプロイヤー Pod は、デプロイメントがトリガーされるたびに、手動または自動であるかを問わず、(古いレプリケーションコントローラーの縮小、新規レプリケーションコントローラーの拡大およびフックの実行などの) デプロイメントを管理します。デプロイメント Pod は、デプロイメントのログを維持するためにデプロイメントの完了後は無期限で保持されます。デプロイメントが別のものに置き換えられる場合、以前のレプリケーションコントローラーは必要に応じて簡単なロールバックを有効にできるように保持されます。

**DeploymentConfig 定義の例**

```
apiVersion: apps.openshift.io/v1
kind: DeploymentConfig
metadata:
  name: frontend
spec:
  replicas: 5
  selector:
    name: frontend
  template: { ... }
  triggers:
  - type: ConfigChange 1
  - imageChangeParams:
      automatic: true
      containerNames:
      - helloworld
      from:
        kind: ImageStreamTag
        name: hello-openshift:latest
    type: ImageChange 2
  strategy:
    type: Rolling 3
```

1 設定変更トリガーにより、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。

2 イメージ変更トリガーにより、新規デプロイメントが、バッキングイメージの新規バージョンが名前付きイメージストリームで利用可能になる際には常に作成されます。

3 デフォルトの **Rolling** ストラテジーにより、デプロイメント間のダウンタイムなしの移行が行われます。

### 8.1.4. Deployment および DeploymentConfig オブジェクトの比較

Kubernetes **Deployment** および OpenShift Container Platform でプロビジョニングされる **DeploymentConfig** オブジェクトの両方が OpenShift Container Platform でサポートされていますが、**DeploymentConfig** オブジェクトで提供される特定の機能または動作が必要でない場合、**Deployment** を使用することが推奨されます。

以下のセクションでは、使用するタイプの決定に役立つ 2 つのオブジェクト間の違いを詳述します。

#### 8.1.4.1. 設計

**Deployment** と **DeploymentConfig** オブジェクトの重要な違いの 1 つとして、ロールアウトプロセスで各設計で選択される CAP theorem (原則) のプロパティーがあります。**DeploymentConfig** オブジェクトは整合性を優先しますが、**Deployments** オブジェクトは整合性よりも可用性を優先します。

**DeploymentConfig** オブジェクトの場合、デプロヤー Pod を実行するノードがダウンする場合、ノードの置き換えは行われません。プロセスは、ノードが再びオンラインになるまで待機するか、手動で削除されます。ノードを手動で削除すると、対応する Pod も削除されます。つまり、kubelet は関連付けられた Pod も削除するため、Pod を削除してロールアウトの固定解除を行うことはできません。

一方、Deployment ロールアウトはコントローラーマネージャーから実行されます。コントローラーマネージャーはマスター上で高可用性モードで実行され、リーダー選択アルゴリズムを使用して可用性を整合性よりも優先するように設定します。障害の発生時には、他の複数のマスターが同時に同じデプロイメントに対して作用する可能性がありますが、この問題は障害の発生直後に調整されます。

#### 8.1.4.2. デプロイメント固有の機能

**ロールオーバー**
**Deployment** オブジェクトのデプロイメントプロセスは、すべての新規ロールアウトにデプロイヤー Pod を使用する **DeploymentConfig** オブジェクトとは対照的に、コントローラーループで実行されます。つまり、**Deployment** オブジェクトにはできるだけ多くの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すべて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。

**DeploymentConfig** オブジェクトでは、実行できるデプロイヤー Pod は最大 1 つとなっています。複数のデプロイヤーがある場合は競合が生じ、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、2 つのレプリケーションコントローラーのみを一度にアクティブにできます。最終的には、**Deployment** オブジェクトのロールアウトが速くなります。

**比例スケーリング**
デプロイメントコントローラーのみが **Deployment** オブジェクトが所有する新旧の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が可能です。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。

**DeploymentConfig** オブジェクトは、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にスケーリングできません。

**ロールアウト中の一時停止**
Deployment 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。ただし、現時点ではデプロイヤー Pod を一時停止できません。ロールアウトの途中でデプロイメントを一時停止しようとすると、デプロイヤープロセスは影響を受けず、完了するまで続行されます。

#### 8.1.4.3. DeploymentConfig オブジェクト固有の機能

**自動ロールバック**
現時点で、デプロイメントでは、問題の発生時の最後に正常にデプロイされたレプリカセットへの自動ロールバックをサポートしていません。

**トリガー**
Deployment の場合、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、暗黙的な設定変更トリガーが含まれます。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>
```

**ライフサイクルフック**
Deployment ではライフサイクルフックをサポートしていません。

**カスタムストラテジー**
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をサポートしていません。

## 8.2. デプロイメントプロセスの管理

### 8.2.1. DeploymentConfig オブジェクトの管理

**DeploymentConfig** オブジェクトは、OpenShift Container Platform Web コンソールの **Workloads** ページからか、**oc** CLI を使用して管理できます。以下の手順は、特に指定がない場合の CLI の使用法を示しています。

#### 8.2.1.1. デプロイメントの開始

アプリケーションのデプロイメントプロセスを開始するために、ロールアウトを開始できます。

**手順**

1. 既存の **DeploymentConfig** から新規デプロイメントプロセスを開始するには、以下のコマンドを実行します。

```
$ 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 はデプロイされません。

#### 8.2.1.2. デプロイメントの表示

アプリケーションの利用可能なすべてのリビジョンについての基本情報を取得するためにデプロイメントを表示できます。

**手順**

1. 現在実行中のデプロイメントプロセスを含む、指定した **DeploymentConfig** オブジェクトについての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についての詳細を表示するには、以下を実行します。

```
$ oc rollout history dc/<name>
```

2. リビジョンに固有の詳細情報を表示するには、**--revision** フラグを追加します。

```
$ oc rollout history dc/<name> --revision=1
```

3. **DeploymentConfig** オブジェクトおよびその最新バージョンの詳細については、**oc describe** コマンドを使用します。

```
$ oc describe dc <name>
```

#### 8.2.1.3. デプロイメントの再試行

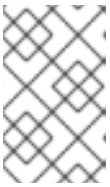
現行リビジョンの **DeploymentConfig** がデプロイに失敗した場合、デプロイメントプロセスを再起動することができます。

**手順**

1. 失敗したデプロイメントプロセスを再起動するには、以下を実行します。

```
$ oc rollout retry dc/<name>
```

最新リビジョンのデプロイメントに成功した場合には、このコマンドによりメッセージが表示され、デプロイメントプロセスは試行されません。



**注記**

デプロイメントを再試行すると、デプロイメントプロセスが再起動され、新しいデプロイメントリビジョンは作成されません。再起動されたレプリケーションコントローラーは、失敗したときと同じ設定を使用します。

#### 8.2.1.4. 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。

**手順**

1. 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>
```

**DeploymentConfig** オブジェクトのテンプレートは、undo コマンドで指定された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、新規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ない場合には、最後に成功したデプロイメントのリビジョンが使用されます。

2. 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、**DeploymentConfig** オブジェクトのイメージ変更トリガーがロールバックの一部として無効にされます。
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**

デプロイメント設定は、最新のデプロイメントプロセスが失敗した場合の、設定の最後に成功したリビジョンへの自動ロールバックもサポートします。この場合、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、ユーザーがその設定の修正を行う必要があります。

#### 8.2.1.5. コンテナー内でのコマンドの実行

コマンドをコンテナーに追加して、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。

**手順**

1. **command** パラメーターを、**DeploymentConfig** オブジェクトの **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**)。

```
kind: DeploymentConfig
apiVersion: apps.openshift.io/v1
metadata:
  name: example-dc
# ...
spec:
  template:
# ...
    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。

```
kind: DeploymentConfig
apiVersion: apps.openshift.io/v1
metadata:
```

```
  name: example-dc
# ...
spec:
  template:
# ...
    spec:
      containers:
        - name: example-spring-boot
          image: 'image'
          command:
            - java
          args:
            - '-jar'
            - /opt/app-root/springboots2idemo.jar
# ...
```

#### 8.2.1.6. デプロイメントログの表示

**手順**

1. 指定の **DeploymentConfig** オブジェクトに関する最新リビジョンのログをストリームするには、以下を実行します。

```
$ 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、コマンドが、Pod のデプロイを行うプロセスのログを返します。成功した場合には、アプリケーションの Pod からのログを返します。

2. 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>
```

#### 8.2.1.7. デプロイメントトリガー

**DeploymentConfig** オブジェクトには、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 **DeploymentConfig** オブジェクトに定義されていない場合は、設定変更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。

**設定変更デプロイメントトリガー**
設定変更トリガーにより、**DeploymentConfig** オブジェクトの Pod テンプレートで設定の変更が検出されるたびに、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**

設定変更トリガーが **DeploymentConfig** オブジェクトに定義されている場合は、**DeploymentConfig** オブジェクト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が自動的に作成され、一時停止されません。

### 設定変更デプロイメントトリガー

```
kind: DeploymentConfig
apiVersion: apps.openshift.io/v1
metadata:
  name: example-dc
# ...
spec:
# ...
  triggers:
    - type: "ConfigChange"
```

**イメージ変更デプロイメントトリガー**
イメージ変更トリガーにより、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。

### イメージ変更デプロイメントトリガー

```
kind: DeploymentConfig
apiVersion: apps.openshift.io/v1
metadata:
  name: example-dc
# ...
spec:
# ...
  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 **DeploymentConfig** オブジェクトの **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**

イメージ変更トリガーが **DeploymentConfig** で定義され (設定変更トリガーおよび **automatic=false** が指定されるか、**automatic=true** が指定される)、イメージ変更トリガーで参照されているイメージストリームタグがまだ存在していない場合、ビルドによりイメージがイメージストリームタグ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。

#### 8.2.1.7.1. デプロイメントトリガーの設定

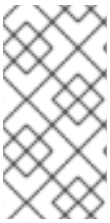
**手順**

1. **oc set triggers** コマンドを使用して、**DeploymentConfig** オブジェクトにデプロイメントトリガーを設定することができます。たとえば、イメージ変更トリガーを設定するには、以下のコマンドを使用します。

```
$ oc set triggers dc/<dc_name> \
    --from-image=<project>/<image>:<tag> -c <container_name>
```

### 8.2.1.8. 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 (メモリーおよび一時ストレージ) 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。



**注記**

デプロイメントの最小メモリー制限は 12 MB です。**Cannot allocate memory** Pod イベントのためにコンテナーの起動に失敗すると、メモリー制限は低くなります。メモリー制限を引き上げるか、これを削除します。制限を削除すると、Pod は制限のないノードのリソースを消費できるようになります。

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。

**手順**

1. 以下の例では、**resources**、**cpu**、**memory**、および **ephemeral-storage** はそれぞれオプションです。

```
kind: Deployment
apiVersion: apps/v1
metadata:
  name: hello-openshift
# ...
spec:
# ...
  type: "Recreate"
  resources:
    limits:
```

```
      cpu: "100m" (1)
      memory: "256Mi" (2)
      ephemeral-storage: "1Gi" (3)
```

(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)。

(3) **ephemeral-storage** はバイト単位です。**1Gi** は 1073741824 バイト (2 ^ 30) を表します。

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:

```
kind: Deployment
apiVersion: apps/v1
metadata:
  name: hello-openshift
# ...
spec:
# ...
  type: "Recreate"
  resources:
    requests: (1)
      cpu: "100m"
      memory: "256Mi"
      ephemeral-storage: "1Gi"
```

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

- プロジェクト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。

デプロイメントリソースを設定するには、上記のいずれかのオプションを選択してください。それ以外の場合は、デプロイ Pod の作成は、クォータ基準を満たしていないことを示すメッセージを出して失敗します。

**関連情報**

- リソース制限および要求の詳細は、Understanding managing application memory を参照してください。

#### 8.2.1.9. 手動のスケーリング

ロールバック以外に、手動スケーリングにより、レプリカの数を詳細に管理できます。

**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。

**手順**

1. **DeploymentConfig** オブジェクトを手動でスケーリングするには、**oc scale** コマンドを使用します。たとえば、以下のコマンドは、**frontend DeploymentConfig** オブジェクトを **3** に設定します。

```
$ oc scale dc frontend --replicas=3
```

レプリカの数は最終的に、**DeploymentConfig** オブジェクトの **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播されます。

#### 8.2.1.10. DeploymentConfig オブジェクトからのプライベートリポジトリーへのアクセス

シークレットを **DeploymentConfig** オブジェクトに追加し、プライベートリポジトリーからイメージにアクセスできるようにします。この手順では、OpenShift Container Platform Web コンソールを使用する方法を示します。

**手順**

1. 新しいプロジェクトを作成する。
2. **Workloads → Secrets** に移動します。
3. 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。
4. **Workloads → DeploymentConfigs** に移動します。
5. **DeploymentConfig** オブジェクトを作成します。
6. **DeploymentConfig** エディターページで、**Pull Secret** を設定し、変更を保存します。

#### 8.2.1.11. 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。

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。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。

**手順**

1. 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
metadata:
  name: my-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

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。ここで指定されるラベルは、クラスター管理者によって追加される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。

#### 8.2.1.12. 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。

**手順**

1. **DeploymentConfig** オブジェクトを編集します。

```
$ 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。

```
apiVersion: apps.openshift.io/v1
kind: DeploymentConfig
metadata:
  name: example-dc
# ...
spec:
# ...
  securityContext: {}
  serviceAccount: <service_account>
  serviceAccountName: <service_account>
```

## 8.3. デプロイメントストラテジーの使用

**デプロイメントストラテジー** は、ユーザーが変更にほとんど気付かないように、ダウンタイムなしでアプリケーションを変更またはアップグレードするために使用されます。

ユーザーは通常、ルーターによって処理されるルートを介してアプリケーションにアクセスするため、デプロイメント戦略は **DeploymentConfig** オブジェクト機能またはルーティング機能に重点を置くことができます。**DeploymentConfig** オブジェクトの機能に焦点を当てた戦略は、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、**DeploymentConfig** オブジェクトでサポートされ、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。

### 8.3.1. デプロイメントストラテジーの選択

デプロイメントストラテジーを選択する場合に、以下を考慮してください。

- 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。

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、**DeploymentConfig** オブジェクトは、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。

### 8.3.2. 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。ローリングストラテジーは、**DeploymentConfig** オブジェクトにストラテジーが指定されていない場合に使用されるデフォルトのデプロイメントストラテジーです。

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。重大な問題が生じる場合、ローリングデプロイメントは中止される場合があります。

**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。

**ローリングストラテジー定義の例**

```
kind: DeploymentConfig
apiVersion: apps.openshift.io/v1
metadata:
  name: example-dc
# ...
spec:
# ...
  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:

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。以下に例を示します。

- **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。

#### 8.3.2.1. 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 チェックが成功しない場合、カナリアインスタンスは削除され、**DeploymentConfig** オブジェクト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あり、新規インスタンスが使用できる状態にするために必要に応じて高度な設定をすることができます。(実際のユーザーワークロードを新規インスタンスに送信するなどの) アプリケーションのより複雑なチェックを実装する必要がある場合、カスタムデプロイメントや blue-green デプロイメントストラテジーの実装を検討してください。

#### 8.3.2.2. ローリングデプロイメントの作成

ローリングデプロイメントは OpenShift Container Platform のデフォルトタイプです。CLI を使用してローリングデプロイメントを作成できます。

**手順**

1. Quay.io にあるデプロイメントイメージのサンプルに基づいてアプリケーションを作成します。

```
$ oc new-app quay.io/openshifttest/deployment-example:latest
```

**注記**

このイメージはポートを公開しません。外部 LoadBalancer サービスでアプリケーションを公開するか、パブリックインターネット経由でアプリケーションにアクセスできるようにする必要がある場合は、この手順を完了した後に **oc expose dc/deployment-example --port=<port>** コマンドを使用してサービスを作成します。

2. 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するか、サービス IP を直接使用してください。

```
$ oc expose svc/deployment-example
```

3. **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。

4. レプリカが最大 3 つになるまで、**DeploymentConfig** オブジェクトをスケーリングします。

```
$ oc scale dc/deployment-example --replicas=3
```

5. 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
```

