Red Hat OpenShift Service on AWS 4

アプリケーションのビルド

アプリケーション向けの Red Hat OpenShift Service on AWS の設定

Last Updated: 2025-11-28

Red Hat OpenShift Service on AWS 4 アプリケーションのビルド

アプリケーション向けの Red Hat OpenShift Service on AWS の設定

Legal Notice

Abstract

このドキュメントでは、アプリケーションのデプロイメント用に Red Hat OpenShift Service on AWS (ROSA) を設定する方法を説明します。カスタムワイルドカードドメインの設定も含まれます。

Table of Contents

第1章 アプリケーションのビルドの概要

Red Hat OpenShift Service on AWS 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。Red Hat OpenShift Service on AWS 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。

Dedicated の管理者権限のあるユーザーは、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ようになっている必要があります。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。Red Hat OpenShift Service on AWS 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

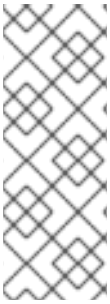
Deployment または **DeploymentConfig** オブジェクトを使用してアプリケーションをデプロイし、Web コンソールからそれらを 管理 できます。アプリケーションの変更またはアップグレード中のダウンタイムを短縮するのに役立つ デプロイメントストラテジー を作成できます。

Helm (アプリケーションやサービスの Red Hat OpenShift Service on AWS 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) も使用できます。

第2章 プロジェクト

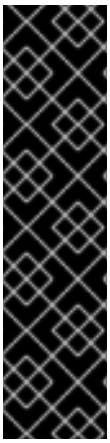
2.1. プロジェクトの使用

プロジェクト を使用すると、ユーザーコミュニティーを他のコミュニティーと切り離した状態で独自のコンテンツを整理し、管理できます。



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 として実行す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、Red Hat OpenShift Service on AWS 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。



重要

デフォルトプロジェクトでワークロードを実行したり、デフォルトプロジェクトへのアクセスを共有したりしないでください。デフォルトのプロジェクトは、コアクラスターコンポーネントを実行するために予約されています。

デフォルトプロジェクトである **default**、**kube-public**、**kube-system**、**openshift**、**openshift-infra**、**openshift-node**、および **openshift.io/run-level** ラベルが **0** または **1** に設定されているその他のシステム作成プロジェクトは、高い特権があるとみなされます。Pod セキュリティーアドミッション、Security Context Constraints、クラスターリソースクォータ、イメージ参照解決などのアドミッションプラグインに依存する機能は、高い特権を持つプロジェクトでは機能しません。

2.1.1. プロジェクトの作成

Red Hat OpenShift Service on AWS 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

Red Hat OpenShift Service on AWS 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 Red Hat OpenShift Service on AWS によって重要 (Critical) と見なされます。そのため、Red Hat OpenShift Service on AWS では、Web コンソールを使用して **openshift-** で始まる名前のプロジェクトを作成できません。

前提条件

- Red Hat OpenShift Service on AWS でプロジェクト、アプリケーション、その他のワークロードを作成するための適切なロールと権限を持っている。

手順

- Administrator パースペクティブを使用している場合は、以下を行います。

  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。

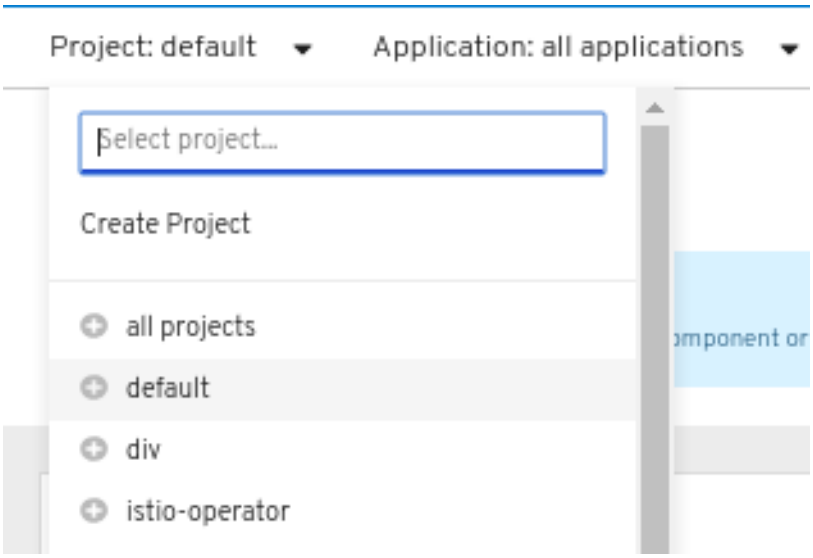
  c. オプション: **Details** タブを選択して、プロジェクトの詳細を表示します。

  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は、以下を行います。

  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。

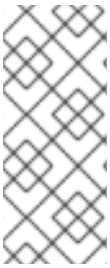
  d. オプション: プロジェクトに対する適切なパーミッションがある場合は、プロジェクトダッシュボードの **Project Access** タブを使用して、プロジェクトの admin、edit、および view 権限を付与または取り消すことができます。

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は、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 Red Hat OpenShift Service on AWS によって重要 (Critical) と見なされます。そのため、Red Hat OpenShift Service on AWS 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はできません。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。

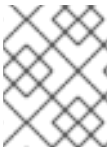
手順

- 以下を実行します。

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

以下は例になります。

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



注記

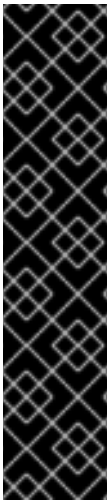
上限に達した後に作成できるプロジェクトの数。新規プロジェクトを作成できるように既存プロジェクトを削除しなければならない場合があります。

2.1.2. プロジェクトの表示

Red Hat OpenShift Service on AWS 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たプロジェクトの表示

Red Hat OpenShift Service on AWS 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。



重要

Red Hat OpenShift Service on AWS 4.19 以降、Web コンソールのパースペクティブが統合されました。**Developer** パースペクティブは、デフォルトでは有効になっていません。

どのユーザーも、Red Hat OpenShift Service on AWS Web コンソールのすべての機能を操作できます。ただし、クラスターの所有者でない場合は、特定の機能にアクセスする権限をクラスターの所有者に要求する必要がある場合があります。

引き続き **Developer** パースペクティブを有効にできます。Web コンソールの **Getting Started** ペインでは、コンソールツアーの実行、クラスター設定に関する情報の検索、**Developer** パースペクティブを有効にするためのクイックスタートの表示、リンク先を表示して新機能の確認などを行えます。

手順

- 管理者としてログインしている場合は、以下を実行します。
  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。
- 開発者としてログインしている場合は、以下を実行します。
  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。
  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 操作で、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

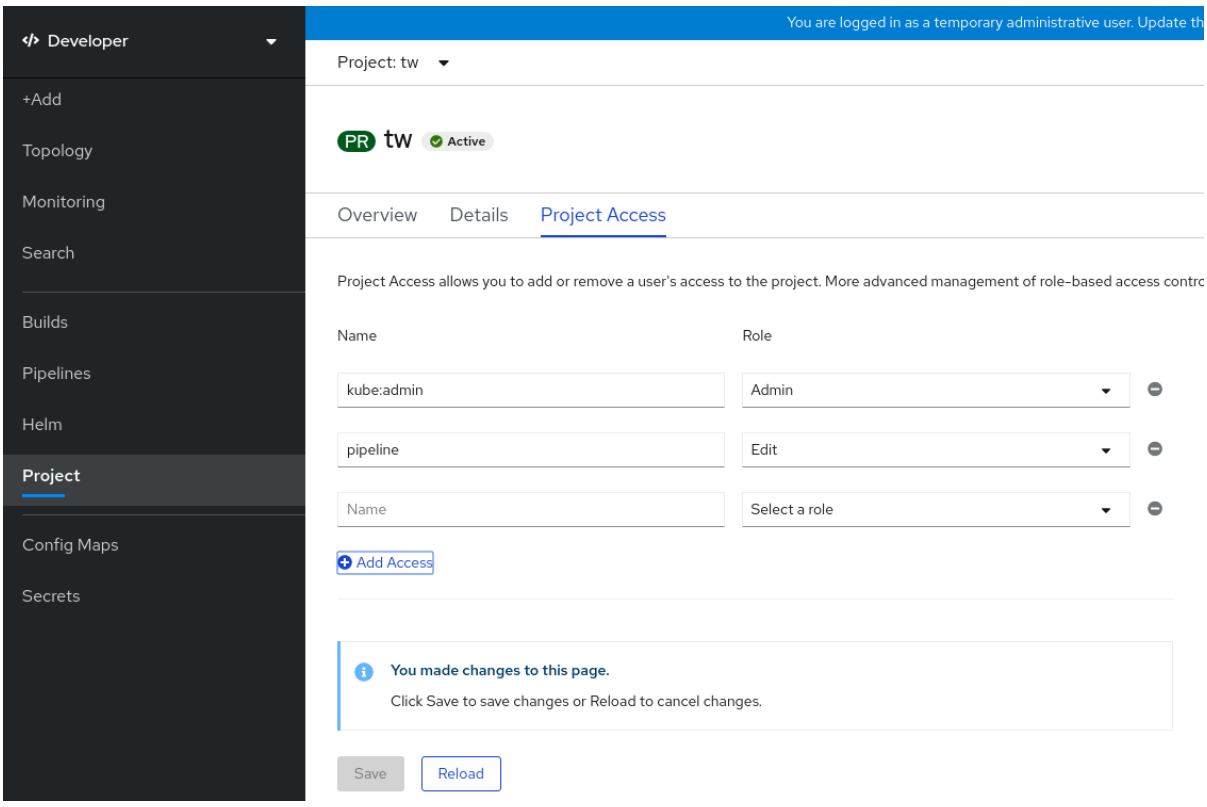
- プロジェクトを作成している。

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

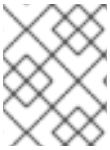


5. ユーザー名を入力し、**Select a role** ドロップダウンリストをクリックし、適切なロールを選択します。

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、**Console operator.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