6. 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。

7. 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、Pod が徐々に v2 に追加され、v1 から削除されます。

```
$ 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。(rediness チェックをパスした後に) 新規 Pod に **ready** のマークが付けられると、デプロイメントプロセスは継続されます。

Pod が準備状態にならない場合、プロセスは中止し、デプロイメントは直前のバージョンにロールバックします。

#### 8.3.2.3. 開発者パースペクティブを使用したデプロイメントの編集

**Developer** パースペクティブを使用して、デプロイメントのデプロイメントストラテジー、イメージ設定、環境変数、詳細オプションを編集できます。

**前提条件**

- Web コンソールの **Developer** パースペクティブを使用している。
- アプリケーションを作成している。

**手順**

1. **Topology** ビューに移動します。
2. アプリケーションをクリックして、**Details** パネルを表示します。
3. **Actions** ドロップダウンメニューで **Edit Deployment** を選択し、**Edit Deployment** ページを表示します。
4. デプロイメントの以下の **Advanced options** を編集できます。
   a. オプション: **Pause rollouts** をクリックして **Pause rollouts for this deployment**チェックボックスを選択すると、ロールアウトを一時停止できます。
   ロールアウトを一時停止すると、ロールアウトをトリガーせずにアプリケーションを変更できます。ロールアウトはいつでも再開できます。

   b. オプション: **Scaling** をクリックし、**Replicas** のカズを変更することでイメージのインスタンス数を変更できます。
5. **Save** をクリックします。

#### 8.3.2.4. 開発者パースペクティブを使用したローリングデプロイメントの開始

ローリングデプロイメントを開始することで、アプリケーションをアップグレードできます。

**前提条件**

- Web コンソールの **Developer** パースペクティブを使用している。

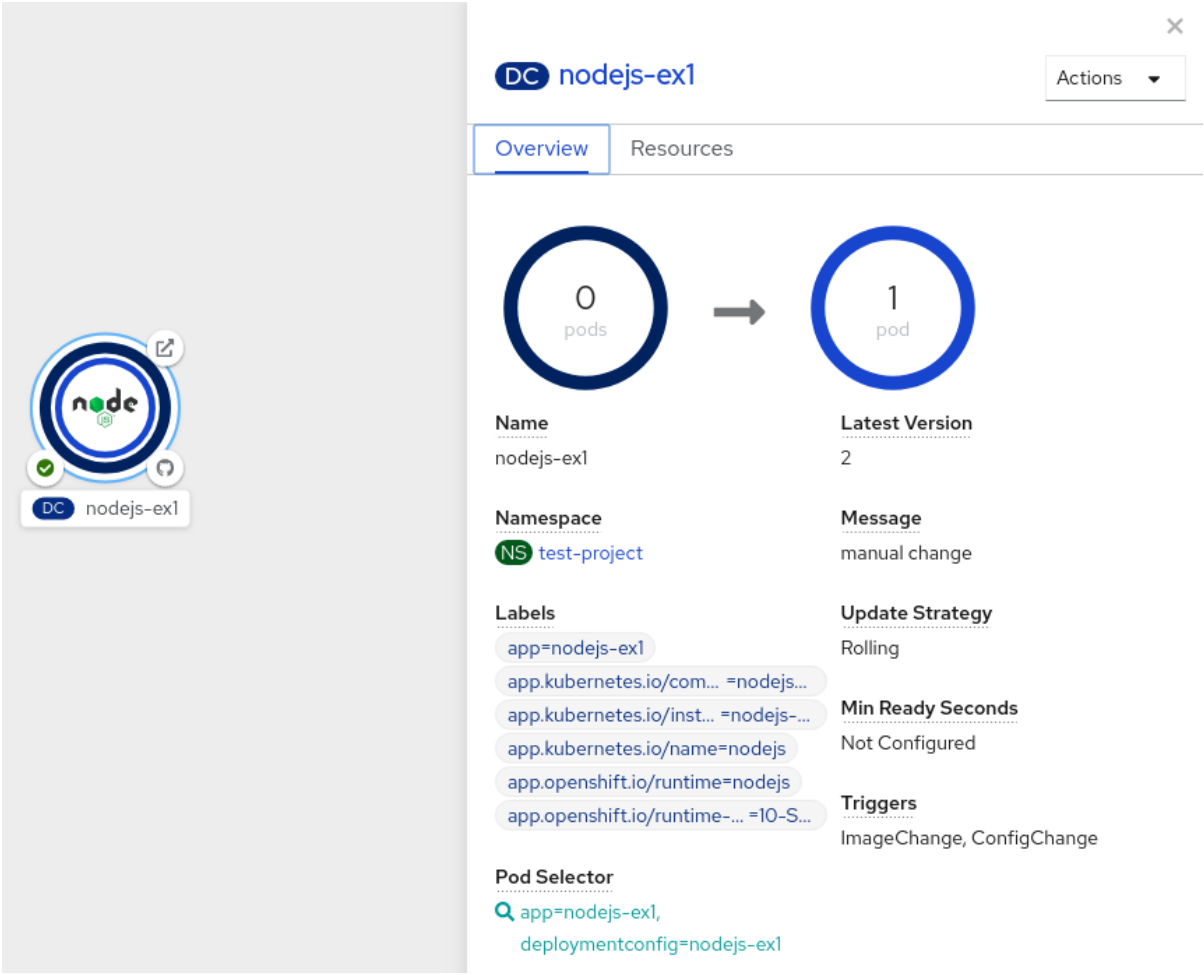
- アプリケーションを作成している。

**手順**

1. **Topology** ビューでアプリケーションノードをクリックすると、サイドパネルに **Overview** タブが表示されます。**Update Strategy** がデフォルトの **Rolling** ストラテジーに設定されていることに注意してください。

2. **Actions** ドロップダウンメニューで、**Start Rollout** を選択し、ローリング更新を開始します。ローリングデプロイメントは、新しいバージョンのアプリケーションを起動してから、古いバージョンを終了します。

**図8.1 ローリング更新**




**関連情報**

- Developer パースペクティブを使用して OpenShift Container Platform でアプリケーションを作成し、デプロイする
- Topology ビューを使用してプロジェクトにアプリケーションを表示し、デプロイメントのステータスを確認し、それらと対話する

### 8.3.3. 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。

**再作成ストラテジー定義の例**

```
kind: Deployment
apiVersion: apps/v1
metadata:
  name: hello-openshift
# ...
spec:
# ...
  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:

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. 任意の **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。

**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。

#### 8.3.3.1. 開発者パースペクティブを使用したデプロイメントの編集

Developer パースペクティブを使用して、デプロイメントのデプロイメントストラテジー、イメージ設定、環境変数、詳細オプションを編集できます。

**前提条件**

**前提条件**

- Web コンソールの **Developer** パースペクティブを使用している。
- アプリケーションを作成している。

**手順**

1. **Topology** ビューに移動します。
2. アプリケーションをクリックして、**Details** パネルを表示します。
3. **Actions** ドロップダウンメニューで **Edit Deployment** を選択し、**Edit Deployment** ページを表示します。
4. デプロイメントの以下の **Advanced options** を編集できます。
    a. オプション: **Pause rollouts** をクリックして **Pause rollouts for this deployment**チェックボックスを選択すると、ロールアウトを一時停止できます。
    ロールアウトを一時停止すると、ロールアウトをトリガーせずにアプリケーションを変更できます。ロールアウトはいつでも再開できます。
    b. オプション: **Scaling** をクリックし、**Replicas** のカズを変更することでイメージのインスタンス数を変更できます。
5. **Save** をクリックします。

#### 8.3.3.2. 開発者パースペクティブを使用した再作成デプロイメントの開始

Web コンソールの **Developer** パースペクティブを使用して、デプロイメントストラテジーをデフォルトのローリング更新から再作成更新に切り替えることができます。

**前提条件**

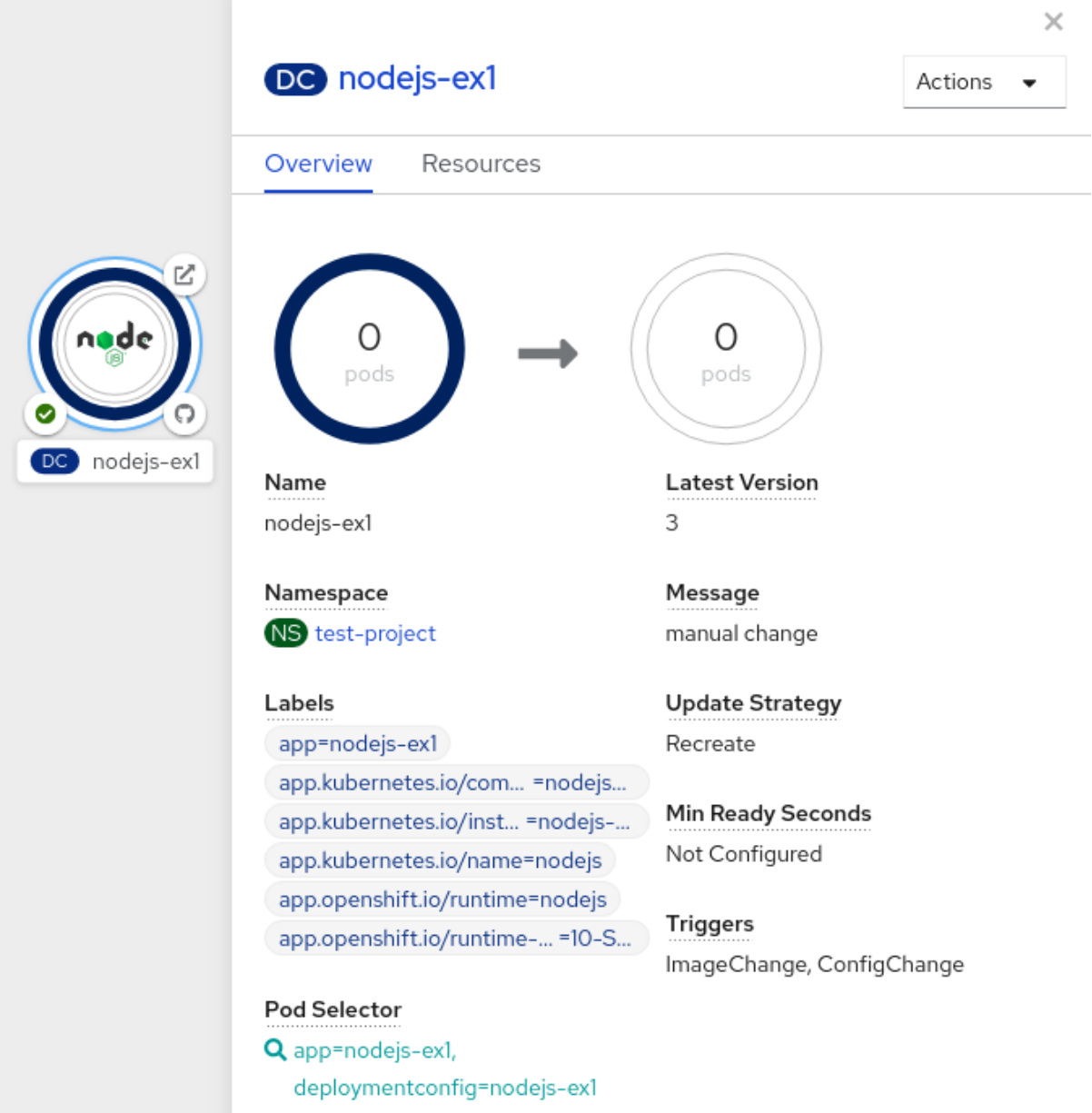
- Web コンソールの **Developer** パースペクティブにいることを確認します。
- **Add** ビューを使用してアプリケーションを作成し、これが **Topology** ビューにデプロイされていることを確認します。

**手順**

再作成更新ストラテジーに切り替え、アプリケーションをアップグレードするには、以下を実行します。

1. アプリケーションをクリックして、**Details** パネルを表示します。
2. **Actions** ドロップダウンメニューで、**Edit Deployment Config** を選択し、アプリケーションのデプロイメント設定の詳細を確認します。
3. YAML エディターで **spec.strategy.type** を **Recreate** に変更し、**Save** をクリックします。
4. **Topology** ビューでノードを選択し、サイドパネルの **Overview** タブを表示します。これで、**Update Strategy** は **Recreate** に設定されます。
5. **Actions** ドロップダウンメニューを使用し、**Start Rollout** を選択し、再作成ストラテジーを使用して更新を開始します。再作成ストラテジーはまず、アプリケーションの古いバージョンの Pod を終了してから、新規バージョンの Pod を起動します。

図8.2 再作成更新




**関連情報**

- Developer パースペクティブを使用して OpenShift Container Platform でアプリケーションを作成し、デプロイする
- Topology ビューを使用してプロジェクトにアプリケーションを表示し、デプロイメントのステータスを確認し、それらと対話する

### 8.3.4. 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。

**カスタムストラテジー定義の例**

```
kind: DeploymentConfig
apiVersion: apps.openshift.io/v1
metadata:
  name: example-dc
# ...
```

```
spec:
# ...
  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 は以下の環境変数を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。

または **customParams** オブジェクトを使用して、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。カスタムのシェルスクリプト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。

```
kind: DeploymentConfig
apiVersion: apps.openshift.io/v1
metadata:
  name: example-dc
# ...
spec:
# ...
  strategy:
    type: Rolling
    customParams:
      command:
      - /bin/sh
      - -c
      - |
        set -e
        openshift-deploy --until=50%
        echo Halfway there
        openshift-deploy
        echo Complete
```

この設定により、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。

#### 8.3.4.1. 開発者パースペクティブを使用したデプロイメントの編集

**Developer** パースペクティブを使用して、デプロイメントのデプロイメントストラテジー、イメージ設定、環境変数、詳細オプションを編集できます。

**前提条件**

- Web コンソールの **Developer** パースペクティブを使用している。
- アプリケーションを作成している。

**手順**

1. **Topology** ビューに移動します。
2. アプリケーションをクリックして、**Details** パネルを表示します。
3. **Actions** ドロップダウンメニューで **Edit Deployment** を選択し、**Edit Deployment** ページを表示します。
4. デプロイメントの以下の **Advanced options** を編集できます。
    a. オプション: **Pause rollouts** をクリックして **Pause rollouts for this deployment**チェックボックスを選択すると、ロールアウトを一時停止できます。
    ロールアウトを一時停止すると、ロールアウトをトリガーせずにアプリケーションを変更できます。ロールアウトはいつでも再開できます。
    b. オプション: **Scaling** をクリックし、**Replicas** のカズを変更することでイメージのインスタンス数を変更できます。
5. **Save** をクリックします。

### 8.3.5. ライフサイクルフック

ローリングおよび再作成ストラテジーは、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** またはデプロイメントフックをサポートします。

**pre ライフサイクルフックの例**

```
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 **失敗ポリシー** 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 **execNewPod** フィールドで指定されます。

**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、**DeploymentConfig** オブジェクト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の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: apps.openshift.io/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** です。
- 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、**DeploymentConfig** オブジェクト Pod から **data** ボリュームを継承します。

#### 8.3.5.1. ライフサイクルフックの設定

CLI を使用してデプロイメント用に、ライフサイクルフックまたはデプロイメントフックを設定できます。

**手順**

1. **oc set deployment-hook** コマンドを使用して、必要なフックのタイプを設定します (**--pre**、**--mid**、または **--post**)。たとえば、デプロイメント前のフックを設定するには、以下を実行します。

```
$ oc set deployment-hook dc/frontend \
    --pre -c helloworld -e CUSTOM_VAR1=custom_value1 \
    --volumes data --failure-policy=abort -- /usr/bin/command arg1 arg2
```

## 8.4. ルートベースのデプロイメントストラテジーの使用

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 **Deployment** オブジェクトを使用して、アプリケーションに解決されるすべてのルートのユーザーが確認できる変更を実行します。このセクションで説明される他の高度なストラテジーでは、ルーターを **Deployment** オブジェクトと併用して特定のルートに影響を与えます。

最も一般的なルートベースの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 (green バージョン) を、テストと評価用に起動しつつ、安定版 (blue 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、green 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、blue 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イ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。希望と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る場合があります。

### 8.4.1. プロキシーシャードおよびトラフィック分割

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。

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。

### 8.4.2. 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、以前のバージョン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**と呼ばれる複雑な問題です。

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 デプロイメントを使用できます。 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。

### 8.4.3. 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します。これにより、ロードバランサーによって他のアクティブなインスタンスにトラフィックが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か、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 テンプレートは正常に終了する期間 (デフォルトの 30 秒) を制御し、必要に応じてこれらを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。

### 8.4.4. Blue-Green デプロイメント

Blue-green デプロイメントでは、同時にアプリケーションの 2 つのバージョンを実行し、実稼働版 (blue バージョン) からより新しいバージョン (green 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。

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。

#### 8.4.4.1. Blue-Green デプロイメントの設定

Blue-green デプロイメントでは 2 つの **Deployment** を使用します。どちらも実行され、実稼働のデプロイメントはルートが指定するサービスによって変わります。この際、各 **Deployment** オブジェクトは異なるサービスに公開されます。

**注記**

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。

新規バージョンに新規ルートを作成し、これをテストすることができます。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 (green) 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 (blue) バージョンにロールバックすることができます。

**手順**

1. 2 つの独立したアプリケーションコンポーネントを作成します。

   a. **v1** イメージを **example-blue** サービスで実行するサンプルアプリケーションのコピーを作成します。

   ```
   $ oc new-app openshift/deployment-example:v1 --name=example-blue
   ```

b. **example-green** サービスで **v2** イメージを使用する 2 つ目のコピーを作成します。

```
$ oc new-app openshift/deployment-example:v2 --name=example-green
```

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-blue --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。

4. ルートを編集して、サービス名を **example-green** に変更します。

```
$ oc patch route/bluegreen-example -p '{"spec":{"to":{"name":"example-green"}}}'
```

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。

### 8.4.5. 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割合を制御できるので、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際に、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生じる場合があります。

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イ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を有効にするには、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 CLI で N-1 互換性をサポートします。

#### 8.4.5.1. 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** の場合は、サービスはロードバランシングに参加せず、既存の持続する接続を継続的に提供します。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では、最終的に **weight** は意図される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、予想される負荷分散の **weight** を得るために Pod の数を減らします。

### 手順

A/B 環境を設定するには、以下を実行します。

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。それぞれが **Deployment** オブジェクトを作成します。これらのアプリケーションは同じプログラム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。

   a. 最初のアプリケーションを作成します。以下の例では、**ab-example-a** という名前のアプリケーションを作成します。

   ```
   $ oc new-app openshift/deployment-example --name=ab-example-a
   ```

   b. 2 番目のアプリケーションを作成します。

   ```
   $ oc new-app openshift/deployment-example:v2 --name=ab-example-b
   ```

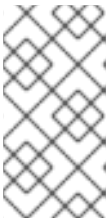
   どちらのアプリケーションもデプロイされ、サービスが作成されます。

2. 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-a.<project>.<router_domain>** でアプリケーションを参照して、予想されるバージョンが表示されていることを確認します。

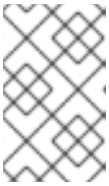
3. ルートをデプロイする場合には、ルーターはサービスに指定した **weights** に従ってトラフィックを分散します。この時点では、デフォルトの **weight=1** と指定されたサービスが 1 つ存在するので、すべての要求がこのサービスに送られます。他のサービスを **alternateBackends** として追加し、**weights** を調整すると、A/B 設定が機能するようになります。これは、**oc set route-backends** コマンドを実行するか、ルートを編集して実行できます。



**注記**

また、**alternateBackends** を使用する場合は、**roundrobin** ロードバランシング戦略を使用して、重みに基づいてリクエストが想定どおりにサービスに分散されるようにします。**roundrobin** は、ルートアノテーション を使用してルートに設定できます。

**oc set route-backend** を **0** に設定することは、サービスがロードバランシングに参加しないが、既存の持続する接続を提供し続けることを意味します。



**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>
```

**出力例**

```
apiVersion: route.openshift.io/v1
kind: Route
metadata:
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
# ...
```

#### 8.4.5.1.1. Web コンソールを使用した既存ルートの重みの管理

**手順**

1. **Networking → Routes** ページに移動します。
2. 編集するルートの横にある  Actions メニューをクリックし、**Edit Route** を選択します。
3. YAML ファイルを編集します。**weight** を **0** から **256** の間の整数になるように更新します。これは、他のターゲット参照オブジェクトに対するターゲットの相対的な重みを指定します。値 **0** はこのバックエンドへの要求を抑制します。デフォルトは **100** です。オプションについての詳細は、**oc explain routes.spec.alternateBackends** を実行します。
4. **Save** をクリックします。

#### 8.4.5.1.2. Web コンソールを使用した新規ルートの重みの管理

1. **Networking → Routes** ページに移動します。
2. **Create Route** をクリックします。
3. ルートの **Name** を入力します。
4. **Service** を選択します。
5. **Add Alternate Service** をクリックします。

6. **Weight** および **Alternate Service Weight** の値を入力します。他のターゲットとの相対的な重みを示す **0** から **255** の間の数字を入力します。デフォルトは **100** です。

7. **Target Port** を選択します。

8. **Create** をクリックします。

#### 8.4.5.1.3. CLI を使用した重みの管理

**手順**

1. サービスおよび対応する重みのルートによる負荷分散を管理するには、**oc set route-backends** コマンドを使用します。

```
$ 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 ab-example 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。

2. フラグなしのコマンドを実行して、現在の設定を確認します。

```
$ oc set route-backends ab-example
```

**出力例**

```
NAME                KIND     TO          WEIGHT
routes/ab-example   Service  ab-example-a 198 (99%)
routes/ab-example   Service  ab-example-b 2   (1%)
```

3. **--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。
以下の例では、**ab-example-a** および **ab-example-b** サービスの重みを変更します。

```
$ oc set route-backends ab-example --adjust ab-example-a=200 ab-example-b=10
```

または、パーセンテージを指定してサービスの重みを変更します。

```
$ oc set route-backends ab-example --adjust ab-example-b=5%
```