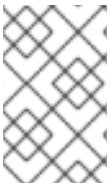
+Add ページを使用して、プロジェクトにアイテムを追加できます。

前提条件

- プロジェクトを作成している。

手順

1. **+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

Red Hat OpenShift Service on AWS 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

1. **Home → Projects** に移動します。
2. 一覧からプロジェクトを選択します。
3. **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、プロジェクトの名前に置き換えます。

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

Red Hat OpenShift Service on AWS 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。その後、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、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** の場合は、新規のコンテンツをプロジェクトに追加することができません。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は、以下を行います。

  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は、以下を行います。

  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と、Delete Project オプションを選択できません。

  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. プロジェクト作成の設定

Red Hat OpenShift Service on AWS では、**プロジェクト** を使用して、関連するオブジェクトをグループ化および分離します。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、Red Hat OpenShift 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。

2.2.1. プロジェクト作成について

Red Hat OpenShift Service on AWS 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と、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。

2.2.2. 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。

前提条件

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

手順

1. **cluster-admin** 権限を持つユーザーとしてログインします。
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 に作成する必要があります。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。

2.2.3. 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ます。
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は、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は、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に、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。

2.2.4. 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は、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ます。

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。

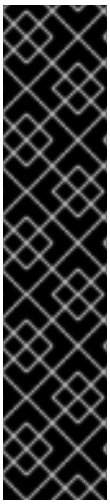
```
apiVersion: config.openshift.io/v1
kind: Project
metadata:
# ...
spec:
  projectRequestMessage: To request a project, contact your system administrator at
projectname@example.com.
# ...
```

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 Red Hat OpenShift Service on AWS 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。



重要

Red Hat OpenShift Service on AWS 4.19 以降、Web コンソールのパースペクティブが統合されました。**Developer** パースペクティブは、デフォルトでは有効になっていません。

どのユーザーも、Red Hat OpenShift Service on AWS Web コンソールのすべての機能を操作できます。ただし、クラスターの所有者でない場合は、特定の機能にアクセスする権限をクラスターの所有者に要求する必要がある場合があります。

引き続き **Developer** パースペクティブを有効にできます。Web コンソールの **Getting Started** ペインでは、コンソールツアーの実行、クラスター設定に関する情報の検索、**Developer** パースペクティブを有効にするためのクイックスタートの表示、リンク先を表示して新機能の確認などを行えます。

- **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、Red Hat OpenShift Service on AWS 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を確認してください。
  - **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。
  - **すべてのサービス**: カタログを参照して、Red Hat OpenShift Service on AWS 全体のサービスを見つけます。
  - **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、Red Hat OpenShift Service on AWS でアプリケーションをビルドしてデプロイします。
- **Container images**: 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 Red Hat OpenShift Service on AWS にデプロイします。
- **Pipelines**: Tekton パイプラインを使用して Red Hat OpenShift Service on AWS 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、Red Hat OpenShift Service on AWS 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 オプションは、OpenShift Pipelines Operator がインストールされている場合にのみ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。

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

- Red Hat OpenShift Service on AWS Web コンソールにログインしており、**Developer** パースペクティブを使用している。

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

クイックスタート ページには、Red Hat OpenShift Service on AWS でアプリケーションを作成、インポート、実行する方法が、段階的な手順とタスクとともに示されています。

前提条件

- Red Hat OpenShift Service on AWS Web コンソールにログインしており、**Developer** パースペクティブを使用している。

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 Red Hat OpenShift Service on AWS でアプリケーションを作成、ビルド、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

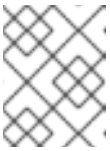
1. **+Add** ビューで、**Git Repository** タイルの **From Git** をクリックし、**Import from git** フォームを表示します。
2. **Git** セクションで、アプリケーションの作成に使用するコードベースの Git リポジトリー URL を入力します。たとえば、このサンプル nodejs アプリケーションの URL **https://github.com/sclorg/nodejs-ex** を入力します。その後、URL は検証されます。
3. オプション: **Show Advanced Git Options** をクリックし、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: **Devfile**、**Dockerfile**、**Builder Image**、または **Serverless Function** が Git リポジトリーからインポートして、デプロイをさらにカスタマイズできます。
   - Git リポジトリーに **Devfile**、**Dockerfile**、**Builder Image**、または **func.yaml** が含まれている場合は自動的に検出され、それぞれのパスフィールドに入力されます。
   - **Devfile**、**Dockerfile**、または **Builder Image** が同じリポジトリーで検出された場合は、デフォルトで **Devfile** が選択されます。
   - Git リポジトリーで **func.yaml** が検出されると、Import Strategy が **Serverless Function** に変更になります。
   - または、Git リポジトリー URL を使用して **+Add** ビューでサーバー **Create Serverless function** をクリックして、サーバーレス関数を作成することもできます。
   - 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 **Devfiles**、**Dockerfiles**、または **Builder Images** が検出された場合、特定のインスタンスをインポートするには、コンテキストディレクトリーからの相対パスをそれぞれ指定します。
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オーバーライドします。

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。
- **デプロイメント Config**: Red Hat OpenShift Service on AWS 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

アプリケーションをインポートするためのデフォルトのリソース設定を設定するには、**User Preferences → Applications → Resource type** フィールドに移動します。**Serverless Deployment** オプションは、OpenShift 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。**Resources** セクションは、サーバーレス関数の作成時には利用できません。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一覧から必要なパイプラインを選択できます。

注記

次の基準が満たされている場合、**Add pipeline** チェックボックスがオンになり、**Configure PAC** がデフォルトで選択されます。

- パイプラインオペレーターがインストールされています
- **pipelines-as-code** が有効になっています
- **.tekton** ディレクトリーが Git リポジトリーで検出される

9. Webhook をリポジトリーに追加します。**Configure PAC** がオンになっており、GitHub アプリがセットアップされている場合は、**Use GitHub App** と **Setup a webhook** オプションが表示されます。GitHub アプリケーションがセットアップされていない場合は、**Setup a webhook** オプションのみが表示されます。

   a. **Settings → Webhooks** に移動し、**Add webhook** をクリックします。

   b. **Payload URL** を Pipelines as Code コントローラーのパブリック URL に設定します。

   c. コンテンツタイプを **application/json** として選択します。

   d. Webhook シークレットを追加し、別の場所に書き留めます。**openssl** がローカルマシンにインストールされた状態で、ランダムなシークレットを生成します。

   e. **Let me select individual events** をクリックし、**Commit comments**、**Issue comments**、**Pull request**、および **Pushes** のイベントを選択します。

   f. **Add webhook** をクリックします。

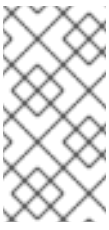
10. オプション: **Advanced Options** セクションでは、**Target port** および **Create a route to the application** がデフォルトで選択されるため、公開されている URL を使用してアプリケーションにアクセスできます。
アプリケーションがデフォルトのパブリックポート 80 でデータを公開しない場合は、チェックボックスの選択を解除し、公開する必要のあるターゲットポート番号を設定します。

11. 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関する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、オートスケーラー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。

12. **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. コンテナーイメージをデプロイしてアプリケーションを作成

外部イメージレジストリーまたは内部レジストリーのイメージストリームタグを使用して、アプリケーションをクラスターにデプロイできます。

前提条件

- Red Hat OpenShift Service on AWS Web コンソールにログインしており、**Developer** パースペクティブを使用している。

手順

1. **+Add** ビューで、**Container images** をクリックして、**Deploy Images** ページを表示します。
2. **Image** セクションで以下を行います。
   a. **Image name from external registry** を選択してパブリックレジストリーまたはプライベートレジストリーからイメージをデプロイメントするか、**Image stream tag from internal registry** を選択して内部レジストリーからイメージをデプロイメントします。
   b. **Runtime icon** タブでイメージのアイコンを選択します。
3. **General** セクションで、以下を実行します。
   a. **Application name** フィールドに、アプリケーションを分類するための一意の名前を入力します。
   b. **Name** フィールドに、このコンポーネント用に作成されたリソースを識別するための一意の名前を入力します。
4. **Resource type** セクションで、生成するリソースタイプを選択します。
   a. **Deployment** を選択して、**Pod** および **ReplicaSet** オブジェクトの宣言的更新を有効にします。

b. **DeploymentConfig** を選択して、**Pod** オブジェクトのテンプレートを定義し、新しいイメージと設定ソースのデプロイを管理します。

c. アイドル時にゼロへのスケーリングを有効にするには、**Serverless Deployment** を選択します。

5. **Create** をクリックします。**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。

3.1.6. 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。
- Red Hat OpenShift Service on AWS Web コンソールにアクセスし、**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: **Resource type** ドロップダウンリストを使用して、リソースタイプを変更します。

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。

3.1.7. 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、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。

3.1.8. 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。

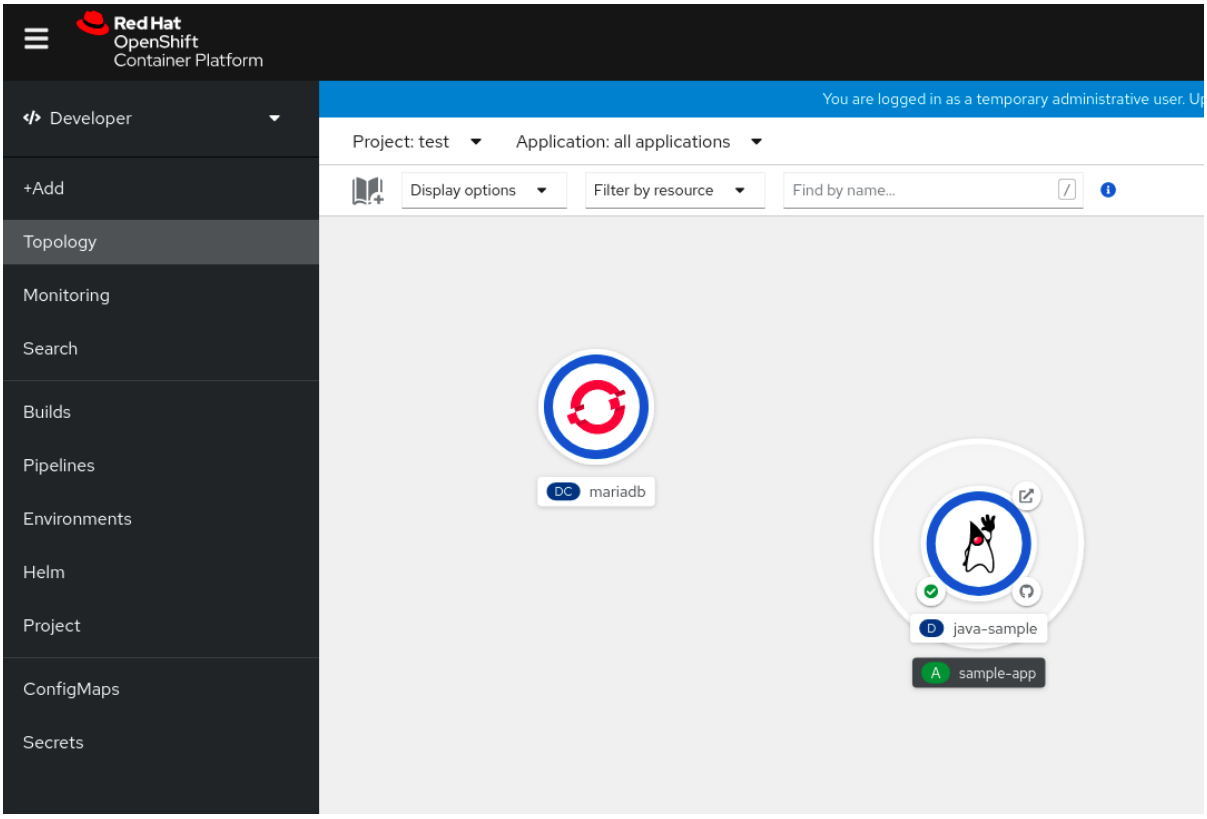
手順

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



3.1.9. 関連情報

- OpenShift Serverless の Knative ルーティング設定の詳細は、ルーティング を参照してください。
- OpenShift Serverless のドメインマッピング設定の詳細は、Knative サービスのカスタムドメインの設定 を参照してください。
- OpenShift Serverless の Knative 自動スケーリング設定の詳細は、自動スケーリング 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、デプロイ、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 Red Hat OpenShift Service on AWS で作成できます。

以下では、開発者を対象に、Red Hat OpenShift Service on AWS 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。

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 クラスターの作成を説明します。

前提条件

- Red Hat OpenShift Service on AWS クラスターへのアクセス。
- 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 Red Hat OpenShift Service on AWS 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. **Ecosystem → Installed Operators** ページに移動します。このページには、dedicated-admin によってクラスターにインストールされた使用可能な Operators が、クラスターサービスバージョン (CSV) のリストの形で表示されます。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. 次のページでは、**EtcdCluster** オブジェクト (クラスターのサイズなど) のテンプレートを起動する最小条件を変更できます。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、**Resources** タブの順にクリックし、Operator が自動的に作成および設定した多数のリソースが含まれてい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 Operators 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。最も重要なことは、適切なアクセス権を持つ dedicated-admin または開発者が、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った点です。

3.3. CLI を使用したアプリケーションの作成

Red Hat OpenShift Service on AWS 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 Red Hat OpenShift Service on AWS 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。

Red Hat OpenShift Service on AWS は、パイプラインまたはソース、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、Red Hat OpenShift Service on AWS 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。

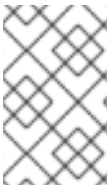
3.3.1.2. リモート

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
```

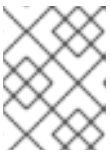
また、リモート URL を指定する場合は、以下のように URL の最後に **#<branch_name>** を追加することで、使用する Git ブランチを指定できます。

```
$ 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

Red Hat OpenShift Service on AWS は、特定のファイルを検出することで、使用するビルドストラテジーを自動的に決定します。

- 新規アプリケーションの作成時に Jenkins ファイル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、Red Hat OpenShift Service on AWS 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。

- 新しいアプリケーションの作成時に、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 Dockerfile が存在する場合、Red Hat OpenShift Service on AWS 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、Red Hat OpenShift Service on AWS 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オーバーライドします。

```
$ 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3.1new-app が検出する言語

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

言語の検出後、**new-app** は Red Hat OpenShift Service on AWS サーバーで、検出された言語と一致する **supports** アノテーションが指定されたイメージストリームタグか、または検出された言語の名前に一致するイメージストリームを検索します。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。

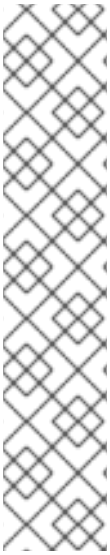
~ をセパレーターとして使用し、イメージ (イメージストリームまたはコンテナーの仕様) とリポジトリーを指定して、ビルダーが特定のソースリポジトリーを使用するようにイメージをオーバーライドすることができます。この方法を使用すると、ビルドストラテジーの検出および言語の検出は実行されない点に留意してください。

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、Red Hat OpenShift Service on AWS 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、または **-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 Red Hat OpenShift Service on AWS 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
```

Red Hat OpenShift Service on AWS 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。以下は例になります。

```
$ 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、Red Hat OpenShift Service on AWS 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。
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 Red Hat OpenShift Service on AWS 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
```

3.3.4.10. インポートモードの設定

oc new-app を使用するときにインポートモードを設定するには、**--import-mode** フラグを追加します。このフラグには **Legacy** または **PreserveOriginal** を追加できます。これにより、それぞれ単一のサブマニフェストまたはすべてのマニフェストを使用してイメージストリームを作成するオプションがユーザーに提供されます。

```
$ oc new-app --image=registry.redhat.io/ubi8/httpd-24:latest  --import-mode=Legacy --name=test
```

```
$ oc new-app --image=registry.redhat.io/ubi8/httpd-24:latest  --import-mode=PreserveOriginal --
name=test
```

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。
- **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。

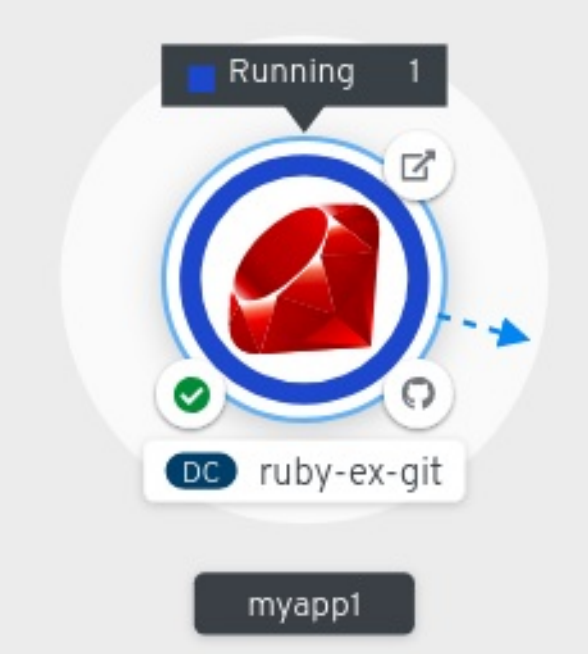
- カーソルを Pod の左下のアイコンの上に置き、最新ビルドおよびそのステータスを確認します。アプリケーションビルドのステータスは、**New** ( )、**Pending** ()、**Running** ( )、**Completed** ( )、**Failed** ( )、および **Canceled** ( ) と表示されます。
- Pod のステータスまたはフェーズは、色で区別され、ツールチップで次のように表示されます。
  - **Running** ( ): Pod はノードにバインドされ、すべてのコンテナーが作成されます。1 つ以上のコンテナーが実行中か、起動または再起動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。

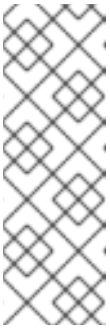
- アプリケーションを作成し、イメージがデプロイされると、ステータスは **Pending** と表示されます。アプリケーションをビルドすると、**Running** と表示されます。