パーセンテージ宣言の前に **+** を指定すると、現在の設定に対して重み付けを調整できます。以下に例を示します。

```
$ oc set route-backends ab-example --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 ab-example --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ま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。

#### 8.4.5.1.4.1 サービス、複数の Deployment オブジェクト

**手順**

1. すべてのシャードに共通の **ab-example=true** ラベルを追加して新規アプリケーションを作成します。

```
$ oc new-app openshift/deployment-example --name=ab-example-a --as-deployment-config=true --labels=ab-example=true --env=SUBTITLE\=shardA
```

```
$ oc delete svc/ab-example-a
```

アプリケーションがデプロイされ、サービスが作成されます。これは最初のシャードです。

2. 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 deployment ab-example-a --name=ab-example --selector=ab-example\=true
```

```
$ oc expose service ab-example
```

3. **ab-example-<project_name>.<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。

4. 1 つ目のシャードと同じソースイメージおよびラベルに基づくが、別のバージョンがタグ付けされたバージョンと一意の環境変数を指定して 2 つ目のシャードを作成します。

```
$ oc new-app openshift/deployment-example:v2 \
    --name=ab-example-b --labels=ab-example=true \
    SUBTITLE="shard B" COLOR="red" --as-deployment-config=true
```

```
$ oc delete svc/ab-example-b
```

5. 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 つの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、以下を実行します。

a. **oc scale** コマンドを使用して、**ab-example-a** のレプリカを **0** に減らします。

```
$ oc scale dc/ab-example-a --replicas=0
```

ブラウザーを更新して、**v2** および **shard B** (赤) を表示させます。

b. **ab-example-a** を **1** レプリカに、**ab-example-b** を **0** にスケーリングします。

```
$ oc scale dc/ab-example-a --replicas=1; oc scale dc/ab-example-b --replicas=0
```

ブラウザーを更新して、**v1** および **shard A** (青) を表示します。

6. いずれかのシャードでデプロイメントをトリガーする場合、そのシャードの Pod のみが影響を受けます。どちらかの **Deployment** オブジェクトで **SUBTITLE** 環境変数を変更してデプロイメントをトリガーできます。

```
$ oc edit dc/ab-example-a
```

または

```
$ oc edit dc/ab-example-b
```

# 第9章 クォータ

## 9.1. プロジェクトごとのリソースクォータ

**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する可能性のあ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。

本書では、リソースクォータの仕組みや、クラスター管理者がリソースクォータはプロジェクトごとにどのように設定し、管理できるか、および開発者やクラスター管理者がそれらをどのように表示できるかについて説明します。

### 9.1.1. 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9.1 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9.2 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。 |
| **ephemeral-storage** | 非終了状態のすべての Pod におけるローカルの一時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。**ephemeral-storage** および **requests.ephemeral-storage** 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.ephemeral-storage** | 非終了状態のすべての Pod における一時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。**ephemeral-storage** および **requests.ephemeral-storage** 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.ephemeral-storage** | 非終了状態のすべての Pod における一時ストレージ制限の合計は、この値を超えることができません。 |

**表9.3 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 ReplicationController 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。 |
| **services.loadbalancers** | プロジェクトに存在できるタイプ **LoadBalancer** のサービスの合計数です。 |
| **services.nodeports** | プロジェクトに存在できるタイプ **NodePort** の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。 |

| リソース名 | 説明 |
| --- | --- |
| **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。 |

### 9.1.2. 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 |
| --- | --- |
| **BestEffort** | **cpu** または **memory** のいずれかについてのサービスの QoS (Quality of Service) が Best Effort の Pod に一致します。 |
| **NotBestEffort** | **cpu** および **memory** についてのサービスの QoS (Quality of Service) が Best Effort ではない Pod に一致します。 |

**BestEffort** スコープは、以下のリソースに制限するようにクォータを制限します。

- **pods**

**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**

### 9.1.3. 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。設定可能な時間を指定して、クォータ使用量の統計値を現在確認されるシステム値まで下げるのに必要な時間を決定します。

プロジェクト変更がクォータ使用制限を超える場合、サーバーはそのアクションを拒否し、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ユーザーに返されます。

### 9.1.4. 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、メモリー、一時ストレージのそれぞれに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。

### 9.1.5. リソースクォータ定義の例

**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
    services.loadbalancers: "2" 6
```

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。
6. プロジェクトに存在できるタイプ **LoadBalancer** の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サービスの 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 のいずれかのサービスの 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限り **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 終了状態の Pod の合計数です。

2 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 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)
    requests.ephemeral-storage: 2Gi (8)
    limits.ephemeral-storage: 4Gi (9)
```

(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。

(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。

(8) 非終了状態のすべての Pod において、一時ストレージ要求の合計は 2 Gi を超えることができません。

(9) 非終了状態のすべての Pod において、一時ストレージ制限の合計は 4 Gi を超えることができません。

### 9.1.6. クォータの作成

特定のプロジェクトでリソースの使用を制限するためにクォータを作成することができます。

**手順**

1. ファイルにクォータを定義します。
2. クォータを作成し、これをプロジェクトに適用するためにファイルを使用します。

-

```
$ oc create -f <file> [-n <project_name>]
```

以下に例を示します。

```
$ oc create -f core-object-counts.yaml -n demoproject
```

#### 9.1.6.1. オブジェクトカウントクォータの作成

**BuildConfig** および **DeploymentConfig** オブジェクトなどの、OpenShift Container Platform の標準的な namespace を使用しているリソースタイプのすべてにオブジェクトカウントクォータを作成できます。オブジェクトクォータカウントは、定義されたクォータをすべての標準的な namespace を使用しているリソースタイプに設定します。

リソースクォータを使用する際に、オブジェクトは作成時クォータに基づいてチャージされます。以下のクォータのタイプはリソースが使い切られることから保護するのに役立ちます。クォータは、プロジェクト内に余分なリソースが十分にある場合にのみ作成できます。

**手順**

リソースのオブジェクトカウントクォータを設定するには、以下を実行します。

1. 以下のコマンドを実行します。

```
$ oc create quota <name> \
    --hard=count/<resource>.<group>=<quota>,count/<resource>.<group>=<quota> 1
```

1 **<resource>** 変数はリソースの名前であり、**<group>** は API グループです (該当する場合)。リソースおよびそれらの関連付けられた API グループのリストに **oc api-resources** コマンドを使用します。

以下に例を示します。

```
$ oc create quota test \
    --
hard=count/deployments.extensions=2,count/replicasets.extensions=4,count/pods=3,count/secr
ets=4
```

**出力例**

```
resourcequota "test" created
```

この例では、リスト表示されたリソースをクラスター内の各プロジェクトのハード制限に制限します。

2. クォータが作成されていることを確認します。

```
$ oc describe quota test
```

**出力例**

```
Name:                   test
Namespace:              quota
```

```
Resource                    Used  Hard
--------                    ----  ----
count/deployments.extensions  0     2
count/pods                    0     3
count/replicasets.extensions  0     4
count/secrets                 0     4
```

#### 9.1.6.2. 拡張リソースのリソースクォータの設定

リソースのオーバーコミットは拡張リソースには許可されません。そのため、クォータで同じ拡張リソースについて **requests** および **limits** を指定する必要があります。現時点で、接頭辞 **requests.** のあるクォータ項目のみが拡張リソースに許可されます。以下は、GPU リソース **nvidia.com/gpu** のリソースクォータを設定する方法についてのシナリオ例です。

**手順**

1. クラスター内のノードで利用可能な GPU の数を判別します。以下に例を示します。

```
# oc describe node ip-172-31-27-209.us-west-2.compute.internal | egrep
'Capacity|Allocatable|gpu'
```

**出力例**

```
                    openshift.com/gpu-accelerator=true
Capacity:
 nvidia.com/gpu:  2
Allocatable:
 nvidia.com/gpu:  2
  nvidia.com/gpu  0           0
```

この例では、2 つの GPU が利用可能です。

2. **ResourceQuota** オブジェクトを作成して、namespace **nvidia** にクォータを設定します。この例では、クォータは **1** です。

**出力例**

```
apiVersion: v1
kind: ResourceQuota
metadata:
  name: gpu-quota
  namespace: nvidia
spec:
  hard:
    requests.nvidia.com/gpu: 1
```

3. クォータを作成します。

```
# oc create -f gpu-quota.yaml
```

**出力例**

```
resourcequota/gpu-quota created
```

4. namespace に正しいクォータが設定されていることを確認します。

```
# oc describe quota gpu-quota -n nvidia
```

**出力例**

```
Name:                   gpu-quota
Namespace:              nvidia
Resource                Used  Hard
--------                ----  ----
requests.nvidia.com/gpu  0     1
```

5. 単一 GPU を要求する Pod を定義します。以下の定義ファイルのサンプルの名前は **gpu-pod.yaml** です。

```
apiVersion: v1
kind: Pod
metadata:
  generateName: gpu-pod-
  namespace: nvidia
spec:
  restartPolicy: OnFailure
  containers:
  - name: rhel7-gpu-pod
    image: rhel7
    env:
      - name: NVIDIA_VISIBLE_DEVICES
        value: all
      - name: NVIDIA_DRIVER_CAPABILITIES
        value: "compute,utility"
      - name: NVIDIA_REQUIRE_CUDA
        value: "cuda>=5.0"
    command: ["sleep"]
    args: ["infinity"]
    resources:
      limits:
        nvidia.com/gpu: 1
```

6. Pod を作成します。

```
# oc create -f gpu-pod.yaml
```

7. Pod が実行されていることを確認します。

```
# oc get pods
```

**出力例**

```
NAME            READY     STATUS      RESTARTS   AGE
gpu-pod-s46h7     1/1       Running     0          1m
```

8. クォータ **Used** のカウンターが正しいことを確認します。

```
# oc describe quota gpu-quota -n nvidia
```

**出力例**

```
Name:                   gpu-quota
Namespace:              nvidia
Resource                Used  Hard
--------                ----  ----
requests.nvidia.com/gpu  1     1
```

9. **nvidia** namespace で 2 番目の GPU Pod の作成を試行します。2 つの GPU があるので、これをノード上で実行することは可能です。

```
# oc create -f gpu-pod.yaml
```

**出力例**

```
Error from server (Forbidden): error when creating "gpu-pod.yaml": pods "gpu-pod-f7z2w" is
forbidden: exceeded quota: gpu-quota, requested: requests.nvidia.com/gpu=1, used:
requests.nvidia.com/gpu=1, limited: requests.nvidia.com/gpu=1
```

クォータが 1 GPU であり、この Pod がそのクォータを超える 2 つ目の GPU の割り当てを試行したため、**Forbidden** エラーメッセージが表示されることが予想されます。

### 9.1.7. 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。

**手順**

1. プロジェクトで定義されるクォータのリストを取得します。たとえば、**demoproject** というプロジェクトの場合、以下を実行します。

```
$ oc get quota -n demoproject
```

**出力例**

```
NAME                     AGE    REQUEST
LIMIT
besteffort               4s     pods: 1/2
compute-resources-time-bound   10m    pods: 0/2
limits.cpu: 0/1, limits.memory: 0/1Gi
core-object-counts       109s   configmaps: 2/10, persistentvolumeclaims: 1/4,
replicationcontrollers: 1/20, secrets: 9/10, services: 2/10
```

2. 関連す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
```

**出力例**

```
Name:   core-object-counts
Namespace:  demoproject
Resource  Used Hard
--------  ---- ----
configmaps  3 10
persistentvolumeclaims 0 4
replicationcontrollers 3 20
secrets   9 10
services  2 10
```

### 9.1.8. 明示的なリソースクォータの設定

プロジェクト要求テンプレートで明示的なリソースクォータを設定し、新規プロジェクトに特定のリソースクォータを適用します。

**前提条件**

- cluster-admin ロールを持つユーザーとしてのクラスターへのアクセスがあること。
- OpenShift CLI (**oc**) がインストールされている。

**手順**

1. プロジェクト要求テンプレートにリソースクォータ定義を追加します。
   - プロジェクト要求テンプレートがクラスターに存在しない場合:
     a. ブートストラッププロジェクトテンプレートを作成し、これを **template.yaml** というファイルに出力します。

```
$ oc adm create-bootstrap-project-template -o yaml > template.yaml
```

     b. リソースクォータの定義を **template.yaml** に追加します。以下の例では、storage-consumption という名前のリソースクォータを定義します。テンプレートの **parameters:** セクションの前に定義を追加する必要があります。

```
- apiVersion: v1
  kind: ResourceQuota
  metadata:
    name: storage-consumption
    namespace: ${PROJECT_NAME}
  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。
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。この値が **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ません。
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。この値が **0** に設定される場合、bronze ストレージクラスは要求を作成できません。

c. **openshift-config** namespace の変更された **template.yaml** ファイルでプロジェクト要求テンプレートを作成します。

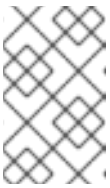
```
$ oc create -f template.yaml -n openshift-config
```

**注記**

設定を **kubectl.kubernetes.io/last-applied-configuration** アノテーションとして追加するには、 **--save-config** オプションを **oc create** コマンドに追加します。

デフォルトでは、テンプレートは **project-request** という名前になります。

- プロジェクト要求テンプレートがクラスター内にすでに存在する場合は、以下を実行します。



**注記**

設定ファイルを使用してクラスター内のオブジェクトを宣言的または命令的に管理する場合は、これらのファイルを使用して既存のプロジェクト要求テンプレートを編集します。

a. **openshift-config** namespace のテンプレートをリスト表示します。

```
$ oc get templates -n openshift-config
```

b. 既存のプロジェクト要求テンプレートを編集します。

```
$ oc edit template <project_request_template> -n openshift-config
```

c. 前述の **storage-consumption** の例などのリソースクォータ定義を既存のテンプレートに追加します。テンプレートの **parameters:** セクションの前に定義を追加する必要があります。

2. プロジェクト要求テンプレートを作成した場合は、クラスターのプロジェクト設定リソースでこれを参照します。

a. 編集するプロジェクト設定リソースにアクセスします。

- Web コンソールの使用

i. **Administration → Cluster Settings** ページに移動します。

ii. **Configuration** をクリックし、すべての設定リソースを表示します。

iii. **Project** のエントリーを見つけ、**Edit YAML** をクリックします。

- CLI の使用

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

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

b. プロジェクト設定リソースの **spec** セクションを更新し、**projectRequestTemplate** および **name** パラメーターを追加します。以下の例は、**project-request** というデフォルトのプロジェクト要求テンプレートを参照します。

```
apiVersion: config.openshift.io/v1
kind: Project
metadata:
#  ...
spec:
  projectRequestTemplate:
    name: project-request
```

3. プロジェクトの作成時にリソースクォータが適用されていることを確認します。

a. プロジェクトを作成します。

```
$ oc new-project <project_name>
```

b. プロジェクトのリソースクォータをリスト表示します。

```
$ oc get resourcequotas
```

c. リソースクォータを詳細に記述します。

```
$ oc describe resourcequotas <resource_quota_name>
```

## 9.2. 複数のプロジェクト間のリソースクォータ

**ClusterResourceQuota** オブジェクトで定義される複数プロジェクトのクォータは、複数プロジェクト間でクォータを共有できるようにします。それぞれの選択されたプロジェクトで使用されるリソースは集計され、その集計は選択したすべてのプロジェクトでリソースを制限するために使用されます。

以下では、クラスター管理者が複数のプロジェクトでリソースクォータを設定および管理する方法について説明します。

### 9.2.1. クォータ作成時の複数プロジェクトの選択

クォータの作成時に、アノテーションの選択、ラベルの選択、またはその両方に基づいて複数のプロジェクトを選択することができます。

**手順**

1. アノテーションに基づいてプロジェクトを選択するには、以下のコマンドを実行します。

```
$ oc create clusterquota for-user \
    --project-annotation-selector openshift.io/requester=<user_name> \
    --hard pods=10 \
    --hard secrets=20
```

これにより、以下の **ClusterResourceQuota** オブジェクトが作成されます。

```
apiVersion: quota.openshift.io/v1
kind: ClusterResourceQuota
metadata:
  name: for-user
spec:
  quota: 1
    hard:
      pods: "10"
      secrets: "20"
  selector:
    annotations: 2
      openshift.io/requester: <user_name>
    labels: null 3
status:
  namespaces: 4
  - namespace: ns-one
    status:
      hard:
        pods: "10"
        secrets: "20"
      used:
        pods: "1"
        secrets: "9"
  total: 5
    hard:
      pods: "10"
      secrets: "20"
    used:
      pods: "1"
      secrets: "9"
```

1. 選択されたプロジェクトに対して実施される **ResourceQuotaSpec** オブジェクトです。
2. アノテーションの単純なキー/値のセレクターです。

3 プロジェクトを選択するために使用できるラベルセレクターです。

4 選択された各プロジェクトの現在のクォータの使用状況を記述する namespace ごとのマップです。

5 選択されたすべてのプロジェクトにおける使用量の総計です。

この複数プロジェクトのクォータの記述は、デフォルトのプロジェクト要求エンドポイントを使用して **<user_name>** によって要求されるすべてのプロジェクトを制御します。ここでは、10 Pod および 20 シークレットに制限されます。

2. 同様にラベルに基づいてプロジェクトを選択するには、以下のコマンドを実行します。

```
$ oc create clusterresourcequota for-name \ 1
    --project-label-selector=name=frontend \ 2
    --hard=pods=10 --hard=secrets=20
```

1 **clusterresourcequota** および **clusterquota** はいずれも同じコマンドのエイリアスです。**for-name** は **ClusterResourceQuota** オブジェクトの名前です。

2 ラベル別にプロジェクトを選択するには、**--project-label-selector=key=value** 形式を使用してキーと値のペアを指定します。

これにより、以下の **ClusterResourceQuota** オブジェクト定義が作成されます。

```
apiVersion: quota.openshift.io/v1
kind: ClusterResourceQuota
metadata:
  creationTimestamp: null
  name: for-name
spec:
  quota:
    hard:
      pods: "10"
      secrets: "20"
  selector:
    annotations: null
    labels:
      matchLabels:
        name: frontend
```

### 9.2.2. 該当するクラスターリソースクォータの表示

プロジェクト管理者は、各自のプロジェクトを制限する複数プロジェクトのクォータを作成したり、変更したりすることはできませんが、それぞれのプロジェクトに適用される複数プロジェクトのクォータを表示することはできます。プロジェクト管理者は、**AppliedClusterResourceQuota** リソースを使用してこれを実行できます。

**手順**

1. プロジェクトに適用されているクォータを表示するには、以下を実行します。

```
$ oc describe AppliedClusterResourceQuota
```

**出力例**

```
Name:   for-user
Namespace:  <none>
Created:  19 hours ago
Labels:   <none>
Annotations:  <none>
Label Selector: <null>
AnnotationSelector: map[openshift.io/requester:<user-name>]
Resource  Used  Hard
--------  ----  ----
pods        1     10
secrets     9     20
```

### 9.2.3. 選択における粒度

クォータの割り当てを要求する際にロックに関して考慮する必要があるため、複数プロジェクトのクォータで選択されるアクティブなプロジェクトの数は重要な考慮点になります。単一の複数プロジェクトクォータで 100 を超えるプロジェクトを選択すると、それらのプロジェクトの API サーバーの応答に負の影響が及ぶ可能性があります。