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。

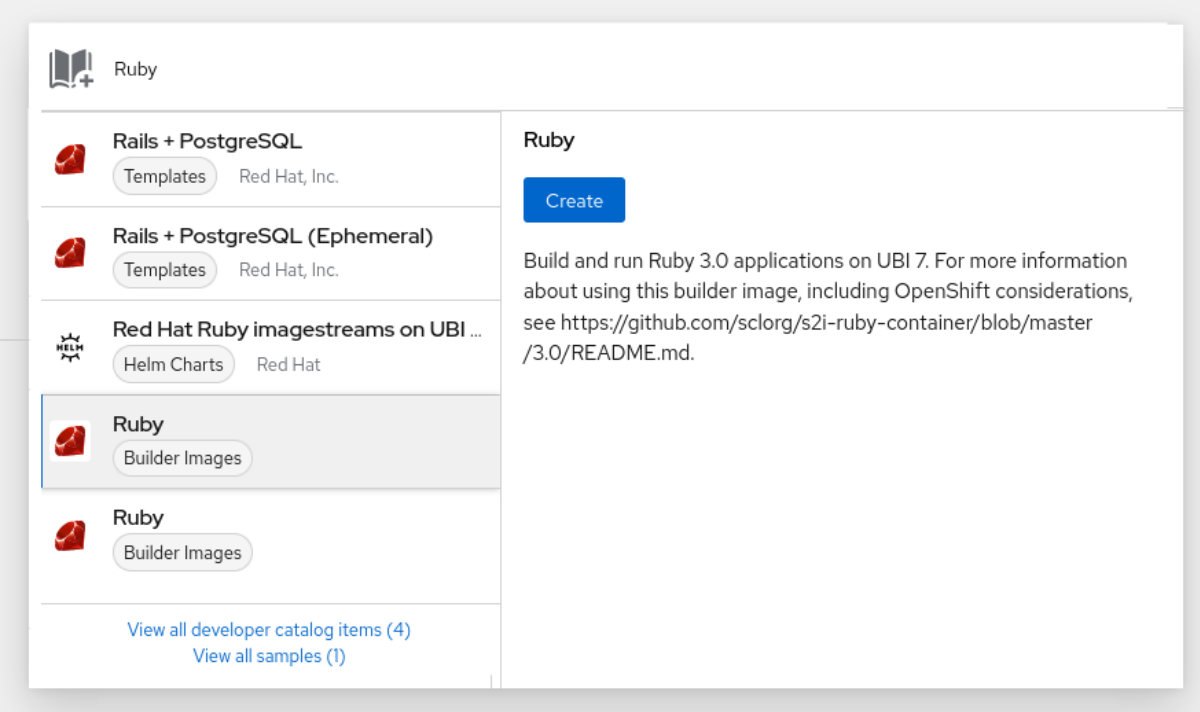
4.5. コンポーネントの既存プロジェクトへの追加

プロジェクトにコンポーネントを追加できます。

手順

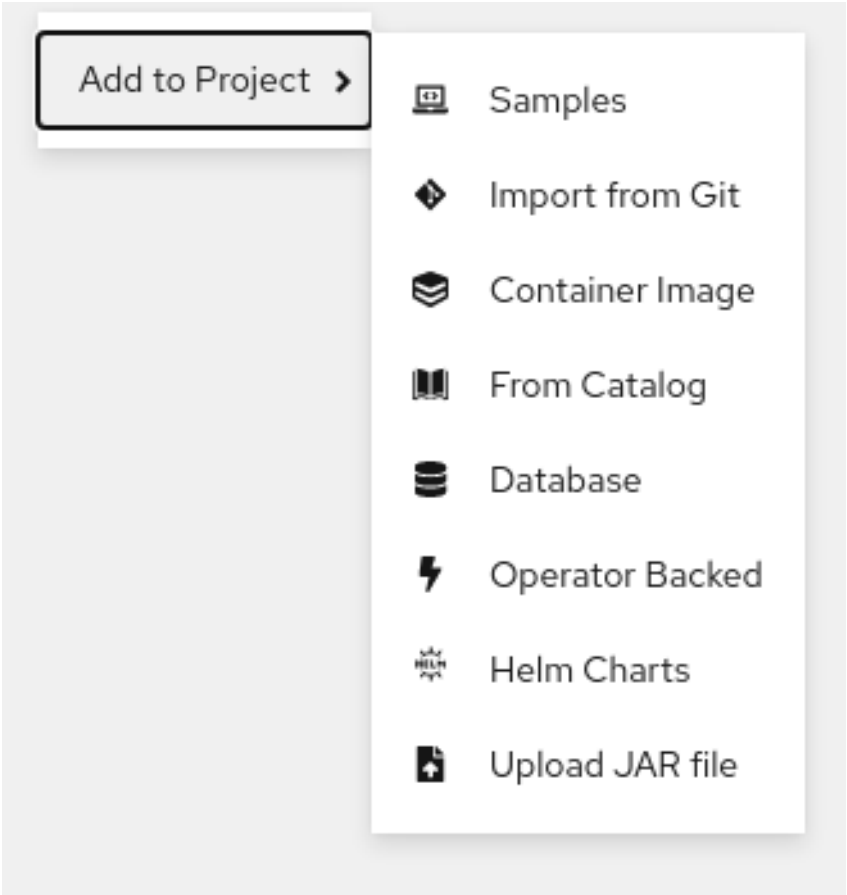
1. **+Add** ビューに移動します。
2. **Add to Project** () をクリックし、左側のナビゲーションペインまたは **Ctrl**+**Space** を押します。
3. コンポーネントを検索し、**Start**/**Create**/**Install** ボタンをクリックするか、**Enter** をクリックしてコンポーネントをプロジェクトに追加し、トポロジー **Graph view** で確認します。

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、トポロジー**グラフビュー** 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。

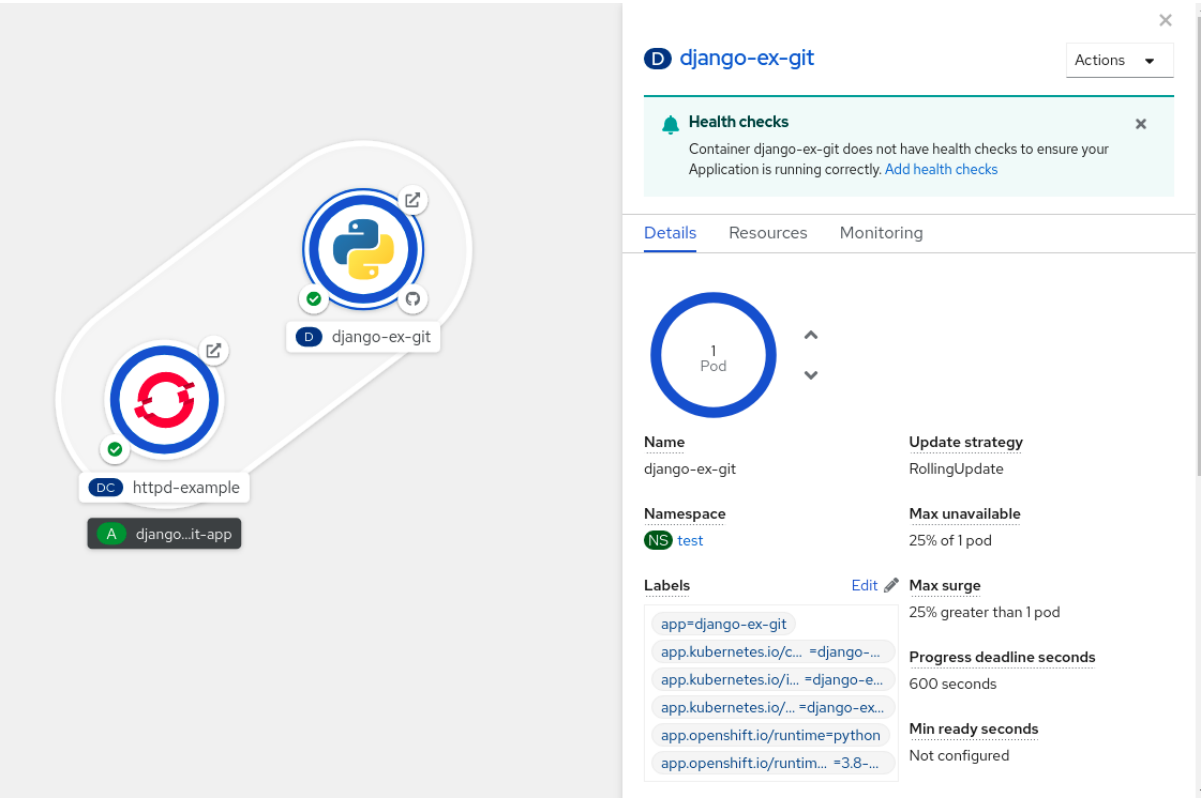
前提条件

- **Developer** パースペクティブを使用して、Red Hat OpenShift Service on AWS に少なくとも 2 つ以上のコンポーネントが作成、デプロイされている。