# 第10章 アプリケーションでの設定マップの使用

設定マップにより、設定アーティファクトをイメージコンテンツから切り離し、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ことができます。

以下のセクションでは、設定マップおよびそれらを作成し、使用する方法を定義します。

## 10.1. 設定マップについて

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。OpenShift Container Platform では、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されます。

**ConfigMap** オブジェクトは、コンテナーを OpenShift Container Platform に依存させないようにする一方で、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。設定マップは、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** 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ポーネント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。以下に例を示します。

**ConfigMap オブジェクト定義**

```
kind: ConfigMap
apiVersion: v1
metadata:
  creationTimestamp: 2016-02-18T19:14:38Z
  name: example-config
  namespace: my-namespace
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
binaryData:
  bar: L3Jvb3QvMTAw 2
```

1 設定データが含まれます。

2 バイナリー Java キーストアファイルなどの UTF8 以外のデータを含むファイルを参照します。Base 64 のファイルデータを入力します。

**注記**

イメージなどのバイナリーファイルから設定マップを作成する場合に、**binaryData** フィールドを使用できます。

設定データはさまざまな方法で Pod 内で使用できます。設定マップは以下を実行するために使用できます。

- コンテナーへの環境変数値の設定
- コンテナーのコマンドライン引数の設定
- 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設定マップに保存できます。

設定マップ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。

**設定マップの制限**
**設定マップは、コンテンツを Pod で使用される前に作成する必要があります。**

コントローラーは、設定データが不足していても、その状況を許容して作成できます。ケースごとに設定マップを使用して設定される個々のコンポーネントを参照してください。

**ConfigMap オブジェクトはプロジェクト内にあります。**

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 の設定マップ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)。

**関連情報**

- 設定マップの作成および使用

## 10.2. ユースケース: POD で設定マップを使用する

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。

### 10.2.1. 設定マップの使用によるコンテナーでの環境変数の設定

config map を使用して、コンテナーで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を使用してコンテナーで環境変数を設定するために使用したりすることができます。

例として、以下の設定マップ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 2
data:
  special.how: very 3
  special.type: charm 4
```

–

1 設定マップの名前。

2 設定マップが存在するプロジェクト。設定マップは同じプロジェクトの Pod によってのみ参照されます。

3 4 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 設定マップの名前。

2 挿入する環境変数。

**手順**

- **configMapKeyRef** セクションを使用して、Pod のこの **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 2
          valueFrom:
            configMapKeyRef:
              name: special-config 3
              key: special.how 4
        - name: SPECIAL_TYPE_KEY
          valueFrom:
            configMapKeyRef:
              name: special-config 5
              key: special.type 6
              optional: true 7
      envFrom: 8
```

```
      - configMapRef:
          name: env-config 9
  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. キーの値を挿入する Pod 環境変数の名前です。
3. 5 特定の環境変数のプルに使用する **ConfigMap** の名前です。
4. 6 **ConfigMap** からプルする環境変数です。
7. 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。
8. **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。
9. 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、Pod のログには以下の出力が含まれます。

```
SPECIAL_LEVEL_KEY=very
log_level=INFO
```

**注記**

**SPECIAL_TYPE_KEY=charm** は出力例にリスト表示されません。**optional: true** が設定されているためです。

### 10.2.2. 設定マップを使用したコンテナーコマンドのコマンドライン引数の設定

config map を使用すると、Kubernetes 置換構文 **$(VAR_NAME)** を使用してコンテナー内のコマンドまたは引数の値を設定できます。

例として、以下の設定マップ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
```

**手順**

- コンテナー内のコマンドに値を挿入するには、環境変数として使用す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)" ]
1
      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
```

1 環境変数として使用するキーを使用して、コンテナーのコマンドに値を挿入します。

この Pod が実行されると、test-container コンテナーで実行される echo コマンドの出力は以下のようになります。

```
very charm
```

### 10.2.3. 設定マップの使用によるボリュームへのコンテンツの挿入

設定マップを使用して、コンテンツをボリュームに挿入することができます。

**ConfigMap カスタムリソース (CR) の例**

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: special-config
  namespace: default
data:
  special.how: very
  special.type: charm
```

**手順**

設定マップを使用してコンテンツをボリュームに挿入するには、2 つの異なるオプションを使用できます。

- 設定マップを使用してコンテンツをボリュームに挿入するための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", "-c", "cat", "/etc/config/special.how" ]
      volumeMounts:
      - name: config-volume
        mountPath: /etc/config
  volumes:
    - name: config-volume
      configMap:
        name: special-config 1
  restartPolicy: Never
```

1 キーを含むファイル。

この Pod が実行されると、cat コマンドの出力は以下のようになります。

```
very
```

- 設定マップキーが投影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", "-c", "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 1
  restartPolicy: Never
```

1 設定マップキーへのパス。

この Pod が実行されると、cat コマンドの出力は以下のようになります。

```
very
```

# 第11章 開発者パースペクティブを使用したプロジェクトおよびアプリケーションメトリクスのモニタリング

Developer パースペクティブの Observe ビューは、CPU、メモリー、帯域幅の使用状況、ネットワーク関連の情報などのプロジェクトまたはアプリケーションのメトリクスを監視するオプションを提供します。

## 11.1. 前提条件

- OpenShift Container Platform にアプリケーションを作成し、デプロイ している。
- Web コンソールにログイン し、Developer パースペクティブ に切り替えている。

## 11.2. プロジェクトメトリクスのモニタリング

プロジェクトでアプリケーションを作成し、それらをデプロイした後に、Web コンソールで Developer パースペクティブを使用し、プロジェクトのメトリックを表示できます。

**手順**

1. Observe に移動して、プロジェクトの Dashboard、Metrics、Alerts、および Events を表示します。
2. オプション: Dashboard タブを使用して、次のアプリケーションメトリックを示すグラフを表示します:
   - CPU usage (CPU の使用率)
   - メモリー使用量
   - 帯域幅の使用
   - 送受信パケットのレートやドロップされたパケットのレートなど、ネットワーク関連の情報。

   Dashboard タブで、Kubernetes コンピュートリソースダッシュボードにアクセスできます。

   **注記**

   Dashboard リストでは、デフォルトで Kubernetes / Compute Resources / Namespace (Pods) ダッシュボードが選択されています。

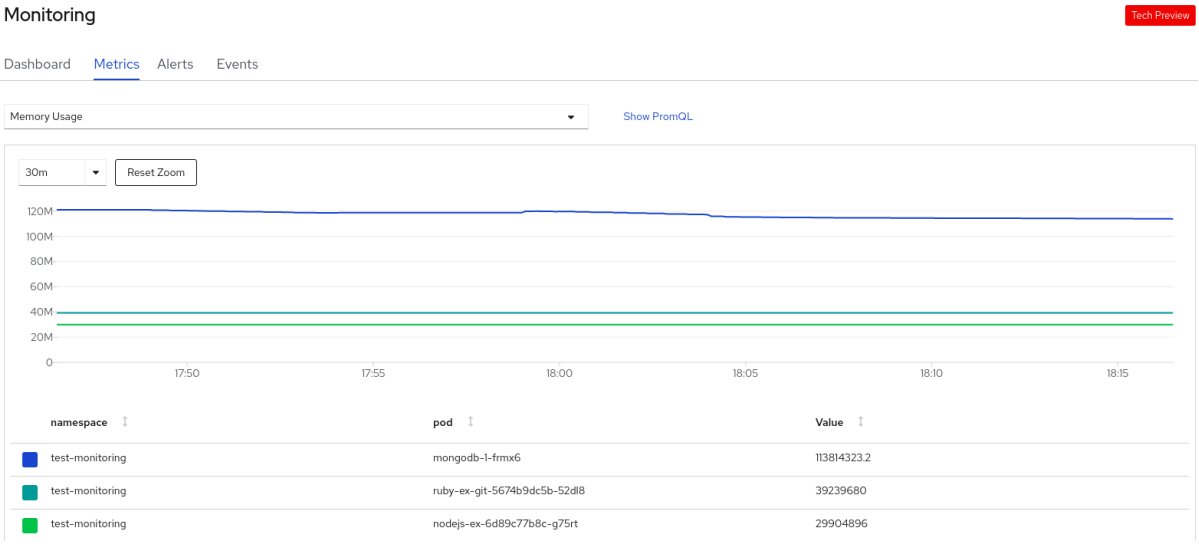
   詳細は、以下のオプションを使用します。

   - Dashboard リストからダッシュボードを選択し、フィルタリングされたメトリクスを表示します。すべてのダッシュボードは、Kubernetes / Compute Resources / Namespace(Pod) を除く、選択時に追加のサブメニューを生成します。
   - Time Range 一覧からオプションを選択し、キャプチャーされるデータの期間を判別します。
   - Time Range リストで Custom time range を選択して、カスタムの時間範囲を設定します。From および To の日付と時間を入力または選択します。Save をクリックして、カスタムの時間範囲を保存します。

- **Refresh Interval** 一覧からオプションを選択し、データの更新後の期間を判別します。
- カーソルをグラフの上に置き、Pod の特定の詳細を表示します。
- 各グラフの右上隅にある **Inspect** をクリックして、特定のグラフの詳細を表示します。グラフの詳細は **Metrics** タブに表示されます。

3. オプション: **Metrics** タブを使用して、必要なプロジェクトメトリックについてクエリーします。

**図11.1 メトリクスのモニタリング**




a. **Select Query** リストで、プロジェクトに必要な詳細をフィルターするオプションを選択します。プロジェクト内のすべてのアプリケーション Pod のフィルターされたメトリックがグラフに表示されます。プロジェクトの Pod も以下に記載されています。

b. Pod のリストから色の付いた四角のボックスをクリアし、特定の Pod のメトリックを削除してクエリーの結果をさらに絞り込みます。

c. **Show PromQL** をクリックし、 Prometheus クエリーを表示します。このクエリーをプロンプトのヘルプを使用してさらに変更し、クエリーをカスタマイズして、該当する namespace に表示するメトリックをフィルターすることができます。

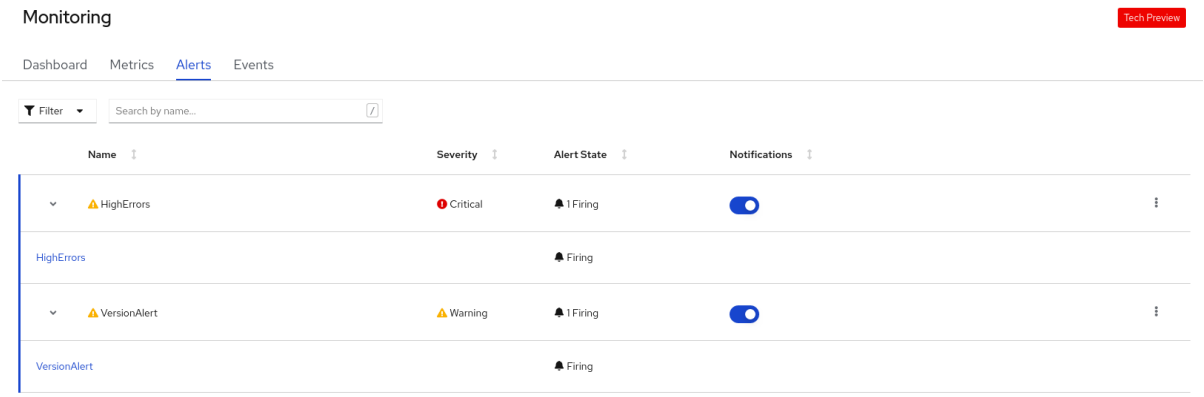
d. ドロップダウンリストを使用して、表示されるデータの時間の範囲を設定します。**Reset Zoom** をクリックして、これをデフォルトの時間の範囲にリセットできます。

e. オプションで、**Select Query** 一覧で **Custom Query** を選択し、カスタム Prometheus クエリーを作成し、関連するメトリクスをフィルターします。

4. オプション: **Alerts** タブを使用して、次のタスクを実行します:

- プロジェクト内のアプリケーションのアラートをトリガーするルールを確認します。
- プロジェクトで発生しているアラートを特定します。
- 必要に応じて、そのようなアラートを解除します。

**図11.2 アラートのモニタリング**

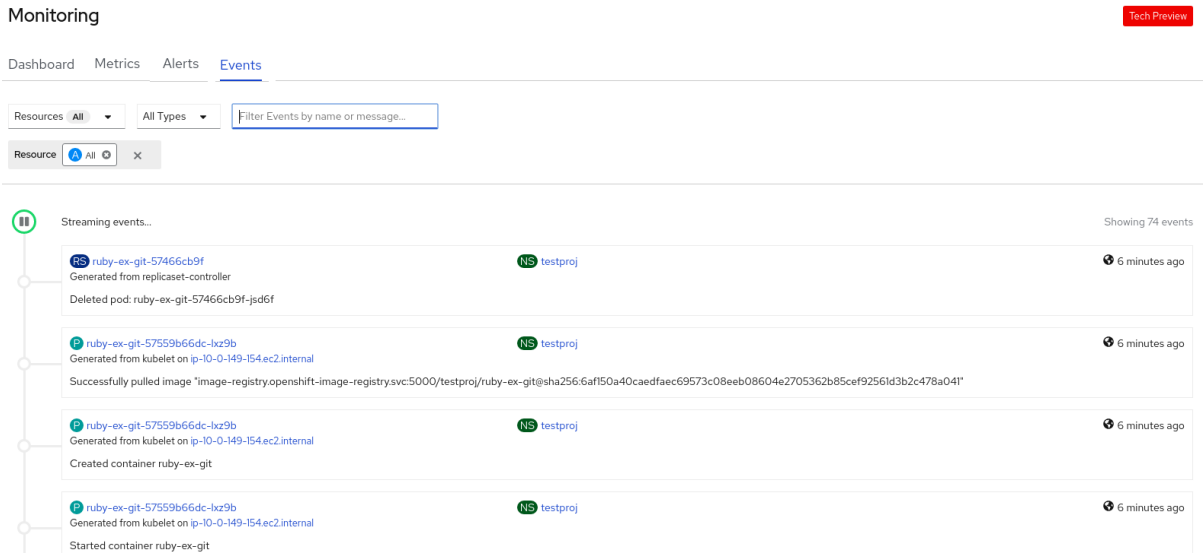



詳細は、以下のオプションを使用します。

- **Filter** 一覧を使用して **Alert State** および **Severity** でアラートをフィルターします。
- アラートをクリックして、そのアラートの詳細ページに移動します。**Alerts Details** ページで、**View Metrics** をクリックし、アラートのメトリクスを表示できます。
- アラートルールに隣接する **Notifications** トグルを使用して、そのルールのすべてのアラートをサイレンスにし、**Silence for** 一覧からアラートをサイレンスにする期間を選択します。**Notifications** トグルを表示するには、アラートを編集するパーミッションが必要です。
- アラートルールに隣接する **Options** メニュー を使用して、アラートルールの詳細を表示します。

5. オプション: **Events** タブを使用してプロジェクトのイベントを表示します。

**図11.3 イベントのモニタリング**




以下のオプションを使用して、表示されるイベントをフィルターできます。

- **Resources** リストで、リソースを選択し、そのリソースのイベントを表示します。
- **All Types** リストで、イベントのタイプを選択し、そのタイプに関連するイベントを表示します。

- **Filter events by names or messages** フィールドを使用して特定のイベントを検索します。

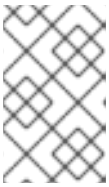
## 11.3. アプリケーションメトリクスのモニタリング

プロジェクトでアプリケーションを作成し、それらをデプロイした後に、**Developer** パースペクティブで **Topology** ビューを使用し、アプリケーションのアラートおよびメトリックを表示できます。アプリケーションの重大な問題および警告のアラートは、**Topology** ビューでワークロードノードについて示されます。

### 手順

ワークロードのアラートを表示するには、以下を実行します。

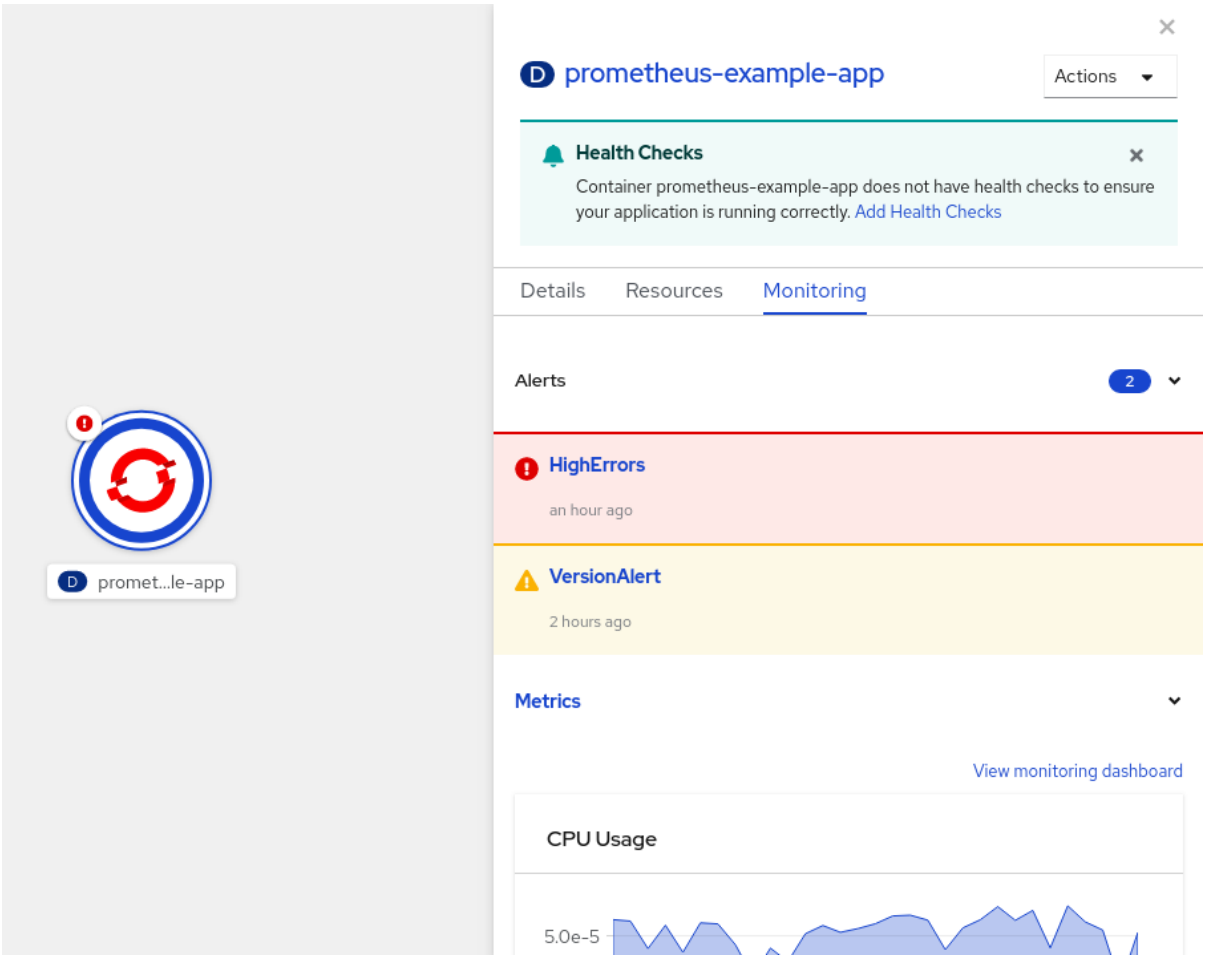
1. **Topology** ビューで、ワークロードをクリックし、ワークロードの詳細を右側のパネルに表示します。
2. **Observe** タブをクリックして、アプリケーションの重大な問題および警告のアラート、CPU、メモリー、および帯域幅の使用状況などのメトリクスのグラフ、およびアプリケーションのすべてのイベントを表示します。



**注記**

**Firing** 状態の重大な問題および警告のアラートのみが **Topology** ビューに表示されます。**Silenced**、**Pending** および **Not Firing** 状態のアラートは表示されません。

**図11.4 アプリケーションメトリクスのモニタリング**



a. 右側のパネルにリスト表示されるアラートをクリックし、アラートの詳細を **Alert Details** ページに表示します。

b. チャートのいずれかをクリックして **Metrics** タブに移動し、アプリケーションの詳細なメトリックを表示します。

c. **View monitoring dashboard** をクリックし、そのアプリケーションのモニタリングダッシュボードを表示します。

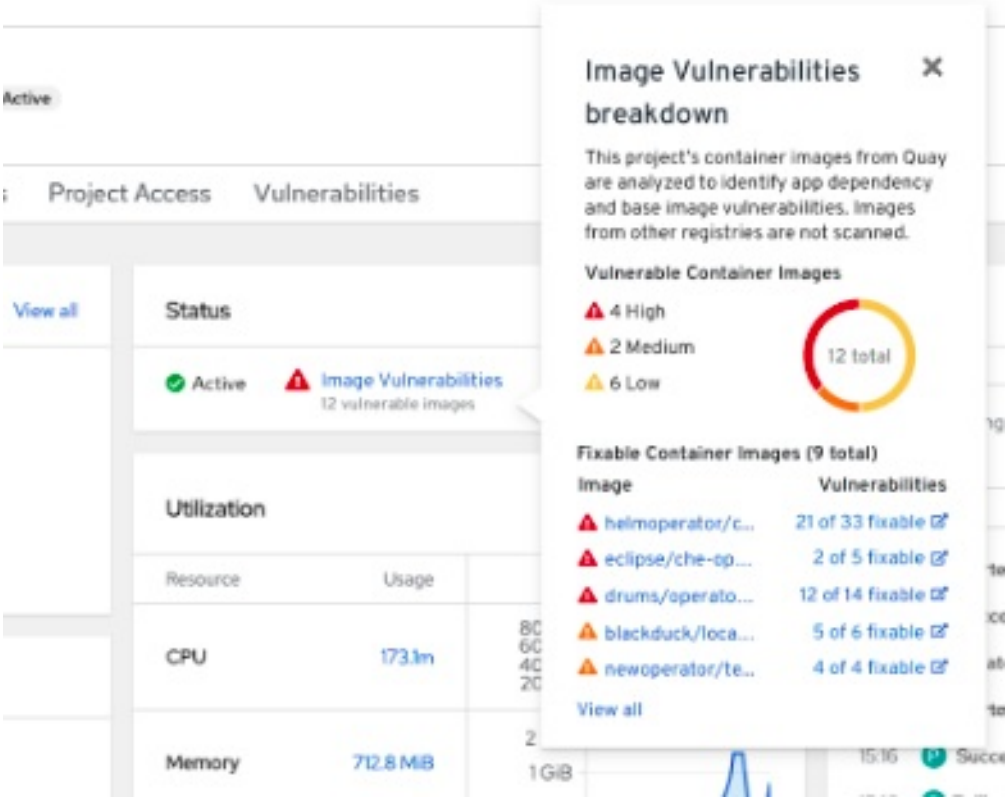
## 11.4. イメージの脆弱性の内訳

**Developer** パースペクティブでは、プロジェクトダッシュボードの **Status** セクションに **Image Vulnerabilities** リンクが表示されます。このリンクを使用すると、脆弱なコンテナーイメージと修正可能なコンテナーイメージに関する詳細を含む、**Image Vulnerabilities breakdown** ウィンドウを表示できます。アイコンの色は重大度を示します。

- 赤: 高優先度。すぐに修正してください。
- オレンジ: 中優先度。優先度の高い脆弱性の後に修正できます。
- 黄色: 低優先度。高優先度および中優先度の脆弱性の後に修正できます。

重大度レベルに基づいて、脆弱性に優先順位を付け、系統立てて修正できます。

**図11.5 イメージ脆弱性の表示**



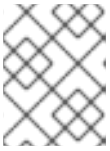

## 11.5. アプリケーションとイメージの脆弱性メトリックの監視

プロジェクトでアプリケーションを作成してデプロイしたら、Web コンソールの **Developer** パースペクティブを使用して、クラスター全体におけるアプリケーションの依存関係の脆弱性に関するメトリックを表示します。メトリックは、次のイメージの脆弱性を詳しく分析するのに役立ちます。

- 選択したプロジェクト内の脆弱なイメージの総数
- 選択したプロジェクト内のすべての脆弱なイメージの重大度別の数
- 脆弱性の数、修正可能な脆弱性の数、各脆弱なイメージの影響を受ける Pod の数など、重大度をドリルダウンした詳細

**前提条件**

- Operator Hub から Red Hat Quay Container SecurityOperator をインストールしている。



**注記**

Red Hat Quay Container Security Operator は、quay レジストリーにあるイメージをスキャンして脆弱性を検出します。

**手順**

1. イメージの脆弱性の一般的な概要については、**Developer** パースペクティブのナビゲーションパネルで **Project** をクリックして、プロジェクトダッシュボードを表示します。
2. **Status** セクションで **Image Vulnerabilities** をクリックします。開いたウィンドウには、**Vulnerable Container Images** や **Fixable Container Images** などの詳細が表示されます。
3. 脆弱性の詳細な概要については、プロジェクトダッシュボードの **Vulnerabilities** タブをクリックしてください。
   a. イメージの詳細を表示するには、その名前をクリックします。
   b. **Details** タブで、すべてのタイプの脆弱性のデフォルトグラフを表示します。
   c. オプション: 切り替えボタンをクリックして、特定のタイプの脆弱性を表示します。たとえば、**App dependency** をクリックすると、アプリケーションの依存関係に固有の脆弱性が表示されます。
   d. オプション: **Severity** および **Type** に基づき脆弱性一覧をフィルタリングするか、**Severity**、**Package**、**Type**、**Source**、**Current Version**、**Fixed in Version** でソートできます。
   e. **Vulnerability** をクリックして、関連する詳細を取得します。
      - **Base image** の脆弱性には、Red Hat Security Advisory (RHSA) からの情報が表示されます。
      - **App dependency** の脆弱性には、Snyk セキュリティーアプリケーションからの情報が表示されます。

## 11.6. 関連情報

- モニタリングの概要

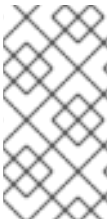
# 第12章 ヘルスチェックの使用によるアプリケーションの正常性の監視

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。

## 12.1. ヘルスチェックについて

ヘルスチェックは、readiness、liveness、および startup ヘルスチェックの組み合わせを使用して、実行中のコンテナーで診断を定期的に実行します。

ヘルスチェックを実行するコンテナーが含まれる Pod の仕様に、1 つ以上のプローブを含めることができます。



**注記**

既存の Pod でヘルスチェックを追加または編集する必要がある場合、Pod の **DeploymentConfig** オブジェクトを編集するか、Web コンソールで **Developer** パースペクティブを使用する必要があります。CLI を使用して既存の Pod のヘルスチェックを追加したり、編集したりすることはできません。

**readiness プローブ**

**readiness プローブ** はコンテナーがサービス要求を受け入れることができるかどうかを判別します。コンテナーの readiness プローブが失敗すると、kubelet は利用可能なサービスエンドポイントのリストから Pod を削除します。
失敗後、プローブは Pod の検証を継続します。Pod が利用可能になると、kubelet は Pod を利用可能なサービスエンドポイントのリストに追加します。

**liveness ヘルスチェック**

**liveness プローブ** は、コンテナーが実行中かどうかを判別します。デッドロックなどの状態のために liveness プローブが失敗する場合、kubelet はコンテナーを強制終了します。その後、Pod は再起動ポリシーに基づいて応答します。
たとえば、**restartPolicy** として **Always** または **OnFailure** が設定されている Pod での liveness プローブは、コンテナーを強制終了してから再起動します。

**スタートアッププローブ**

**スタートアッププローブ** は、コンテナー内のアプリケーションが起動しているかどうかを示します。その他のプローブはすべて、起動に成功するまで無効にされます。スタートアッププローブが指定の期間内に成功しない場合、kubelet はコンテナーを強制終了し、コンテナーは Pod の **restartPolicy** の対象となります。
一部のアプリケーションでは、最初の初期化時に追加の起動時間が必要になる場合があります。liveness または readiness プローブで startup プローブを使用して、**failureThreshold** および **periodSeconds** パラメーターを使用し、長い起動時間に十分に対応できるようにプローブを遅延させることができます。

たとえば、**failureThreshold** が 30 回 (30 failure) で、 **periodSeconds** が 10 秒の最大 5 分 (30 * 10s = 300s) を指定して startup プローブを liveness プローブに追加できます。startup プローブが初回に成功すると、liveness プローブがこれを引き継ぎます。

以下のテストのタイプのいずれかを使用して、liveness、readiness、および startup プローブを設定できます。

- HTTP **GET**: HTTP **GET** テストを使用する場合、テストは Web hook を使用してコンテナーの正常性を判別します。このテストは、HTTP の応答コードが **200** から **399** までの値の場合に正常と見なされます。
  完全に初期化されている場合に、HTTP ステータスコードを返すアプリケーションで HTTP **GET** テストを使用できます。

- コンテナーコマンド: コンテナーコマンドテストを使用すると、プローブはコンテナー内でコマンドを実行します。テストが **0** のステータスで終了すると、プローブは成功します。

- TCP ソケット: TCP ソケットテストを使用する場合、プローブはコンテナーに対してソケットを開こうとします。コンテナーはプローブ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。TCP ソケットテストは、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で使用できます。

複数のフィールドを設定して、プローブの動作を制御できます。

- **initialDelaySeconds**: コンテナーが起動してからプローブがスケジュールされるまでの時間 (秒単位)。デフォルトは 0 です。

- **periodSeconds**: プローブの実行間の遅延 (秒単位)。デフォルトは **10** です。この値は **timeoutSeconds** よりも大きくなければなりません。

- **timeoutSeconds**: プローブがタイムアウトし、コンテナーが失敗した想定されてから非アクティブになるまでの時間 (秒数)。デフォルトは **1** です。この値は **periodSeconds** 未満である必要があります。

- **successThreshold**: コンテナーのステータスを successful にリセットするために、プローブが失敗後に成功を報告する必要のある回数。liveness プローブの場合は、値は **1** である必要があります。デフォルトは **1** です。

- **failureThreshold**: プローブが失敗できる回数。デフォルトは 3 です。指定される試行の後に、以下を実行します。

  - liveness プローブの場合、コンテナーが再起動します。

  - readiness プローブの場合、Pod には **Unready** というマークが付けられます。

  - startup プローブの場合、コンテナーは強制終了され、Pod の **restartPolicy** の対象となります。

### プローブの例

以下は、オブジェクト仕様に表示されるさまざまなプローブの例です。

#### Pod 仕様のコンテナーコマンド readiness プローブを含む readiness プローブの例

```
apiVersion: v1
kind: Pod
metadata:
  labels:
    test: health-check
  name: my-application
# ...
spec:
  containers:
```

```
  - name: goproxy-app 1
    args:
    image: registry.k8s.io/goproxy:0.1 2
    readinessProbe: 3
      exec: 4
        command: 5
        - cat
        - /tmp/healthy
# ...
```

1 コンテナー名。

2 デプロイするコンテナーイメージ。

3 readiness プローブ

4 コンテナーコマンドのテスト。

5 コンテナーで実行するコマンド。

### Pod 仕様のコンテナーコマンドテストを含むコンテナーコマンドの startup プローブおよび liveness プローブの例

```
apiVersion: v1
kind: Pod
metadata:
  labels:
    test: health-check
  name: my-application
# ...
spec:
  containers:
  - name: goproxy-app 1
    args:
    image: registry.k8s.io/goproxy:0.1 2
    livenessProbe: 3
      httpGet: 4
        scheme: HTTPS 5
        path: /healthz
        port: 8080 6
        httpHeaders:
        - name: X-Custom-Header
          value: Awesome
    startupProbe: 7
      httpGet: 8
        path: /healthz
        port: 8080 9
      failureThreshold: 30 10
      periodSeconds: 10 11
# ...
```

1 コンテナー名。

2 デプロイするコンテナーイメージを指定します。

3 liveness プローブ

4 HTTP **GET** テスト。

5 インターネットスキーム: **HTTP** または **HTTPS**デフォルト値は **HTTP** です。

6 コンテナーがリッスンしているポート。

7 スタートアッププローブ。

8 HTTP **GET** テスト。

9 コンテナーがリッスンしているポート。

10 失敗後にプローブを試行する回数。

11 プローブを実行する秒数。

**Pod 仕様でタイムアウトを使用するコンテナーコマンドテストを使用した liveness プローブの例**

```
apiVersion: v1
kind: Pod
metadata:
  labels:
    test: health-check
  name: my-application
# ...
spec:
  containers:
  - name: goproxy-app 1
    args:
    image: registry.k8s.io/goproxy:0.1 2
    livenessProbe: 3
      exec: 4
        command: 5
        - /bin/bash
        - '-c'
        - timeout 60 /opt/eap/bin/livenessProbe.sh
      periodSeconds: 10 6
      successThreshold: 1 7
      failureThreshold: 3 8
# ...
```

1 コンテナー名。

2 デプロイするコンテナーイメージを指定します。

3 liveness プローブ。

4 プローブのタイプ。この場合はコンテナーコマンドプローブです。

5 コンテナー内で実行するコマンドライン。

6 プローブを実行する頻度 (秒単位)。

7 失敗後の成功を示すために必要な連続する成功の数。

8 失敗後にプローブを試行する回数。

**デプロイメントでの TCP ソケットテストを含む readiness プローブおよび liveness プローブの例**

```
kind: Deployment
apiVersion: apps/v1
metadata:
  labels:
    test: health-check
  name: my-application
spec:
# ...
  template:
    spec:
      containers:
        - resources: {}
          readinessProbe: 1
            tcpSocket:
              port: 8080
            timeoutSeconds: 1
            periodSeconds: 10
            successThreshold: 1
            failureThreshold: 3
          terminationMessagePath: /dev/termination-log
          name: ruby-ex
          livenessProbe: 2
            tcpSocket:
              port: 8080
            initialDelaySeconds: 15
            timeoutSeconds: 1
            periodSeconds: 10
            successThreshold: 1
            failureThreshold: 3
# ...
```

1 readiness プローブ。

2 liveness プローブ。

## 12.2. CLI を使用したヘルスチェックの設定

readiness、liveness、および startup プローブを設定するには、1 つ以上のプローブをヘルスチェックを実行するコンテナーが含まれる Pod の仕様に追加します。

**注記**

既存の Pod でヘルスチェックを追加または編集する必要がある場合、Pod の **DeploymentConfig** オブジェクトを編集するか、Web コンソールで **Developer** パースペクティブを使用する必要があります。CLI を使用して既存の Pod のヘルスチェックを追加したり、編集したりすることはできません。

**手順**

コンテナーのプローブを追加するには、以下を実行します。

1. **Pod** オブジェクトを作成して、1 つ以上のプローブを追加します。

```
apiVersion: v1
kind: Pod
metadata:
  labels:
    test: health-check
  name: my-application
spec:
  containers:
  - name: my-container (1)
    args:
    image: registry.k8s.io/goproxy:0.1 (2)
    livenessProbe: (3)
      tcpSocket: (4)
        port: 8080 (5)
      initialDelaySeconds: 15 (6)
      periodSeconds: 20 (7)
      timeoutSeconds: 10 (8)
    readinessProbe: (9)
      httpGet: (10)
        host: my-host (11)
        scheme: HTTPS (12)
        path: /healthz
        port: 8080 (13)
    startupProbe: (14)
      exec: (15)
        command: (16)
        - cat
        - /tmp/healthy
      failureThreshold: 30 (17)
      periodSeconds: 20 (18)
      timeoutSeconds: 10 (19)
```

(1) コンテナー名を指定します。

(2) デプロイするコンテナーイメージを指定します。

(3) オプション: liveness プローブを作成します。

(4) 実行するテストを指定します。この場合は TCP ソケットテストです。

5 コンテナーがリッスンするポートを指定します。

6 コンテナーが起動してからプローブがスケジュールされるまでの時間 (秒単位) を指定します。

7 プローブを実行する秒数を指定します。デフォルトは **10** です。この値は **timeoutSeconds** よりも大きくなければなりません。

8 プローブが失敗したと想定されてから非アクティブになる時間 (秒数)。デフォルトは **1** です。この値は **periodSeconds** 未満である必要があります。

9 オプション: readiness プローブを作成します。

10 実行するテストのタイプを指定します。この場合は HTTP テストです。

11 ホストの IP アドレスを指定します。**host** が定義されていない場合は、**PodIP** が使用されます。

12 **HTTP** または **HTTPS** を指定します。**scheme** が定義されていない場合は、**HTTP** スキームが使用されます。

13 コンテナーがリッスンするポートを指定します。

14 オプション: スタートアッププローブを作成します。

15 実行するテストのタイプを指定します。この場合はコンテナー実行プローブです。

16 コンテナーで実行するコマンドを指定します。

17 失敗後にプローブを試行する回数を指定します。

18 プローブを実行する秒数を指定します。デフォルトは **10** です。この値は **timeoutSeconds** よりも大きくなければなりません。

19 プローブが失敗したと想定されてから非アクティブになる時間 (秒数)。デフォルトは **1** です。この値は **periodSeconds** 未満である必要があります。

**注記**

**initialDelaySeconds** 値が **periodSeconds** 値よりも低い場合、最初の readiness プローブがタイマーの問題により 2 つの期間の間のある時点で生じます。

**timeoutSeconds** 値は **periodSeconds** の値よりも低い値である必要があります。

2. **Pod** オブジェクトを作成します。

```
$ oc create -f <file-name>.yaml
```

3. ヘルスチェック Pod の状態を確認します。

```
$ oc describe pod my-application
```

**出力例**

```
-
```

```
Events:
  Type    Reason     Age   From                           Message
  ----    ------     ----  ----                           -------
  Normal  Scheduled  9s    default-scheduler              Successfully assigned openshift-
logging/liveness-exec to ip-10-0-143-40.ec2.internal
  Normal  Pulling    2s    kubelet, ip-10-0-143-40.ec2.internal  pulling image
"registry.k8s.io/liveness"
  Normal  Pulled     1s    kubelet, ip-10-0-143-40.ec2.internal  Successfully pulled image
"registry.k8s.io/liveness"
  Normal  Created    1s    kubelet, ip-10-0-143-40.ec2.internal  Created container
  Normal  Started    1s    kubelet, ip-10-0-143-40.ec2.internal  Started container
```