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。

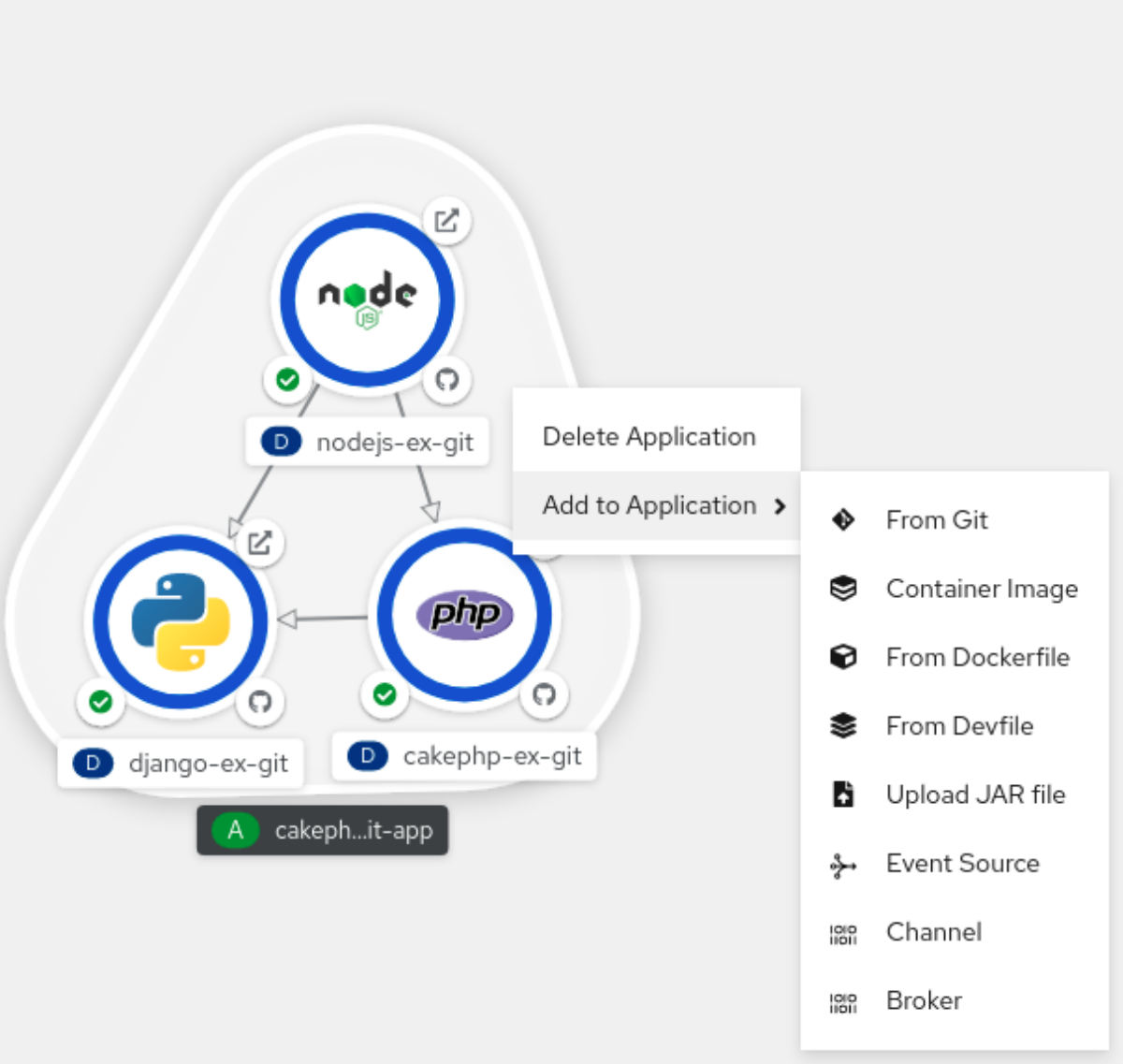
注記

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

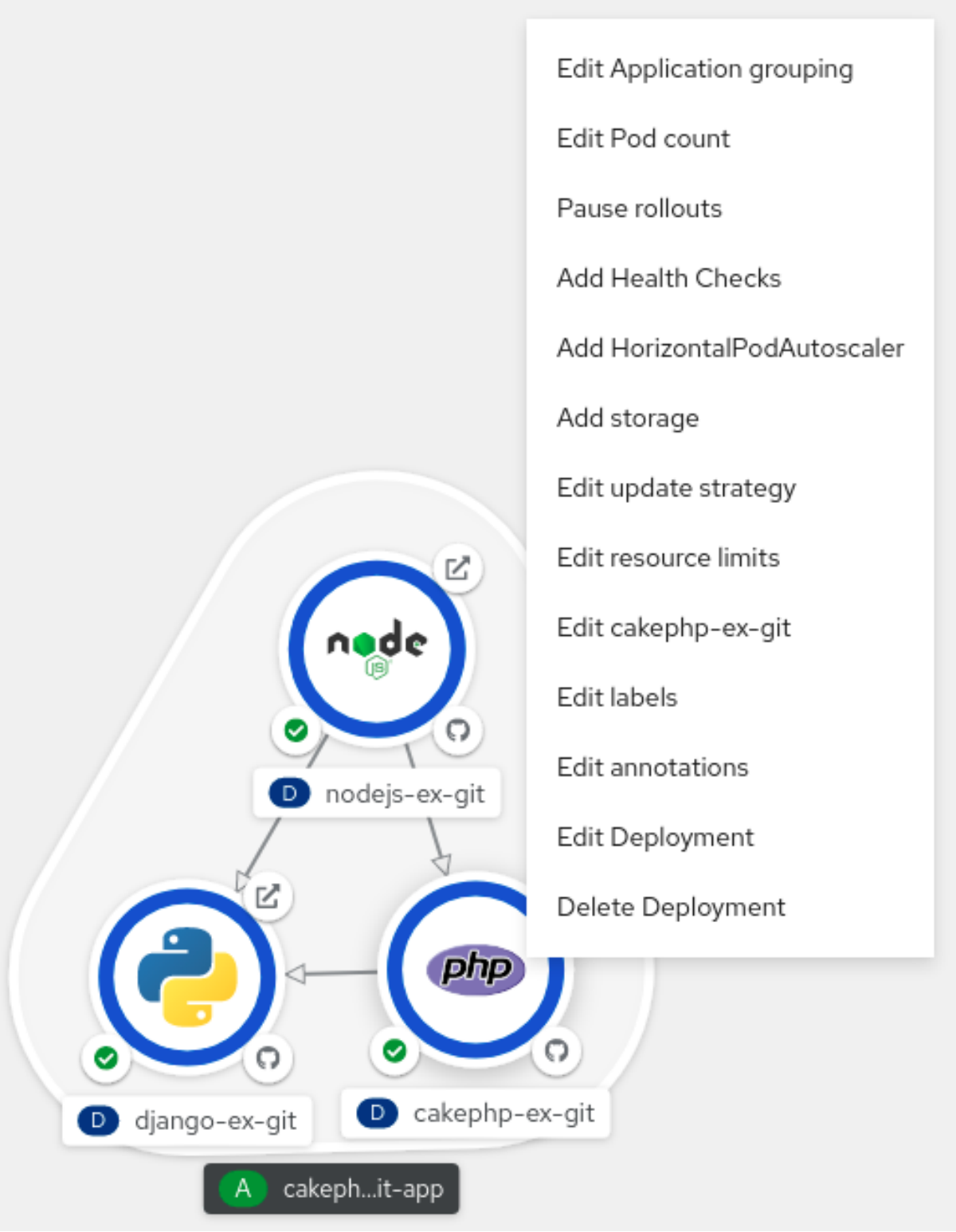
トポロジー **Graph view** のコンテキストメニューでアプリケーションからサービスを削除します。

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。

Red Hat OpenShift Service on AWS 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。

第5章 HELM チャートの使用

5.1. HELM について

Helm は、アプリケーションやサービスの Red Hat OpenShift Service on AWS 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、Red Hat OpenShift Service on AWS リソースを記述するファイルのコレクションです。

クラスターにチャートを作成すると、**リリース** と呼ばれる、チャートの実行中のインスタンスが作成されます。

チャートが作成されるか、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。

5.1.1. 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。
- Red Hat OpenShift Service on AWS 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。

5.1.2. OpenShift の Helm チャートの Red Hat 認定

Red Hat OpenShift Service on AWS 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。

5.1.3. 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。
- チャートのリストは、Red Hat **Helm index** ファイル を参照してください。

5.2. 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、Red Hat OpenShift Service on AWS 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。

5.2.1. Linux の場合

1. Linux x86_64 または Linux amd64 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-linux-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"}
```

5.2.2. 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。

5.2.3. 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** をクリックし、**env** または **environment** と入力します。
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。

5.2.4. 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"}
```

5.3. カスタム HELM チャートリポジトリーの設定

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。チャートのリストは、Red Hat **Helm index** 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。

5.3.1. 開発者パースペクティブを使用した Helm リリースの作成

Web コンソールの **Developer** パースペクティブまたは CLI を使用して、**Developer Catalog** にリストされている Helm チャートからリリースを選択して作成できます。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

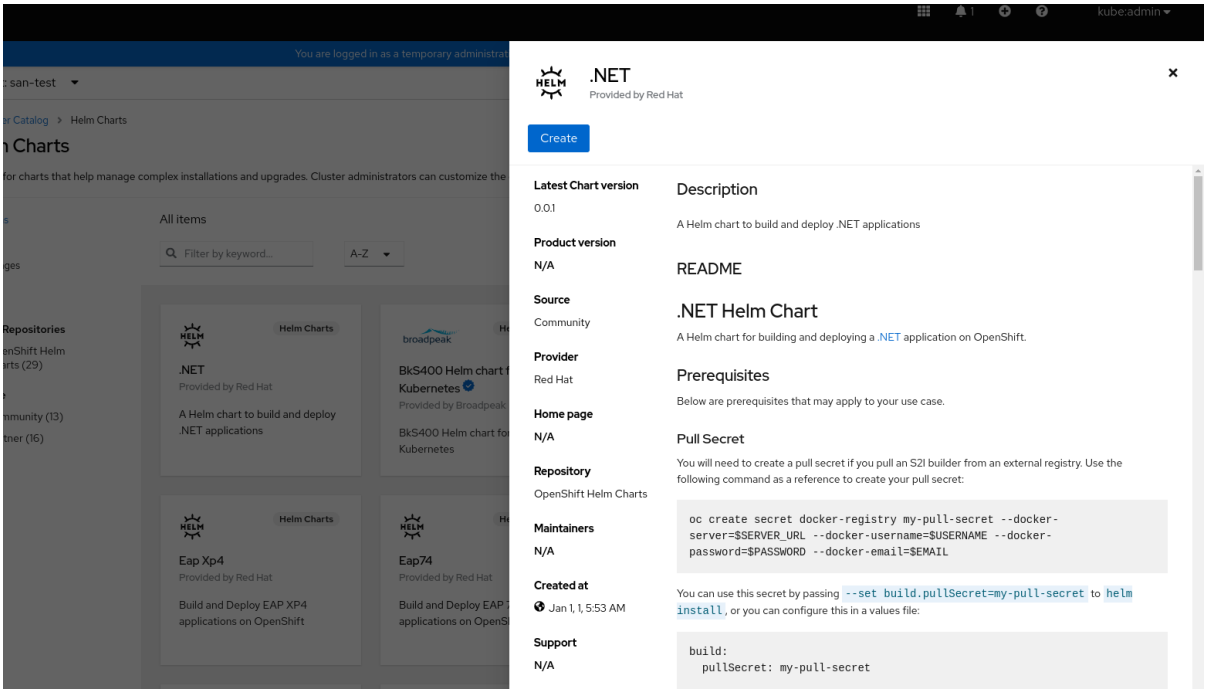
- Web コンソールにログインし、Developer パースペクティブ に切り替えている。

手順

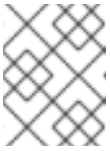
Developer Catalog で提供される Helm チャートから Helm リリースを作成するには、以下を実行します。

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、チャートに関するその他の詳細を確認します。
3. Create をクリックします。

図5.1 Developer カタログの Helm チャート



4. Create Helm Release ページで:
   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. Create をクリックして Helm リリースを作成します。Web コンソールは、Topology ビューに新しいリリースを表示します。
   Helm チャートにリリースノートがある場合は、Web コンソールに表示されます。

Helm チャートがワークロードを作成する場合、Web コンソールはそれらを **Topology** または **Helm リリース詳細** ページに表示します。ワークロードは、**DaemonSet**、**CronJob**、**Pod**、**Deployment**、および **DeploymentConfig** です。

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。

サイドパネルの **Actions** ボタンを使用するか、Helm リリースを右クリックして、Helm リリースをアップグレード、ロールバック、または削除できます。