以下は、コンテナーを再起動した障害のあるプローブの出力です。

**正常ではないコンテナーについての liveness チェック出力の例**

```
$ oc describe pod pod1
```

**出力例**

```
....

Events:
  Type     Reason          Age               From                                        Message
  ----     ------          ----              ----                                        -------
  Normal   Scheduled       <unknown>                                                     Successfully
assigned aaa/liveness-http to ci-ln-37hz77b-f76d1-wdpjv-worker-b-snzrj
  Normal   AddedInterface  47s               multus                                      Add eth0
[10.129.2.11/23]
  Normal   Pulled          46s               kubelet, ci-ln-37hz77b-f76d1-wdpjv-worker-b-snzrj
Successfully pulled image "registry.k8s.io/liveness" in 773.406244ms
  Normal   Pulled          28s               kubelet, ci-ln-37hz77b-f76d1-wdpjv-worker-b-snzrj
Successfully pulled image "registry.k8s.io/liveness" in 233.328564ms
  Normal   Created         10s (x3 over 46s)  kubelet, ci-ln-37hz77b-f76d1-wdpjv-worker-b-snzrj
Created container liveness
  Normal   Started         10s (x3 over 46s)  kubelet, ci-ln-37hz77b-f76d1-wdpjv-worker-b-snzrj
Started container liveness
  Warning  Unhealthy       10s (x6 over 34s)  kubelet, ci-ln-37hz77b-f76d1-wdpjv-worker-b-
snzrj  Liveness probe failed: HTTP probe failed with statuscode: 500
  Normal   Killing         10s (x2 over 28s)  kubelet, ci-ln-37hz77b-f76d1-wdpjv-worker-b-snzrj
Container liveness failed liveness probe, will be restarted
  Normal   Pulling         10s (x3 over 47s)  kubelet, ci-ln-37hz77b-f76d1-wdpjv-worker-b-snzrj
Pulling image "registry.k8s.io/liveness"
  Normal   Pulled          10s               kubelet, ci-ln-37hz77b-f76d1-wdpjv-worker-b-snzrj
Successfully pulled image "registry.k8s.io/liveness" in 244.116568ms
```

## 12.3. DEVELOPER パースペクティブを使用したアプリケーションの正常性の監視

**Developer** パースペクティブを使用して、3 種類のヘルスプローブをコンテナーに追加し、アプリケーションが正常であることを確認することができます。

- Readiness プローブを使用して、コンテナーが要求を処理する準備ができているかどうかを確認します。

- Liveness プローブを使用して、コンテナーが実行中であることを確認します。
- Startup プローブを使用して、コンテナー内のアプリケーションが起動しているかどうかを確認します。

アプリケーションの作成およびデプロイ中、またはアプリケーションをデプロイした後にヘルスチェックを追加できます。

## 12.4. DEVELOPER パースペクティブを使用したヘルスチェックの編集

**Topology** ビューを使用して、アプリケーションに追加されたヘルスチェックを編集したり、アプリケーションを変更したり、ヘルスチェックを追加したりすることができます。

**前提条件**

- Web コンソールで **Developer** パースペクティブに切り替えていること。
- **Developer** パースペクティブを使用して OpenShift Container Platform でアプリケーションを作成し、デプロイしていること。
- アプリケーションにヘルスチェックを追加していること。

**手順**

1. **Topology** ビューでアプリケーションを右クリックし、**Edit Health Checks** を選択します。または、サイドパネルで **Actions** ドロップダウンリストをクリックし、**Edit Health Checks** を選択します。
2. **Edit Health Checks** ページで以下を行います。
   - 以前に追加したヘルスプローブを削除するには、それに隣接するマイナス記号をクリックします。
   - 既存のプローブのパラメーターを編集するには、以下を実行します。
     a. 以前に追加したプローブの横にある **Edit Probe** リンクをクリックし、プローブのパラメーターを表示します。
     b. 必要に応じてパラメーターを変更し、チェックマークをクリックして変更を保存します。
   - 既存のヘルスチェックに加え、新規のヘルスプローブを追加するには、add probe リンクをクリックします。たとえば、コンテナーが実行中かどうかを確認する Liveness プローブを追加するには、以下を実行します。
     a. **Add Liveness Probe** をクリックし、プローブのパラメーターが含まれているフォームを表示します。
     b. 必要に応じてプローブのパラメーターを編集します。

**注記**

**Timeout** の値は **Period** の値よりも小さくなければなりません。**Timeout** のデフォルト値は **1** です。**Period** のデフォルト値は **10** です。

c. フォームの下部にあるチェックマークをクリックします。**Liveness Probe Added** というメッセージが表示されます。

3. **Save** をクリックして変更を保存し、追加のプローブをコンテナーに追加します。**Topology** ビューにリダイレクトされます。

4. サイドパネルで、**Pods** セクションの下にあるデプロイされた Pod をクリックして、プローブが追加されたことを確認します。

5. **Pod Details** ページで、 **Containers** セクションにリスト表示されているコンテナーをクリックします。

6. **Container Details** ページで、以前の既存プローブに加えて Liveness probe - **HTTP Get 10.129.4.65:8080/** がコンテナーに追加されていることを確認します。

## 12.5. DEVELOPER パースペクティブを使用したヘルスチェックの失敗の監視

アプリケーションのヘルスチェックに失敗した場合、**Topology** ビューを使用してこれらのヘルスチェックの違反を監視できます。

**前提条件**

- Web コンソールで **Developer** パースペクティブに切り替えていること。
- **Developer** パースペクティブを使用して OpenShift Container Platform でアプリケーションを作成し、デプロイしていること。
- アプリケーションにヘルスチェックを追加していること。

**手順**

1. **Topology** ビューで、アプリケーションノードをクリックし、サイドパネルを表示します。
2. **Observe** タブをクリックして、**Events(Warning)** セクションにヘルスチェックの失敗を確認します。
3. **Events (Warning)** に隣接する下矢印をクリックし、ヘルスチェックの失敗の詳細を確認します。

**関連情報**

- Web コンソールで **Developer** パースペクティブへの切り替え方法について、詳細は **Developer** パースペクティブの概要 を参照してください。
- アプリケーションの作成およびデプロイ時にヘルスチェックを追加する方法についての詳細は、Developer パースペクティブを使用したアプリケーションの作成セクションの**高度なオプション**を参照してください。

# 第13章 アプリケーションの編集

Topology ビューを使用して、作成するアプリケーションの設定およびソースコードを編集できます。

## 13.1. 前提条件

- 適切なプロジェクト内で、OpenShift Container Platform でアプリケーションを作成および変更するために必要な ロールおよび権限 を持っている。
- Developer パースペクティブを使用して OpenShift Container Platform でアプリケーションを作成し、デプロイしている。
- Web コンソールにログイン し、Developer パースペクティブ に切り替えている。

## 13.2. DEVELOPER パースペクティブを使用したアプリケーションのソースコードの編集

Developer パースペクティブの Topology ビューを使用して、アプリケーションのソースコードを編集できます。

**手順**

- Topology ビューで、デプロイされたアプリケーションの右下に表示される Edit Source code アイコンをクリックして、ソースコードにアクセスし、これを変更します。

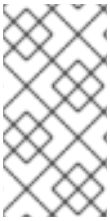
**注記**

この機能は、From Git、From Catalog、および From Dockerfile オプションを使用してアプリケーションを作成する場合にのみ利用できます。

Eclipse Che Operator がクラスターにインストールされている場合、Che ワークスペース (  ) が作成され、ソースコードを編集するためにワークスペースが表示されます。インストールされていない場合は、ソースコードがホストされている Git リポジトリー (  ) が表示されます。

## 13.3. DEVELOPER パースペクティブを使用したアプリケーション設定の編集

Developer パースペクティブの Topology ビューを使用して、アプリケーションの設定を編集できます。



**注記**

現在、Developer パースペクティブの Add ワークフローにある From Git、Container Image、From Catalog、または From Dockerfile オプションを使用して作成されるアプリケーションの設定のみを編集できます。CLI または Add ワークフローからの YAML オプションを使用して作成したアプリケーションの設定は編集できません。

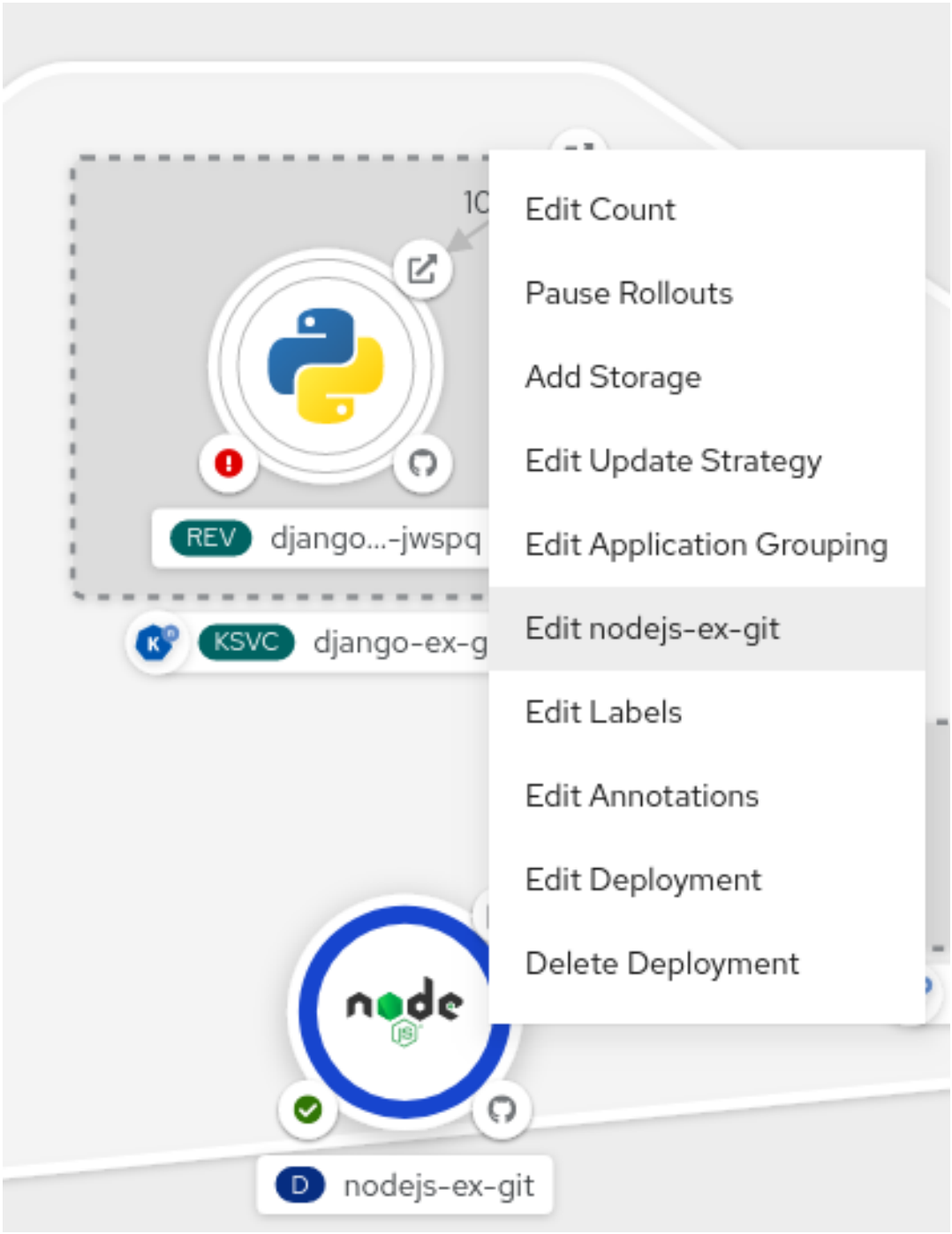
**前提条件**

Add ワークフローの From Git、Container Image、From Catalog、または From Dockerfile オプションを使用してアプリケーションを作成している。

**手順**

1. アプリケーションを作成し、アプリケーションが Topology ビューに表示された後に、アプリケーションを右クリックして選択可能な編集オプションを確認します。

**図13.1 アプリケーションの編集**




2. Edit application-name をクリックし、アプリケーションの作成に使用した Add ワークフローを表示します。このフォームには、アプリケーションの作成時に追加した値が事前に設定されています。

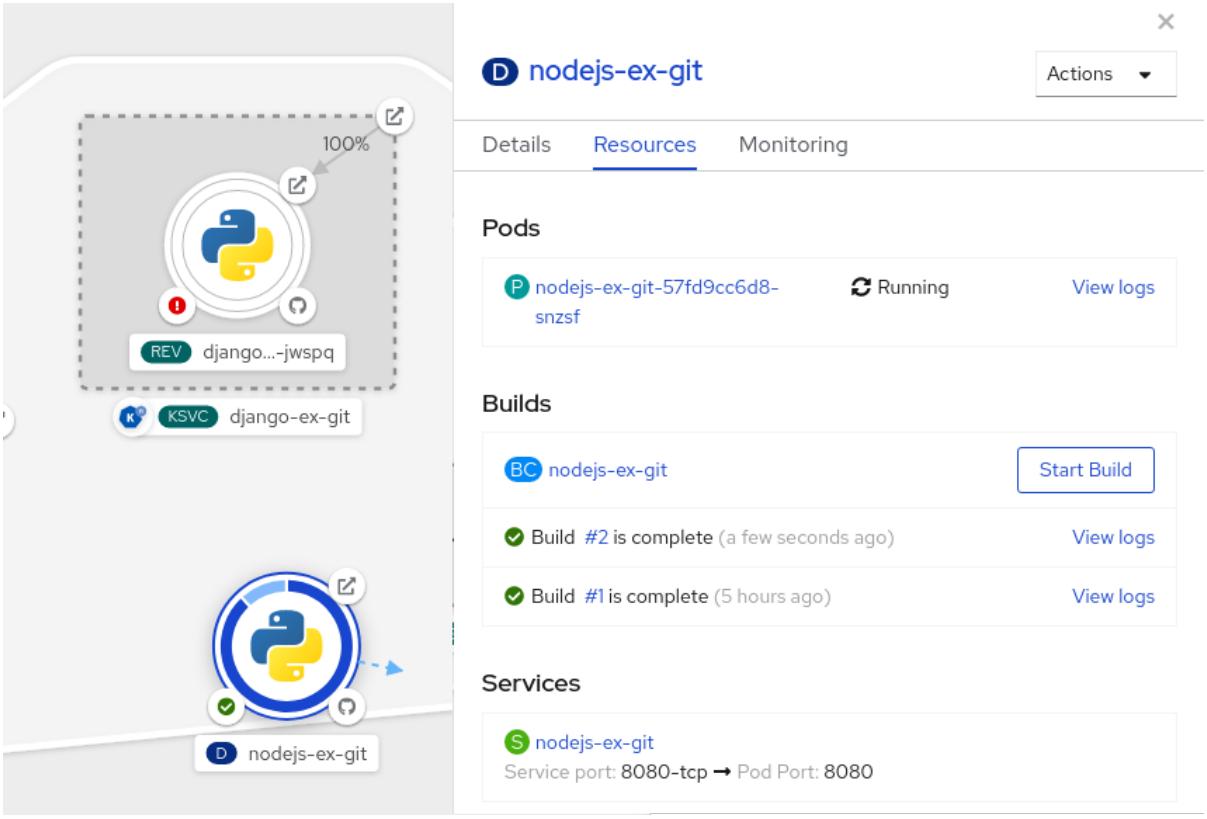
3. アプリケーションに必要な値を編集します。

**注記**

General セクションの Name フィールド、CI/CD パイプライン、または Advanced Options セクションの Create a route to the applicationフィールドを編集することはできません。

4. Save をクリックしてビルドを再起動し、新規イメージをデプロイします。

**図13.2 アプリケーションの編集および再デプロイ**



# 第14章 リソースを回収するためのオブジェクトのプルーニング

時間の経過と共に、OpenShift Container Platform で作成される API オブジェクトは、アプリケーションのビルドおよびデプロイなどの通常のユーザーの操作によってクラスターの etcd データストアに蓄積されます。

クラスター管理者は、不要になった古いバージョンのオブジェクトをクラスターから定期的にプルーニングできま。たとえば、イメージのプルーニングにより、使用されなくなったものの、ディスク領域を使用している古いイメージや層を削除できます。

## 14.1. プルーニングの基本操作

CLI は、共通の親コマンドでプルーニング操作を分類します。

```
$ oc adm prune <object_type> <options>
```

これにより、以下が指定されます。

- **groups**、**builds**、**deployments**、または **images** などのアクションを実行するための **<object_type>**。
- オブジェクトタイプのプルーニングの実行においてサポートされる **<options>**。

## 14.2. グループのプルーニング

グループのレコードを外部プロバイダーからプルーニングするために、管理者は以下のコマンドを実行できます。

```
$ oc adm prune groups \
    --sync-config=path/to/sync/config [<options>]
```

**表14.1oc adm prune groups フラグ**

| オプション | 説明 |
|---|---|
| **--confirm** | ドライランを実行する代わりにプルーニングが実行されることを示します。 |
| **--blacklist** | グループブラックリストファイルへのパス。 |
| **--whitelist** | グループホワイトリストファイルへのパス 。 |
| **--sync-config** | 同期設定ファイルへのパスです。 |

**手順**

1. prune コマンドが削除するグループを表示するには、以下のコマンドを実行します。

   ```
   $ oc adm prune groups --sync-config=ldap-sync-config.yaml
   ```

2. prune 操作を実行するには、**--confirm** フラグを追加します。

```
$ oc adm prune groups --sync-config=ldap-sync-config.yaml --confirm
```

## 14.3. デプロイメントリソースのプルーニング

使用年数やステータスによりシステムで不要となったデプロイメントに関連付けられたリソースをプルーニングできます。

以下のコマンドは、**DeploymentConfig** オブジェクトに関連付けられたレプリケーションコントローラーをプルーニングします。

```
$ oc adm prune deployments [<options>]
```

**注記**

**Deployment** オブジェクトに関連付けられたレプリカセットもプルーニングするには、**--replica-sets** フラグを使用します。このフラグは、現在テクノロジープレビュー機能です。

**表14.2 oc adm prune deployments フラグ**

| オプション | 説明 |
|---|---|
| **--confirm** | ドライランを実行する代わりにプルーニングが実行されることを示します。 |
| **--keep-complete=<N>** | **DeploymentConfig** オブジェクトに基づいて、ステータスが **Complete** でレプリカ数がゼロの最後の **N** レプリケーションコントローラーを維持します。デフォルトは **5** です。 |
| **--keep-failed=<N>** | **DeploymentConfig** オブジェクトに基づいて、ステータスが **Failed** でレプリカ数がゼロの最後の **N** レプリケーションコントローラーを維持します。デフォルトは **1** です。 |
| **--keep-younger-than=<duration>** | 現在の時間との対比で **<duration>** 未満の新しいレプリケーションコントローラーはプルーニングしません。有効な測定単位には、ナノ秒 (**ns**)、マイクロ秒 (**us**)、ミリ秒 (**ms**)、秒 (**s**)、分 (**m**)、および時間 (**h**) が含まれます。デフォルトは **60m** です。 |
| **--orphans** | **DeploymentConfig** オブジェクトを持たない、ステータスが **Complete** または **Failed** で、レプリカ数がゼロのすべてのレプリケーションコントローラーをプルーニングします。 |
| **--replica-sets=true\|false** | **true** の場合、レプリカセットはプルーニングプロセスに含まれます。デフォルトは **false** です。<br>  <br>**重要**<br>このフラグはテクノロジープレビュー機能です。 |

**手順**

**手順**

1. プルーニング操作によって削除されるものを確認するには、以下のコマンドを実行します。

```
$ oc adm prune deployments --orphans --keep-complete=5 --keep-failed=1 \
    --keep-younger-than=60m
```

2. 実際に prune 操作を実行するには、**--confirm** フラグを追加します。

```
$ oc adm prune deployments --orphans --keep-complete=5 --keep-failed=1 \
    --keep-younger-than=60m --confirm
```

## 14.4. ビルドのプルーニング

使用年数やステータスによりシステムで不要となったビルドをプルーニングするために、管理者は以下のコマンドを実行できます。

```
$ oc adm prune builds [<options>]
```

**表14.3 oc adm prune builds フラグ**

| オプション | 説明 |
|---|---|
| **--confirm** | ドライランを実行する代わりにプルーニングが実行されることを示します。 |
| **--orphans** | ビルド設定が存在せず、ステータスが complete (完了)、failed (失敗)、error (エラー)、または canceled (中止) のすべてのビルドをプルーニングします。 |
| **--keep-complete=<N>** | ビルド設定基づいて、ステータスが complete (完了) の最後の **N** ビルドを保持します。デフォルトは **5** です。 |
| **--keep-failed=<N>** | ビルド設定に基づいて、ステータスが failed (失敗)、error (エラー)、または canceled (中止) の最後の **N** ビルドを保持します。デフォルトは **1** です。 |
| **--keep-younger-than=<duration>** | 現在の時間との対比で **<duration>** 未満の新しいオブジェクトはプルーニングしません。デフォルトは **60m** です。 |

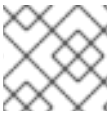
**手順**

1. プルーニング操作によって削除されるものを確認するには、以下のコマンドを実行します。

```
$ oc adm prune builds --orphans --keep-complete=5 --keep-failed=1 \
    --keep-younger-than=60m
```

2. 実際に prune 操作を実行するには、**--confirm** フラグを追加します。

```
$ oc adm prune builds --orphans --keep-complete=5 --keep-failed=1 \
    --keep-younger-than=60m --confirm
```



**注記**

開発者は、ビルドの設定を変更して自動ビルドプルーニングを有効にできます。

**関連情報**

- Performing advanced builds → Pruning builds

## 14.5. イメージの自動プルーニング

経過時間、ステータス、または制限の超過によりシステムで不要になった OpenShift イメージレジストリーのイメージは、自動的にプルーニングされます。クラスター管理者は、Pruning Custom Resource を設定したり、これを保留にしたりすることができます。

**前提条件**

- クラスター管理者のパーミッション。
- **oc** CLI がインストールされている。

**手順**

- **imagepruners.imageregistry.operator.openshift.io/cluster** という名前のオブジェクトに以下の **spec** および **status** フィールドが含まれることを確認します。

```
spec:
  schedule: 0 0 * * * (1)
  suspend: false (2)
  keepTagRevisions: 3 (3)
  keepYoungerThanDuration: 60m (4)
  keepYoungerThan: 3600000000000 (5)
  resources: {} (6)
  affinity: {} (7)
  nodeSelector: {} (8)
  tolerations: [] (9)
  successfulJobsHistoryLimit: 3 (10)
  failedJobsHistoryLimit: 3 (11)
status:
  observedGeneration: 2 (12)
  conditions: (13)
  - type: Available
    status: "True"
    lastTransitionTime: 2019-10-09T03:13:45
    reason: Ready
    message: "Periodic image pruner has been created."
  - type: Scheduled
    status: "True"
    lastTransitionTime: 2019-10-09T03:13:45
    reason: Scheduled
    message: "Image pruner job has been scheduled."
  - type: Failed
    staus: "False"
```

```
    lastTransitionTime: 2019-10-09T03:13:45
    reason: Succeeded
    message: "Most recent image pruning job succeeded."
```

1. **schedule**: **CronJob** 形式のスケジュールこれはオプションのフィールドで、デフォルトは dayly で午前 0 時でに設定されます。
2. **suspend**: **true** に設定されている場合、プルーニングを実行している **CronJob** は中断されます。これはオプションのフィールドで、デフォルトは **false** です。新規クラスターの初期値は **false** です。
3. **keepTagRevisions**: 保持するタグ別のリビジョン数です。これはオプションのフィールドで、デフォルトは **3** です。初期値は **3** です。
4. **keepYoungerThanDuration**: 指定の期間よりも後に作成されたイメージを保持します。これはオプションのフィールドです。値の指定がない場合は、**keepYoungerThan** またはデフォルト値 **60m** (60 分) のいずれかが使用されます。
5. **keepYoungerThan**: 非推奨。**keepYoungerThanDuration** と同じですが、期間は整数 (ナノ秒単位) で指定されます。これはオプションのフィールドです。**keepYoungerThanDuration** を設定すると、このフィールドは無視されます。
6. **resources**: 標準の Pod リソースの要求および制限です。これはオプションのフィールドです。
7. **affinity**: 標準の Pod のアフィニティーです。これはオプションのフィールドです。
8. **nodeSelector**: 標準の Pod ノードセレクターです。これはオプションのフィールドです。
9. **tolerations**: 標準の Pod の容認です。これはオプションのフィールドです。
10. **successfulJobsHistoryLimit**: 保持する成功したジョブの最大数です。メトリックがレポートされるようにするには **>= 1** にする必要があります。これはオプションのフィールドで、デフォルトは **3** です。初期値は **3** です。
11. **failedJobsHistoryLimit**: 保持する失敗したジョブの最大数です。メトリックがレポートされるようにするには **>= 1** にする必要があります。これはオプションのフィールドで、デフォルトは **3** です。初期値は **3** です。
12. **observedGeneration**: Operator によって観察される生成です。
13. **conditions**: 以下のタイプの標準条件オブジェクトです。
    - **Available**: プルーニングジョブが作成されているかどうかを示します。理由には Ready または Error のいずれかを使用できます。
    - **Scheduled**: 次のプルーニングジョブがスケジュールされているかどうかを示します。理由には、Scheduled、Suspended、または Error を使用できます。
    - **Failed**: 最新のプルーニングジョブが失敗したかどうかを示します。

**重要**

プルーナーを管理するためのイメージレジストリー Operator の動作は、イメージレジストリー Operator の **ClusterOperator** オブジェクトで指定される **managementState** とは独立しています。イメージレジストリー Operator が **Managed** 状態ではない場合、イメージプルーナーは Pruning Custom Resource によって設定され、管理できます。

ただし、イメージレジストリー Operator の **managementState** は、デプロイされたイメージプルーナージョブの動作を変更します。

- **Managed**: イメージプルーナーの **--prune-registry** フラグは **true** に設定されます。
- **Removed**: イメージプルーナーの **--prune-registry** フラグは **false** に設定されます。つまり、これは etcd のイメージメタデータのみのプルーニングを実行します。

## 14.6. イメージの手動プルーニング

プルーニングカスタムリソースは、OpenShift イメージレジストリーからのイメージの自動イメージプルーニングを有効にします。ただし、管理者は、使用年数やステータスまたは制限の超過によりシステムで不要となったイメージを手動でプルーニングすることができます。イメージを手動でプルーニングする方法は 2 つあります。

- イメージのプルーニングをクラスター上で **Job** または **CronJob** として実行する。
- **oc adm prune images** コマンドを実行する。

**前提条件**

- イメージをプルーニングするには、まずアクセストークンを使用してユーザーとして CLI にログインする必要があります。ユーザーにはクラスターロール **system:image-pruner** 以上のロールがなければなりません (例: **cluster-admin**)。
- イメージレジストリーを公開します。

**手順**

使用年数やステータスまたは制限の超過によりシステムで不要となったイメージを手動でプルーニングするには、以下の方法のいずれかを使用します。

- 以下の例のように、**pruner** サービスアカウントの YAML ファイルを作成して、イメージプルーニングをクラスター上で **Job** または **CronJob** として実行します。

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

**出力例**

```
kind: List
apiVersion: v1
items:
- apiVersion: v1
  kind: ServiceAccount
  metadata:
    name: pruner
    namespace: openshift-image-registry
```

```
- apiVersion: rbac.authorization.k8s.io/v1
  kind: ClusterRoleBinding
  metadata:
    name: openshift-image-registry-pruner
  roleRef:
    apiGroup: rbac.authorization.k8s.io
    kind: ClusterRole
    name: system:image-pruner
  subjects:
  - kind: ServiceAccount
    name: pruner
    namespace: openshift-image-registry
- apiVersion: batch/v1
  kind: CronJob
  metadata:
    name: image-pruner
    namespace: openshift-image-registry
  spec:
    schedule: "0 0 * * *"
    concurrencyPolicy: Forbid
    successfulJobsHistoryLimit: 1
    failedJobsHistoryLimit: 3
    jobTemplate:
      spec:
        template:
          spec:
            restartPolicy: OnFailure
            containers:
            - image: "quay.io/openshift/origin-cli:4.1"
              resources:
                requests:
                  cpu: 1
                  memory: 1Gi
              terminationMessagePolicy: FallbackToLogsOnError
              command:
              - oc
              args:
              - adm
              - prune
              - images
              - --certificate-authority=/var/run/secrets/kubernetes.io/serviceaccount/service-ca.crt
              - --keep-tag-revisions=5
              - --keep-younger-than=96h
              - --confirm=true
              name: image-pruner
            serviceAccountName: pruner
```

- **oc adm prune images [<options>]** コマンドを実行します。

```
$ oc adm prune images [<options>]
```

**--prune-registry=false** が使用されていない限り、イメージのプルーニングにより、統合レジストリーのデータが削除されます。

**--namespace** フラグの付いたイメージをプルーニングしてもイメージは削除されず、イメージストリームのみが削除されます。イメージは namespace を使用しないリソースです。そのため、プルーニングを特定の namespace に制限すると、現在の使用量を算出できなくなります。

デフォルトで、統合レジストリーは Blob のメタデータをキャッシュしてストレージに対する要求数を減らし、要求の処理速度を高めます。プルーニングによって統合レジストリーのキャッシュが更新されることはありません。プルーニング後の依然としてプルーニングされた層を含むイメージは破損します。キャッシュにメタデータを持つプルーニングされた層はプッシュされないためです。そのため、プルーニング後にキャッシュをクリアするためにレジストリーを再デプロイする必要があります。

```
$ oc rollout restart deployment/image-registry -n openshift-image-registry
```

統合レジストリーが Redis キャッシュを使用する場合、データベースを手動でクリーンアップする必要があります。

プルーニング後にレジストリーを再デプロイすることがオプションでない場合は、キャッシュを永続的に無効にする必要があります。

**oc adm prune images** 操作ではレジストリーのルートが必要です。レジストリーのルートはデフォルトでは作成されません。

Prune images CLI configuration options の表では、**oc adm prune images <options>** コマンドで使用できるオプションについて説明しています。

**表14.4 イメージのプルーニング用の CLI の設定オプション**

| オプション | 説明 |
|---|---|
| **--all** | レジストリーにプッシュされていないものの、プルスルー (pullthrough) でミラーリングされたイメージを組み込みます。これはデフォルトでオンに設定されます。プルーニングを統合レジストリーにプッシュされたイメージに制限するには、**--all=false** を渡します。 |
| **--certificate-authority** | OpenShift Container Platform で管理されるレジストリーと通信する際に使用する認証局ファイルへのパスです。デフォルトは現行ユーザーの設定ファイルの認証局データに設定されます。これが指定されている場合、セキュアな通信が実行されます。 |
| **--confirm** | test-run を実行する代わりにプルーニングが実行されることを示します。これには、統合コンテナーイメージレジストリーへの有効なルートが必要になります。このコマンドがクラスターネットワーク外で実行される場合、ルートは **--registry-url** を使用して指定される必要があります。 |
| **--force-insecure** | このオプションは注意して使用してください。HTTP 経由でホストされるか、無効な HTTPS 証明書を持つコンテナーレジストリーへの非セキュアな接続を許可します。 |
| **--keep-tag-revisions=<N>** | それぞれのイメージストリームについては、タグごとに最大 **N** のイメージリビジョンを保持します (デフォルト: **3**)。 |

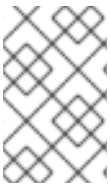
| オプション | 説明 |
| --- | --- |
| **--keep-younger-than=<duration>** | 現在の時間との対比で **<duration>** より後の新しいイメージはプルーニングしません。または、現在の時間との対比で **<duration>** より後の他のオブジェクトで参照されるイメージはプルーニングしません (デフォルト: **60m**)。 |
| **--prune-over-size-limit** | 同じプロジェクトに定義される最小の制限を超える各イメージをプルーニングします。このフラグは **--keep-tag-revisions** または **--keep-younger-than** と共に使用することはできません。 |
| **--registry-url** | レジストリーと通信する際に使用するアドレスです。このコマンドは、管理されるイメージおよびイメージストリームから判別されるクラスター内の URL の使用を試行します。これに失敗する (レジストリーを解決できないか、これにアクセスできない) 場合、このフラグを使用して他の機能するルートを指定する必要があります。レジストリーのホスト名の前には、特定の接続プロトコルを実施する **https://** または **http://** を付けることができます。 |
| **--prune-registry** | 他のオプションで規定される条件と共に、このオプションは、OpenShift Container Platform イメージ API オブジェクトに対応するレジストリーのデータがプルーニングされるかどうかを制御します。デフォルトで、イメージのプルーニングは、イメージ API オブジェクトとレジストリーの対応するデータの両方を処理します。<br><br>このオプションは、イメージオブジェクトの数を減らすなどの目的で etcd の内容のみを削除することを検討していているか (ただしレジストリーのストレージのクリーンアップは検討していない場合)、レジストリーの適切なメンテナンス期間中にレジストリーのハードプルーニングによってこれを別途実行しようとする場合に役立ちます。 |

### 14.6.1. イメージのプルーニングの各種条件

手動でプルーニングされたイメージに条件を適用できます。

- OpenShift Container Platform が管理するイメージ、またはアノテーション **openshift.io/image.managed** を持つイメージを削除するには、以下を実行します。
    - 少なくとも **--keep-younger-than** 分前に作成され、現時点ではいずれによっても参照されていません。
        - **--keep-younger-than** 分前よりも後に作成された Pod
        - **--keep-younger-than** 分前よりも後に作成されたイメージストリーム
        - 実行中の Pod
        - 保留中の Pod
        - レプリケーションコントローラー

        - デプロイメント
        - デプロイメント設定
        - レプリカセット
        - ビルド設定
        - ビルド
        - ジョブ
        - Cronjobs
        - ステートフルセット
        - **stream.status.tags[].items** の **--keep-tag-revisions** の最新のアイテム
    - これは、同じプロジェクトで定義される最小の制限を超えており、現時点ではいずれにも参照されていません。
        - 実行中の Pod
        - 保留中の Pod
        - レプリケーションコントローラー
        - デプロイメント
        - デプロイメント設定
        - レプリカセット
        - ビルド設定
        - ビルド
        - ジョブ
        - Cronjobs
        - ステートフルセット
- 外部レジストリーからのプルーニングはサポートされていません。
- イメージがプルーニングされる際、イメージのすべての参照は **status.tags** にイメージの参照を持つすべてのイメージストリームから削除されます。
- イメージによって参照されなくなったイメージ層は削除されます。



**注記**

**--prune-over-size-limit** フラグは、 **--keep-tag-revisions** フラグまたは **--keep-younger-than** フラグと共に使用することができません。これを実行すると、この操作が許可されないことを示す情報が返されます。

**--prune-registry=false** とその後にレジストリーのハードプルーニングを実行することで、OpenShift

Container Platform イメージ API オブジェクトの削除とイメージデータのレジストリーからの削除を分離することができます。これにより、タイミングウィンドウが制限され、1 つのコマンドで両方をプルーニングする場合よりも安全に実行できるようになります。ただし、タイミングウィンドウを完全に取り除くことはできません。

たとえばプルーニングの実行時にプルーニング対象のイメージを特定する場合も、そのイメージを参照する Pod を引き続き作成することができます。また、プルーニングの操作時にイメージを参照している可能性のある API オブジェクトを追跡することもできます。これにより、削除されたコンテンツの参照に関連して発生する可能性のある問題を軽減できる可能性があります。

**--prune-registry** オプションを指定しないか、**--prune-registry=true** を指定してプルーニングを再実行しても、**--prune-registry=false** を指定して以前にプルーニングされたイメージの、イメージレジストリー内で関連付けられたストレージがプルーニングされる訳ではありません。**--prune-registry=false** を指定してプルーニングされたすべてのイメージは、レジストリーのハードプルーニングによってのみ削除できます。

### 14.6.2. イメージのプルーニング操作の実行

**手順**

1. プルーニング操作によって削除されるものを確認するには、以下を実行します。

   a. 最高 3 つのタグリビジョンを保持し、60 分前よりも後に作成されたリソース (イメージ、イメージストリームおよび Pod) を保持します。

      ```
      $ oc adm prune images --keep-tag-revisions=3 --keep-younger-than=60m
      ```

   b. 定義された制限を超えるすべてのイメージをプルーニングします。

      ```
      $ oc adm prune images --prune-over-size-limit
      ```

2. 前述のステップからオプションを指定してプルーニングの操作を実行するには、以下を実行します。

   ```
   $ oc adm prune images --keep-tag-revisions=3 --keep-younger-than=60m --confirm
   ```

   ```
   $ oc adm prune images --prune-over-size-limit --confirm
   ```

### 14.6.3. セキュアまたは非セキュアな接続の使用

セキュアな通信の使用は優先され、推奨される方法です。これは、必須の証明書検証と共に HTTPS 経由で実行されます。**prune** コマンドは、可能な場合は常にセキュアな通信の使用を試行します。これを使用できない場合には、非セキュアな通信にフォールバックすることがあり、これには危険が伴います。この場合、証明書検証は省略されるか、単純な HTTP プロトコルが使用されます。

非セキュアな通信へのフォールバックは、**--certificate-authority** が指定されていない場合、以下のケースで可能になります。

1. **prune** コマンドが **--force-insecure** オプションと共に実行される。
2. 指定される **registry-url** の前に **http://** スキームが付けられる。
3. 指定される **registry-url** はローカルリンクアドレスまたは **localhost** である。

4. 現行ユーザーの設定が非セキュアな接続を許可する。これは、ユーザーが **--insecure-skip-tls-verify** を使用してログインするか、プロンプトが出される際に非セキュアな接続を選択することによって生じる可能性があります。