5.3.2. 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。

5.3.3. Red Hat OpenShift Service on AWS でのカスタム Helm チャートの作成

手順

1. プロジェクトを新規作成します。

```
$ oc new-project nodejs-ex-k
```

2. Red Hat OpenShift Service on AWS オブジェクトを含むサンプル Node.js チャート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
```

5.3.4. 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。

5.4. 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て、Helm リリースを更新、ロールバック、または削除できます。

5.4.1. 前提条件

- 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切り替えている。

5.4.2. 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。

5.4.3. Helm リリースのロールバック

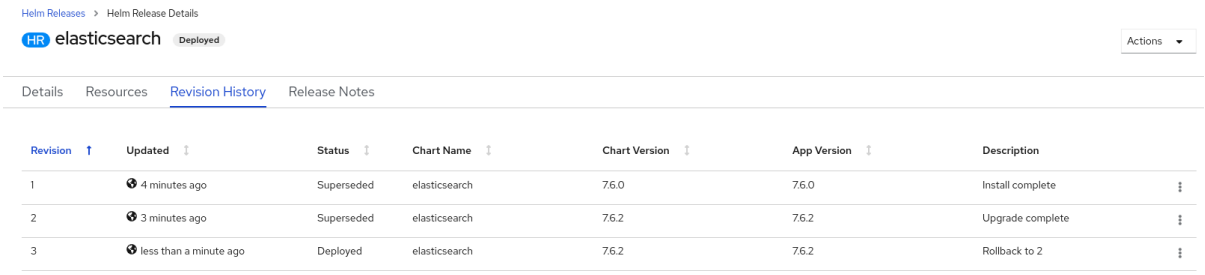
リリースに失敗する場合、Helm リリースを直前のバージョンにロールバックできます。

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5.2 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。

5.4.4. Helm リリースの削除

手順

1. **Topology** ビューで Helm リリースを右クリックし、 **Delete Helm Release** を選択します。
2. 確認プロンプトで、グラフの名前を入力し、**Delete** をクリックします。

第6章 デプロイメント

6.1. アプリケーションのカスタムドメイン

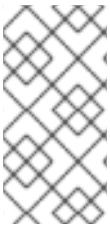
警告

Red Hat OpenShift Service on AWS 4.14 以降、Custom Domain Operator は非推奨になりました。Red Hat OpenShift Service on AWS 4.14 で Ingress を管理するには、Ingress Operator を使用します。Red Hat OpenShift Service on AWS 4.13 以前のバージョンでは機能に変更はありません。

アプリケーションのカスタムドメインを設定できます。カスタムドメインは、Red Hat OpenShift Service on AWS アプリケーションで使用できる特定のワイルドカードドメインです。

6.1.1. アプリケーションのカスタムドメインの設定

トップレベルのドメイン (TLD) は、Red Hat OpenShift Service on AWS クラスターを運用しているお客様が所有しています。カスタムドメイン Operator は、2 日目の操作としてカスタム証明書を使用して新規イングレスコントローラーを設定します。次に、このイングレスコントローラーのパブリック DNS レコードを外部 DNS で使用して、カスタムドメインで使用するワイルドカード CNAME レコードを作成できます。

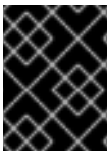


注記

Red Hat は API ドメインを制御するため、カスタム API ドメインはサポートされません。ただし、お客様はアプリケーションドメインを変更することができます。プライベート **IngressController** があるプライベートカスタムドメインの場合は、**CustomDomain** CR で **.spec.scope** を **Internal** に設定します。

前提条件

- **dedicated-admin** 権限を持つユーザーアカウント
- ***.apps.<company_name>.io** などの一意のドメインまたはワイルドカードドメイン
- **CN=*.apps.<company_name>.io** などのカスタム証明書またはワイルドカードカスタム証明書
- 最新バージョンの **oc** CLI がインストールされているクラスターへのアクセス



重要

CustomDomain CR の **metadata/name:** セクションで、予約された名前 **default** または **apps*** (**apps** や **apps2** など) を使用しないでください。

手順

1. 秘密鍵および公開証明書から新しい TLS シークレットを作成します。ここで、**fullchain.pem** および **privkey.pem** は、公開または秘密のワイルドカード証明書です。

例

```
$ oc create secret tls <name>-tls --cert=fullchain.pem --key=privkey.pem -n <my_project>
```

2. 新規の **CustomDomain** カスタムリソース (CR) を作成します。

例: <company_name>-custom-domain.yaml

```
apiVersion: managed.openshift.io/v1alpha1
kind: CustomDomain
metadata:
 name: <company_name>
spec:
 domain: apps.<company_name>.io 1
 scope: External
 loadBalancerType: Classic 2
 certificate:
  name: <name>-tls 3
  namespace: <my_project>
 routeSelector: 4
  matchLabels:
   route: acme
 namespaceSelector: 5
  matchLabels:
   type: sharded
```

1. カスタムドメイン。
2. カスタムドメインのロードバランサーのタイプ。このタイプは、ネットワークロードバランサーを使用する場合は、デフォルトの **classic** または **NLB** にすることができます。
3. 前の手順で作成されたシークレット
4. オプション: CustomDomain イングレスによって提供されるルートのセットをフィルタリングします。値が指定されていない場合、デフォルトはフィルタリングなしです。
5. オプション: CustomDomain イングレスによって提供される namespace のセットをフィルタリングします。値が指定されていない場合、デフォルトはフィルタリングなしです。

3. CR を適用します。

例

```
$ oc apply -f <company_name>-custom-domain.yaml
```

4. 新規に作成された CR のステータスを取得します。

```
$ oc get customdomains
```

出力例

```
NAME            ENDPOINT                                                DOMAIN                  STATUS
<company_name>  xxrywp.<company_name>.cluster-01.opln.s1.openshiftapps.com  *.apps.<company_name>.io  Ready
```

5. エンドポイントの値を使用して、新規のワイルドカード CNAME レコードセットを、Route53、Azure DNS、Google DNS などの管理 DNS プロバイダーに追加します。

例

```
*.apps.<company_name>.io -> xxrywp.<company_name>.cluster-01.opln.s1.openshiftapps.com
```

6. 新規アプリケーションを作成し、これを公開します。

例

```
$ oc new-app --docker-image=docker.io/openshift/hello-openshift -n my-project
```

```
$ oc create route <route_name> --service=hello-openshift hello-openshift-tls --hostname hello-openshift-tls-my-project.apps.<company_name>.io -n my-project
```

```
$ oc get route -n my-project
```

```
$ curl https://hello-openshift-tls-my-project.apps.<company_name>.io
Hello OpenShift!
```

トラブルシューティング

- Error creating TLS secret
- Troubleshooting: CustomDomain in NotReady state

6.1.2. カスタムドメインの証明書の更新

oc CLI ツールを使用して、Custom Domains Operator (CDO) で証明書を更新できます。

前提条件

- 最新バージョンの **oc** CLI ツールがインストールされている。

手順

1. 新しいシークレットを作成します。

```
$ oc create secret tls <secret-new> --cert=fullchain.pem --key=privkey.pem -n <my_project>
```

2. CustomDomain CR にパッチを適用します。

```
$ oc patch customdomain <company_name> --type='merge' -p '{"spec":{"certificate":{"name":"<secret-new>"}}}'
```

3. 古いシークレットを削除します。

```
$ oc delete secret <secret-old> -n <my_project>
```

トラブルシューティング

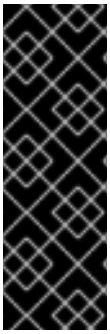
- Error creating TLS secret

6.2. デプロイメントの理解

Red Hat OpenShift Service on AWS 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 オブジェクトで構成されています。

- 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。



重要

Red Hat OpenShift Service on AWS 4.14 では、**DeploymentConfig** オブジェクトは非推奨になりました。**DeploymentConfig** オブジェクトは引き続きサポートされていますが、新規インストールには推奨されません。セキュリティー関連の重大な問題のみが修正されます。

代わりに、**Deployment** オブジェクトまたは別の代替手段を使用して、Pod の宣言的更新を提供します。

6.2.1. 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。

6.2.1.1. 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。

6.2.1.2. レプリケーションコントローラー

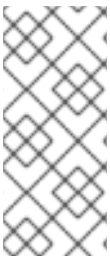
レプリカセットと同様に、レプリケーションコントローラーは、Pod の指定された数のレプリカが常に実行されるようにします。Pod が終了または削除された場合に、レプリケーションコントローラーは定義した数になるまでインスタンス化する数を増やします。同様に、必要以上の数の Pod が実行されている場合には、定義された数に一致させるために必要な数の Pod を削除します。レプリカセットとレプリケーションコントローラーの相違点は、レプリカセットではセットベースのセレクター要件をサポートし、レプリケーションコントローラーは等価ベースのセレクター要件のみをサポートする点です。

レプリケーションコントローラー設定は以下で構成されています。

- 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。

6.2.2. デプロイメント

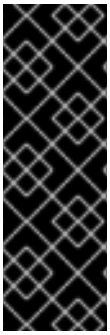
Kubernetes は、**Deployment** と呼ばれるファーストクラスのネイティブ API オブジェクトタイプを Red Hat OpenShift Service on AWS に提供します。**Deployment** オブジェクトは、Pod テンプレートとして、アプリケーションの特定のコンポーネントで希望する状態を記述します。デプロイメントは、Pod のライフサイクルをオーケストレーションするレプリカセットを作成します。

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
```

6.2.3. DeploymentConfig オブジェクト



重要

Red Hat OpenShift Service on AWS 4.14 では、**DeploymentConfig** オブジェクトは非推奨になりました。**DeploymentConfig** オブジェクトは引き続きサポートされていますが、新規インストールには推奨されません。セキュリティー関連の重大な問題のみが修正されます。

代わりに、**Deployment** オブジェクトまたは別の代替手段を使用して、Pod の宣言的更新を提供します。

レプリケーションコントローラーでビルドする Red Hat OpenShift Service on AWS 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レプリケーションコントローラーを作成し、それに Pod を起動させます。

ただし、**DeploymentConfig** オブジェクトの Red Hat OpenShift Service on AWS 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おけるフックの実行も定義します。

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は確実に通常通りに完了することができます。

Red Hat OpenShift Service on AWS **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。

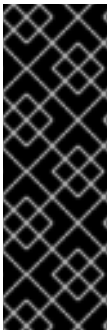
2 イメージ変更トリガーにより、新規デプロイメントが、バッキングイメージの新規バージョンが名前付きイメージストリームで利用可能になる際には常に作成されます。

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。

6.2.4. Deployment および DeploymentConfig オブジェクトの比較

Kubernetes **Deployment** オブジェクトおよび Red Hat OpenShift Service on AWS に提供される **DeploymentConfig** オブジェクトの両方が Red Hat OpenShift Service on AWS 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は、**Deployment** オブジェクト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。