**重要**

レジストリーのセキュリティーが、OpenShift Container Platform で使用されるものとは異なる認証局で保護される場合、これを **--certificate-authority** フラグを使用して指定する必要があります。そうしない場合、**prune** コマンドがエラーを出して失敗します。

### 14.6.4. イメージのプルーニングに関する問題

**イメージがプルーニングされない**
イメージが蓄積し続け、**prune** コマンドが予想よりも小規模な削除を実行する場合、プルーニング候補のイメージについて満たすべきイメージプルーティングの条件があることを確認します。

とくに削除する必要のあるイメージが、それぞれのタグ履歴において選択したタグリビジョンのしきい値よりも高い位置にあることを確認します。たとえば、**sha256:abz** という名前の古く陳腐化したイメージがあるとします。イメージがタグ付けされている namespace で以下のコマンドを実行すると、イメージが **myapp** という単一イメージストリームで 3 回タグ付けされていることに気づかれるでしょう。

```
$ oc get is -n <namespace> -o go-template='{{range $isi, $is := .items}}{{range $ti, $tag :=
$is.status.tags}}'\
'{{range $ii, $item := $tag.items}}{{if eq $item.image "sha256:<hash>"}}{{$is.metadata.name}}:
{{$tag.tag}} at position {{$ii}} out of {{len $tag.items}}\n'\
'{{end}}{{end}}{{end}}{{end}}'
```

**出力例**

```
myapp:v2 at position 4 out of 5
myapp:v2.1 at position 2 out of 2
myapp:v2.1-may-2016 at position 0 out of 1
```

デフォルトオプションが使用される場合、イメージは **myapp:v2.1-may-2016** タグの履歴の **0** の位置にあるためプルーニングされません。イメージがプルーニングの対象とみなされるようにするには、管理者は以下を実行する必要があります。

- **oc adm prune images** コマンドで **--keep-tag-revisions=0** を指定します。

**警告**

このアクションを実行すると、イメージが指定されたしきい値よりも新しいか、これよりも新しいオブジェクトによって参照されていない限り、すべてのタグが基礎となるイメージと共にすべての namespace から削除されます。

- リビジョンのしきい値の下にあるすべての **istags**、つまり **myapp:v2.1** および **myapp:v2.1-may-2016** を削除します。

- 同じ **istag** にプッシュする新規ビルドを実行するか、他のイメージをタグ付けしてイメージを履歴内でさらに移動させます。ただし、これは古いリリースタグの場合には常に適切な操作となる訳ではありません。

特定のイメージのビルド日時が名前の一部になっているタグは、その使用を避ける必要があります (イメージが未定義の期間保持される必要がある場合を除きます)。このようなタグは履歴内で 1 つのイメージのみに関連付けられる可能性があり、その場合にこれらをプルーニングできなくなります。

**非セキュアなレジストリーに対するセキュアな接続の使用**
**oc adm prune images** コマンドの出力で以下のようなメッセージが表示される場合、レジストリーのセキュリティーは保護されておらず、**oc adm prune images** クライアントがセキュアな接続の使用を試行することを示しています。

```
error: error communicating with registry: Get https://172.30.30.30:5000/healthz: http: server gave HTTP response to HTTPS client
```

- 推奨される解決法として、レジストリーのセキュリティーを保護することができます。そうしない場合は、**--force-insecure** をコマンドに追加して、クライアントに対して非セキュアな接続の使用を強制することができますが、これは推奨される方法ではありません。

**セキュリティーが保護されたレジストリーに対する非セキュアな接続の使用**
**oc adm prune images** コマンドの出力に以下のエラーのいずれかが表示される場合、レジストリーのセキュリティー保護に使用されている認証局で署名された証明書が、接続の検証用に **oc adm prune images** クライアントで使用されるものとは異なることを意味します。

```
error: error communicating with registry: Get http://172.30.30.30:5000/healthz: malformed HTTP response "\x15\x03\x01\x00\x02\x02"
error: error communicating with registry: [Get https://172.30.30.30:5000/healthz: x509: certificate signed by unknown authority, Get http://172.30.30.30:5000/healthz: malformed HTTP response "\x15\x03\x01\x00\x02\x02"]
```

デフォルトでは、ユーザーの接続ファイルに保存されている認証局データが使用されます。 これはマスター API との通信の場合も同様です。

**--certificate-authority** オプションを使用してコンテナーイメージレジストリーサーバーに適切な認証局を指定します。

**正しくない認証局の使用**
以下のエラーは、セキュリティーが保護されたコンテナーイメージレジストリーの証明書の署名に使用される認証局がクライアントで使用される認証局とは異なることを示しています。

```
error: error communicating with registry: Get https://172.30.30.30:5000/: x509: certificate signed by unknown authority
```

フラグ **--certificate-authority** を使用して適切な認証局を指定します。

回避策として、**--force-insecure** フラグを代わりに追加することもできます。ただし、これは推奨される方法ではありません。

**関連情報**

- レジストリーへのアクセス
- レジストリーの公開

- レジストリールート作成方法の詳細は、OpenShift Container Platform のイメージレジストリー Operator を参照してください。

## 14.7. レジストリーのハードプルーニング

OpenShift Container レジストリーは、OpenShift Container Platform クラスターの etcd で参照されない Blob を蓄積します。基本的なイメージプルーニングの手順はこれらに対応しません。これらの Blob は **孤立した Blob** と呼ばれています。

孤立した Blob は以下のシナリオで発生する可能性があります。

- **oc delete image <sha256:image-id>** コマンドを使用してイメージを手動で削除すると、etcd のイメージのみが削除され、レジストリーのストレージからは削除されません。
- デーモンの障害によって生じるレジストリーへのプッシュにより、一部の Blob はアップロードされるものの、(最後のコンポーネントとしてアップロードされる) イメージマニフェスト はアップロードされません。固有のイメージ Blob すべてが孤立します。
- OpenShift Container Platform がクォータの制限によりイメージを拒否します。
- 標準のイメージプルーナーがイメージマニフェストを削除するが、関連する Blob を削除する前に中断されます。
- 対象の Blob を削除できないというレジストリープルーナーのバグにより、それらを参照するイメージオブジェクトは削除され、Blob は孤立します。

基本的なイメージプルーニングとは異なるレジストリーの **ハードプルーニング** により、クラスター管理者は孤立した Blob を削除することができます。OpenShift Container レジストリーのストレージ領域が不足している場合や、孤立した Blob があると思われる場合にはハードプルーニングを実行する必要があります。

これは何度も行う操作ではなく、多数の孤立した Blob が新たに作成されているという証拠がある場合にのみ実行する必要があります。または、(作成されるイメージの数によって異なりますが) 1 日 1 回などの定期的な間隔で標準のイメージプルーニングを実行することもできます。

### 手順

孤立した Blob をレジストリーからハードプルーニングするには、以下を実行します。

1. **ログイン**
   CLI で **kubeadmin** として、または **openshift-image-registry** namespace へのアクセスのある別の特権ユーザーとしてクラスターにログインします。
2. **基本的なイメージプルーニングの実行**
   基本的なイメージプルーニングにより、不要になった追加のイメージが削除されます。ハードプルーニングによってイメージが削除される訳ではありません。レジストリーストレージに保存された Blob のみが削除されます。したがって、ハードプルーニングの実行前にこれを実行する必要があります。
3. **レジストリーの読み取り専用モードへの切り替え**
   レジストリーが読み取り専用モードで実行されていない場合、プルーニングと同時に実行されているプッシュの結果は以下のいずれかになります。
   - 失敗する。孤立した Blob が新たに発生します。
   - 成功する。ただし、(参照される Blob の一部が削除されたため) イメージをプルできません。

プッシュは、レジストリーが読み取り書き込みモードに戻されるまで成功しません。したがって、ハードプルーニングは注意してスケジューリングする必要があります。

レジストリーを読み取り専用モードに切り換えるには、以下を実行します。

a. **configs.imageregistry.operator.openshift.io/cluster** で、 **spec.readOnly** を **true** に設定します。

```
$ oc patch configs.imageregistry.operator.openshift.io/cluster -p '{"spec":
{"readOnly":true}}' --type=merge
```

4. **system:image-pruner ロールの追加**
一部のリソースをリスト表示するには、レジストリーインスタンスの実行に使用するサービスアカウントに追加のパーミッションが必要になります。

a. サービスアカウント名を取得します。

```
$ service_account=$(oc get -n openshift-image-registry \
    -o jsonpath='{.spec.template.spec.serviceAccountName}' deploy/image-registry)
```

b. **system:image-pruner** クラスターロールをサービスアカウントに追加します。

```
$ oc adm policy add-cluster-role-to-user \
    system:image-pruner -z \
    ${service_account} -n openshift-image-registry
```

5. **オプション: プルーナーのドライランモードでの実行**
削除される Blob の数を確認するには、ドライランモードでハードプルーナーを実行します。実際の変更は加えられません。以下の例では、**image-registry-3-vhndw** というイメージレジストリー Pod を参照します。

```
$ oc -n openshift-image-registry exec pod/image-registry-3-vhndw -- /bin/sh -c
'/usr/bin/dockerregistry -prune=check'
```

または、プルーニング候補の実際のパスを取得するには、ロギングレベルを上げます。

```
$ oc -n openshift-image-registry exec pod/image-registry-3-vhndw -- /bin/sh -c
'REGISTRY_LOG_LEVEL=info /usr/bin/dockerregistry -prune=check'
```

**出力例**

```
time="2017-06-22T11:50:25.066156047Z" level=info msg="start prune (dry-run mode)"
distribution_version="v2.4.1+unknown" kubernetes_version=v1.6.1+$Format:%h$
openshift_version=unknown
time="2017-06-22T11:50:25.092257421Z" level=info msg="Would delete blob:
sha256:00043a2a5e384f6b59ab17e2c3d3a3d0a7de01b2cabeb606243e468acc663fa5"
go.version=go1.7.5 instance.id=b097121c-a864-4e0c-ad6c-cc25f8fdf5a6
time="2017-06-22T11:50:25.092395621Z" level=info msg="Would delete blob:
sha256:0022d49612807cb348cabc562c072ef34d756adfe0100a61952cbcb87ee6578a"
go.version=go1.7.5 instance.id=b097121c-a864-4e0c-ad6c-cc25f8fdf5a6
time="2017-06-22T11:50:25.092492183Z" level=info msg="Would delete blob:
sha256:0029dd4228961086707e53b881e25eba0564fa80033fbbb2e27847a28d16a37c"
go.version=go1.7.5 instance.id=b097121c-a864-4e0c-ad6c-cc25f8fdf5a6
time="2017-06-22T11:50:26.673946639Z" level=info msg="Would delete blob:
```

```
sha256:ff7664dfc213d6cc60fd5c5f5bb00a7bf4a687e18e1df12d349a1d07b2cf7663"
go.version=go1.7.5 instance.id=b097121c-a864-4e0c-ad6c-cc25f8fdf5a6
time="2017-06-22T11:50:26.674024531Z" level=info msg="Would delete blob:
sha256:ff7a933178ccd931f4b5f40f9f19a65be5eeeec207e4fad2a5bafd28afbef57e"
go.version=go1.7.5 instance.id=b097121c-a864-4e0c-ad6c-cc25f8fdf5a6
time="2017-06-22T11:50:26.674675469Z" level=info msg="Would delete blob:
sha256:ff9b8956794b426cc80bb49a604a0b24a1553aae96b930c6919a6675db3d5e06"
go.version=go1.7.5 instance.id=b097121c-a864-4e0c-ad6c-cc25f8fdf5a6
...
Would delete 13374 blobs
Would free up 2.835 GiB of disk space
Use -prune=delete to actually delete the data
```

6. **ハードプルーニングを実行**します。
   ハードプルーニングを実行するには、**image-registry** Pod の実行中のインスタンスのいずれかで以下のコマンドを実行します。以下の例では、**image-registry-3-vhndw** というイメージレジストリー Pod を参照します。

   ```
   $ oc -n openshift-image-registry exec pod/image-registry-3-vhndw -- /bin/sh -c
   '/usr/bin/dockerregistry -prune=delete'
   ```

   **出力例**

   ```
   Deleted 13374 blobs
   Freed up 2.835 GiB of disk space
   ```

7. **レジストリーを読み取り/書き込みモードに戻す**
   プルーニングの終了後は、レジストリーを読み取り/書き込みモードに戻すことができます。**configs.imageregistry.operator.openshift.io/cluster** で、 **spec.readOnly** を **false** に設定します。

   ```
   $ oc patch configs.imageregistry.operator.openshift.io/cluster -p '{"spec":{"readOnly":false}}' -
   -type=merge
   ```

## 14.8. CRON ジョブのプルーニング

cron ジョブは正常なジョブのプルーニングを実行できますが、失敗したジョブを適切に処理していない可能性があります。そのため、クラスター管理者はジョブの定期的なクリーンアップを手動で実行する必要があります。また、信頼できるユーザーの小規模なグループに cron ジョブへのアクセスを制限し、cron ジョブでジョブや Pod が作成され過ぎないように適切なクォータを設定する必要もあります。

**関連情報**

- ジョブを使用した Pod でのタスクの実行
- 複数のプロジェクト間のリソースクォータ
- RBAC の使用によるパーミッションの定義および適用

# 第15章 アプリケーションのアイドリング

クラスター管理者は、アプリケーションをアイドリング状態にしてリソース消費を減らすことができます。これは、コストがリソース消費と関連付けられるパブリッククラウドにデプロイされている場合に役立ちます。

スケーラブルなリソースが使用されていない場合、OpenShift Container Platform はリソースを検出した後にそれらを **0** レプリカに設定してアイドリングします。ネットワークトラフィックがリソースに送信される場合、レプリカをスケールアップしてアイドリング解除を実行し、通常の操作を続行します。

アプリケーションは複数のサービスやデプロイメント設定などの他のスケーラブルなリソースで設定されています。アプリケーションのアイドリングには、関連するすべてのリソースのアイドリングを実行することが関係します。

## 15.1. アプリケーションのアイドリング

アプリケーションのアイドリングには、サービスに関連付けられたスケーラブルなリソース (デプロイメント設定、レプリケーションコントローラーなど) を検索することが必要です。アプリケーションのアイドルリングには、サービスを検索してこれをアイドリング状態としてマークし、リソースを zero レプリカにスケールダウンすることが関係します。

**oc idle** コマンドを使用して単一サービスをアイドリングするか、**--resource-names-file** オプションを使用して複数のサービスをアイドリングすることができます。

### 15.1.1. 単一サービスのアイドリング

**手順**

1. 単一のサービスをアイドリングするには、以下を実行します。

```
$ oc idle <service>
```

### 15.1.2. 複数サービスのアイドリング

複数サービスのアイドリングは、アプリケーションがプロジェクト内の一連のサービスにまたがる場合や、同じプロジェクト内で複数のアプリケーションを一括してアイドリングするため、複数サービスをスクリプトを併用してアイドリングする場合に役立ちます。

**手順**

1. 複数サービスのリストを含むファイルを作成します (それぞれを各行に指定)。
2. **--resource-names-file** オプションを使用してサービスをアイドリングします。

```
$ oc idle --resource-names-file <filename>
```

**注記**

**idle** コマンドは単一プロジェクトに制限されます。クラスター全体でアプリケーションをアイドリングするには、各プロジェクトに対して **idle** コマンドを個別に実行します。

## 15.2. アプリケーションのアイドリング解除

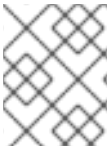
アプリケーションサービスは、ネットワークトラフィックを受信し、直前の状態に再びスケールアップすると再びアクティブになります。これには、サービスへのトラフィックとルートを通るトラフィックの両方が含まれます。

また、アプリケーションはリソースをスケールアップすることにより、手動でアイドリング解除することができます。

**手順**

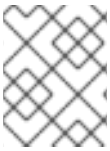
1. DeploymentConfig をスケールアップするには、以下を実行します。

```
$ oc scale --replicas=1 dc <dc_name>
```



**注記**

現時点で、ルーターによる自動アイドルリング解除はデフォルトの HAProxy ルーターのみでサポートされています。



**注記**

Kuryr-Kubernetes を SDN として設定している場合、サービスは自動アイドリング解除をサポートしません。

# 第16章 アプリケーションの削除

プロジェクトで作成されたアプリケーションを削除できます。

## 16.1. DEVELOPER パースペクティブを使用したアプリケーションの削除

**Developer** パースペクティブの **Topology** ビューを使用して、アプリケーションとその関連コンポーネントすべてを削除できます。

1. 削除するアプリケーションをクリックし、アプリケーションのリソースの詳細を含むサイドパネルを確認します。
2. パネルの右上に表示される **Actions** ドロップダウンメニューをクリックし、**Delete Application** を選択して確認ダイアログボックスを表示します。
3. アプリケーションの名前を入力して **Delete** をクリックし、これを削除します。

削除するアプリケーションを右クリックし、**Delete Application** をクリックして削除することもできます。

# 第17章 RED HAT MARKETPLACE の使用

Red Hat Marketplace は、パブリッククラウドおよびオンプレミスで実行されるコンテナーベース環境向けの認定されたソフトウェアの検出とアクセスを容易にする、オープンクラウドマーケットプレースです。

## 17.1. RED HAT MARKETPLACE 機能

クラスター管理者は Red Hat Marketplace を使用して OpenShift Container Platform でソフトウェアを管理し、開発者にアプリケーションインスタンスをデプロイするためのセルフサービスアクセスを付与し、アプリケーションの使用状況をクォータに対して関連付けることができます。

### 17.1.1. OpenShift Container Platform クラスターの Marketplace への接続

クラスター管理者は、Marketplace に接続する OpenShift Container Platform クラスターに、共通のアプリケーションセットをインストールできます。また、Marketplace を使用し、サブスクリプションまたはクォータに対してクラスターの使用状況を追跡することもできます。Marketplace を使用して追加したユーザーは、それぞれの製品のの使用状況を追跡し、組織に対して請求できます。

クラスター接続のプロセス で、イメージレジストリーシークレットを更新し、カタログを管理し、アプリケーションの使用状況を報告する Marketplace Operator がインストールされています。

### 17.1.2. アプリケーションのインストール

クラスター管理者は、OpenShift Container Platform の OperatorHub 内から、または Marketplace Web アプリケーション から Marketplace アプリケーションをインストール できます。

**Operators > Installed Operators**をクリックして、Web コンソールからインストールされたアプリケーションにアクセスできます。

### 17.1.3. 異なるパースペクティブからのアプリケーションのデプロイ

Web コンソールの Administrator および Developer パースペクティブから Marketplace アプリケーションをデプロイすることができます。

**Developer パースペクティブ**
開発者は Developer パースペクティブを使用して、新しくインストールされた機能にアクセスできます。

たとえば、データベース Operator のインストール後に、開発者はプロジェクト内のカタログからインスタンスを作成できます。データベースの使用状況は集計され、クラスター管理者に報告されます。

このパースペクティブには、Operator のインストールやアプリケーション使用状況の追跡は含まれません。

**Administrator パースペクティブ**
クラスター管理者は、Administrator パースペクティブから Operator のインストールおよびアプリケーションの使用状況の情報にアクセスできます。

また、**Installed Operators** リストでカスタムリソース定義 (CRD) を参照してアプリケーションインスタンスを起動することもできます。