重要

Red Hat OpenShift Service on AWS 4.14 では、**DeploymentConfig** オブジェクトは非推奨になりました。**DeploymentConfig** オブジェクトは引き続きサポートされていますが、新規インストールには推奨されません。セキュリティー関連の重大な問題のみが修正されます。

代わりに、**Deployment** オブジェクトまたは別の代替手段を使用して、Pod の宣言的更新を提供します。

6.2.4.1. 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、デプロイメントの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。

6.2.4.2. デプロイメント固有の機能

6.2.4.2.1. 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。

6.2.4.2.2. 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関する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。

6.2.4.2.3. 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。

6.2.4.3. DeploymentConfig オブジェクト固有の機能

6.2.4.3.1. 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。

6.2.4.3.2. 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>
```

6.2.4.3.3. 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。

6.2.4.3.4. 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。

6.3. デプロイメントプロセスの管理

6.3.1. DeploymentConfig オブジェクトの管理

重要

Red Hat OpenShift Service on AWS 4.14 では、**DeploymentConfig** オブジェクトは非推奨になりました。**DeploymentConfig** オブジェクトは引き続きサポートされていますが、新規インストールには推奨されません。セキュリティー関連の重大な問題のみが修正されます。

代わりに、**Deployment** オブジェクトまたは別の代替手段を使用して、Pod の宣言的更新を提供します。

DeploymentConfig オブジェクトは、Red Hat OpenShift Service on AWS Web コンソールの **Workloads** ページからか、または **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。

6.3.1.1. 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。

6.3.1.2. 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関する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** オブジェクトに関する最近作成されたすべてのレプリケーションコントローラーの詳細を表示するには、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は、**oc describe** コマンドを使用します。

```
$ oc describe dc <name>
```

6.3.1.3. デプロイメントの再試行

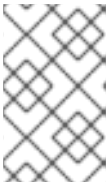
現行リビジョンの **DeploymentConfig** がデプロイに失敗した場合、デプロイメントプロセスを再起動することができます。

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。

6.3.1.4. 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。

6.3.1.5. 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
# ...
```

6.3.1.6. 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>
```

6.3.1.7. 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。

6.3.1.7.1. 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"
```

6.3.1.7.2. 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。

6.3.1.7.3. デプロイメントトリガーの設定

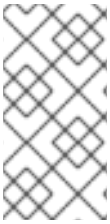
手順

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>
```

6.3.1.8. 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、Rolling または Custom 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。

6.3.1.9. 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。

6.3.1.10. 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、Red Hat OpenShift Service on AWS 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** オブジェクトエディターページで、**Pull Secret** を設定し、変更を保存します。

6.3.1.11. 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>
```

6.4. 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。

6.4.1. 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。

6.4.2. 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。

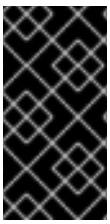
(6) **pre** および **post** はどちらもライフサイクルフックです。

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。

警告

Red Hat OpenShift Service on AWS のすべてのマシン設定プールにおける **maxUnavailable** のデフォルト設定は **1** です。この値を変更せず、一度に 1 つのコントロールプレーンノードを更新することを推奨します。コントロールプレーンプールのこの値を **3** に変更しないでください。

6.4.2.1. canary デプロイメント

Red Hat OpenShift Service on AWS 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。

6.4.2.2. 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 Red Hat OpenShift Service on AWS 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。新しい Pod が (readiness チェックに合格して) **ready** とマークされると、デプロイメントプロセスが続行されます。

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。

6.4.2.3. 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。

6.4.2.4. 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。

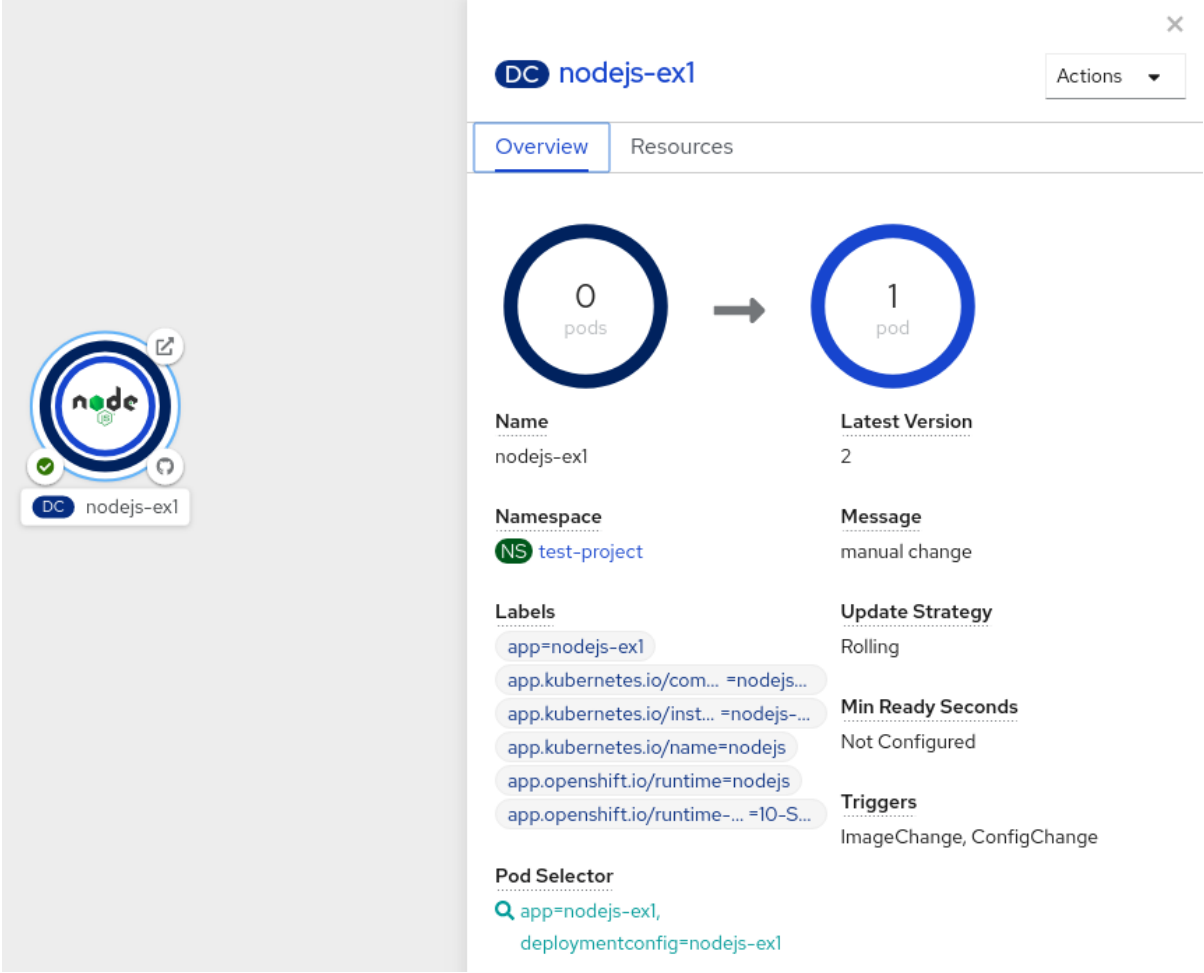
前提条件

- 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6.1 ローリング更新



追加リソース

- Developer パースペクティブを使用して Red Hat OpenShift Service on AWS 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

6.4.3. 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。

6.4.3.1. 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

- 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。

6.4.3.2. 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。

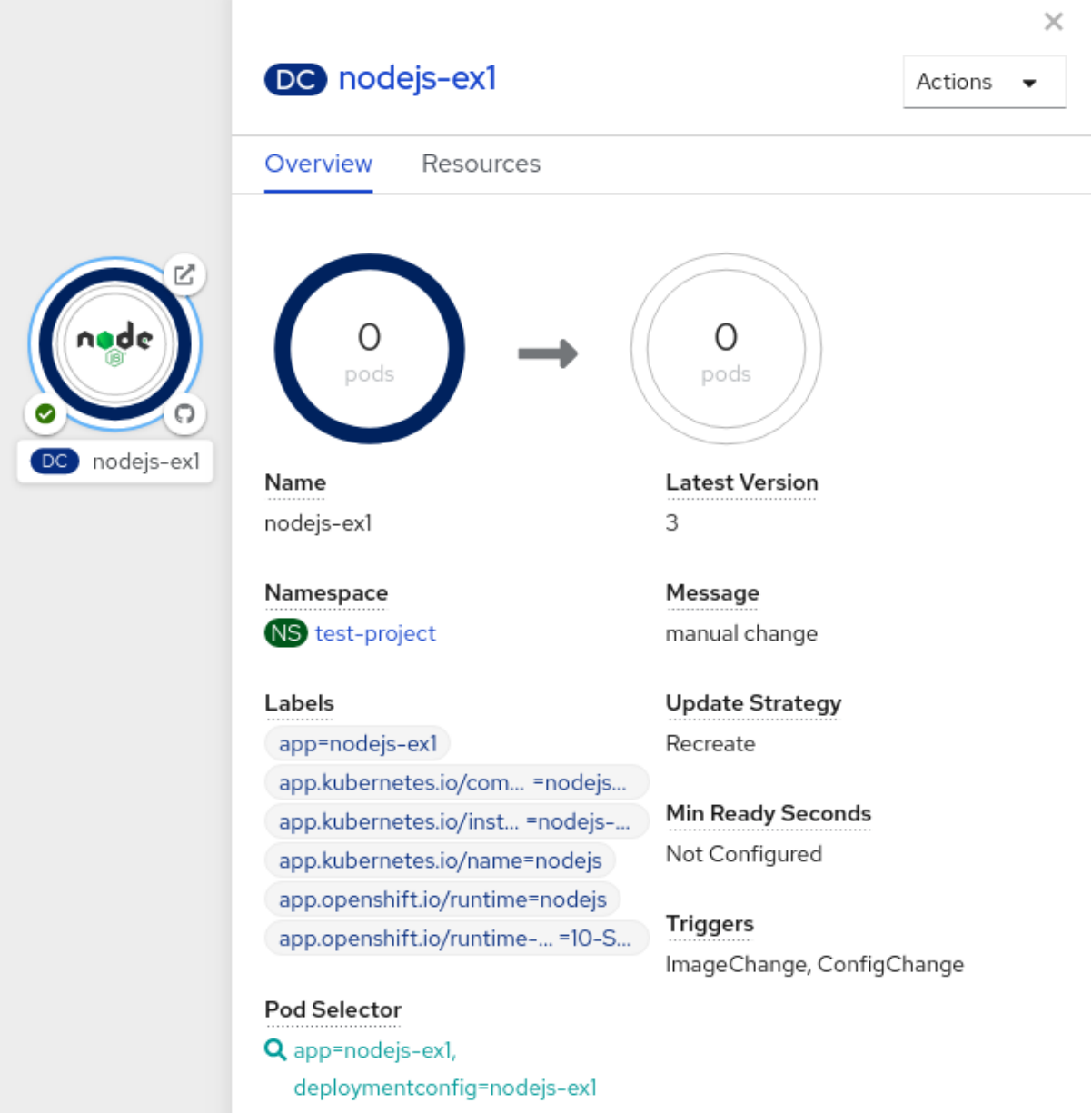
2. **Actions** ドロップダウンメニューで、**Edit Deployment Config** を選択し、アプリケーションのデプロイメント設定の詳細を確認します。

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6.2 再作成更新



追加リソース

- Developer パースペクティブを使用して Red Hat OpenShift Service on AWS 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

6.4.4. 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** ディレクティブをオーバーライドします。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、Red Hat OpenShift Service on AWS 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 Red Hat OpenShift Service on AWS 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、Red Hat OpenShift Service on AWS 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。

6.4.4.1. 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。

6.4.5. 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。

6.4.5.1. Pod ベースのライフサイクルフック

Pod ベースのライフサイクルフックは、**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** をオーバーライドします。

(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。

6.4.5.2. 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
```

6.5. 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。

あるいは、両方のバージョンが同時にアクティブになる **A/B バージョン** ストラテジーを使用することもできます。このストラテジーでは、一部のユーザーは **バージョン A** を使用し、他のユーザーは **バージョン B** を使用できます。このストラテジーを使用すると、ユーザーインターフェイスの変更やその他の機能を試して、ユーザーからのフィードバックを得ることができます。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。

6.5.1. 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、Red Hat OpenShift Service on AWS 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。

6.5.2. 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。

6.5.3. 正常な終了

Red Hat OpenShift Service on AWS 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も正常に終了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、Red Hat OpenShift Service on AWS 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。

6.5.4. 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を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。

6.5.4.1. 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。

6.5.5. 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。

Red Hat OpenShift Service on AWS は、Web コンソールと CLI で N-1 互換性をサポートします。

6.5.5.1. 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
   ```

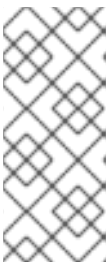
   どちらのアプリケーションもデプロイされ、サービスが作成されます。

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。ルートアノテーションの詳細は、**関連情報** セクションを参照してください。

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:
  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
# ...
```

6.5.5.1.1. 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 Options 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。

6.5.5.1.2. 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。

6.5.5.1.3. 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. 負荷分散アルゴリズムのデフォルト値をオーバーライドするには、アルゴリズムを **roundrobin** に設定してルートのアノテーションを調整します。Red Hat OpenShift Service on AWS のルートの場合、デフォルトの負荷分散アルゴリズムは **random** 値または **source** 値に設定されます。
アルゴリズムを **roundrobin** に設定するには、次のコマンドを実行します。

```
$ oc annotate routes/<route-name> haproxy.router.openshift.io/balance=roundrobin
```

Transport Layer Security (TLS) パススルールートの場合、デフォルト値は **source** です。他のすべてのルートの場合、デフォルトは **random** です。

4. **--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。

6.5.5.1.4.1 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7章 クォータ

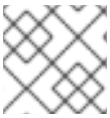
7.1. プロジェクトごとのリソースクォータ

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。

このガイドでは、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を説明します。

7.1.1. 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を説明します。



注記

status.phase in (Failed, Succeeded) が true の場合、Pod は終了状態にあります。

表7.1 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7.2 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永続ボリューム要求の合計数です。
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7.3 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。

7.1.2. 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** のいずれかに関するサービスの QoS (Quality of Service) が Best Effort の Pod に一致します。
NotBestEffort	**cpu** および **memory** に関するサービスの 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**

7.1.3. 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。

7.1.4. 要求と制限

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。

7.1.5. 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 1Gi 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 2Gi 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 2Gi 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 4Gi を超えることができません。

7.1.6. 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
```

7.1.6.1. 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、Red Hat OpenShift Service on AWS 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.apps=2,count/replicasets.apps=4,count/pods=3,count/secrets=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.apps      0     2
count/pods                  0     3
count/replicasets.apps      0     4
count/secrets               0     4
```

7.1.6.2. 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関するシナリオ例です。

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。

7.1.7. 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情報を表示できます。

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. 関連するクォータを記述します。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
```

7.1.8. 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。

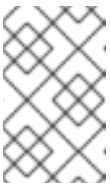
```
$ oc create -f template.yaml -n openshift-config
```

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>
```

7.2. 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を説明します。

重要

デフォルトプロジェクトでワークロードを実行したり、デフォルトプロジェクトへのアクセスを共有したりしないでください。デフォルトのプロジェクトは、コアクラスターコンポーネントを実行するために予約されています。

デフォルトプロジェクトである **default**、**kube-public**、**kube-system**、**openshift**、**openshift-infra**、**openshift-node**、および **openshift.io/run-level** ラベルが **0** または **1** に設定されているその他のシステム作成プロジェクトは、高い特権があるとみなされます。Pod セキュリティーアドミッション、Security Context Constraints、クラスターリソースクォータ、イメージ参照解決などのアドミッションプラグインに依存する機能は、高い特権を持つプロジェクトでは機能しません。

7.2.1. 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
```

7.2.2. 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
```

7.2.3. 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8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。

8.1. 設定マップについて

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。Red Hat OpenShift Service on AWS 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 Red Hat OpenShift Service on AWS 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注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。

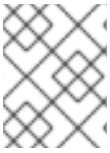
ConfigMap オブジェクトは、Pod で使用したり、コントローラーなどのシステムコンポーネントの設定データを保存するために使用できる設定データのキーと値のペアを保持します。以下に例を示します。

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。

8.1.1. 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、Red Hat OpenShift Service on AWS ノードの **--manifest-url** フラグ、**--config** フラグ、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ないためです。

関連情報

- 設定マップの作成および使用

8.2. 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を説明します。

8.2.1. 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、以下の設定マップを見てみましょう。

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 注入する環境変数。

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 注入する環境変数。

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。

  特定の環境変数を注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:
  securityContext:
    runAsNonRoot: true
    seccompProfile:
      type: RuntimeDefault
  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
    securityContext:
      allowPrivilegeEscalation: false
      capabilities:
        drop: [ALL]
  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 キーの値を注入する 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。

8.2.2. 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、以下の設定マップを見てみましょう。

```
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注入するには、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。

  特定の環境変数を注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:
  securityContext:
    runAsNonRoot: true
    seccompProfile:
      type: RuntimeDefault
  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
      securityContext:
        allowPrivilegeEscalation: false
        capabilities:
          drop: [ALL]
  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
```

8.2.3. 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

config map を使用して、コンテンツをボリュームに注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

config map を使用してコンテンツをボリュームに注入するには、2 つの異なるオプションを使用できます。

- config map を使用してコンテンツをボリュームに注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:
  securityContext:
    runAsNonRoot: true
    seccompProfile:
      type: RuntimeDefault
  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
      securityContext:
        allowPrivilegeEscalation: false
        capabilities:
          drop: [ALL]
  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:
```

```
  securityContext:
    runAsNonRoot: true
    seccompProfile:
      type: RuntimeDefault
  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
      securityContext:
        allowPrivilegeEscalation: false
        capabilities:
          drop: [ALL]
  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9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。

9.1. 前提条件

- Red Hat OpenShift Service on AWS 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ており、**Developer** パースペクティブに切り替えている。

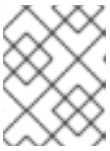
9.2. プロジェクトメトリクスのモニタリング

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をクエリーします。

図9.1 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。

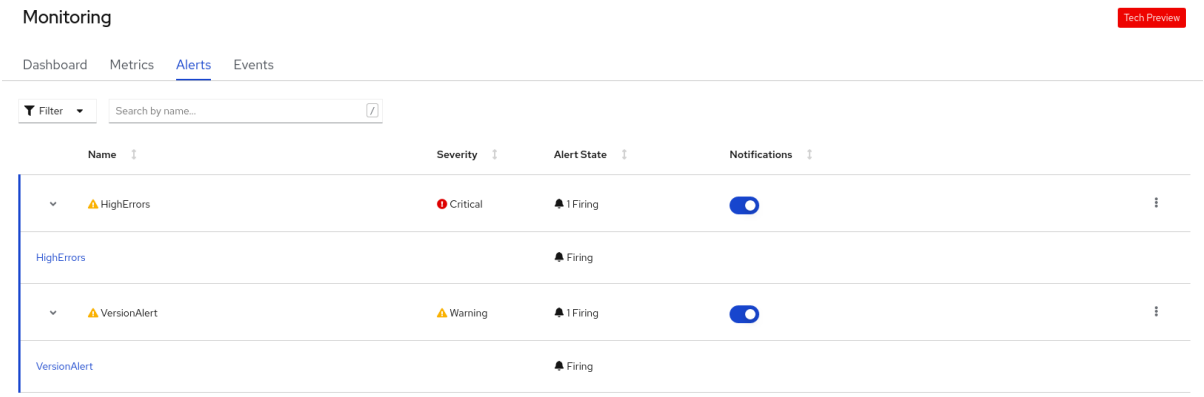
d. ドロップダウンリストを使用して、表示されるデータの時間の範囲を設定します。**Reset Zoom** をクリックして、これをデフォルトの時間の範囲にリセットできます。

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9.2 アラートのモニタリング

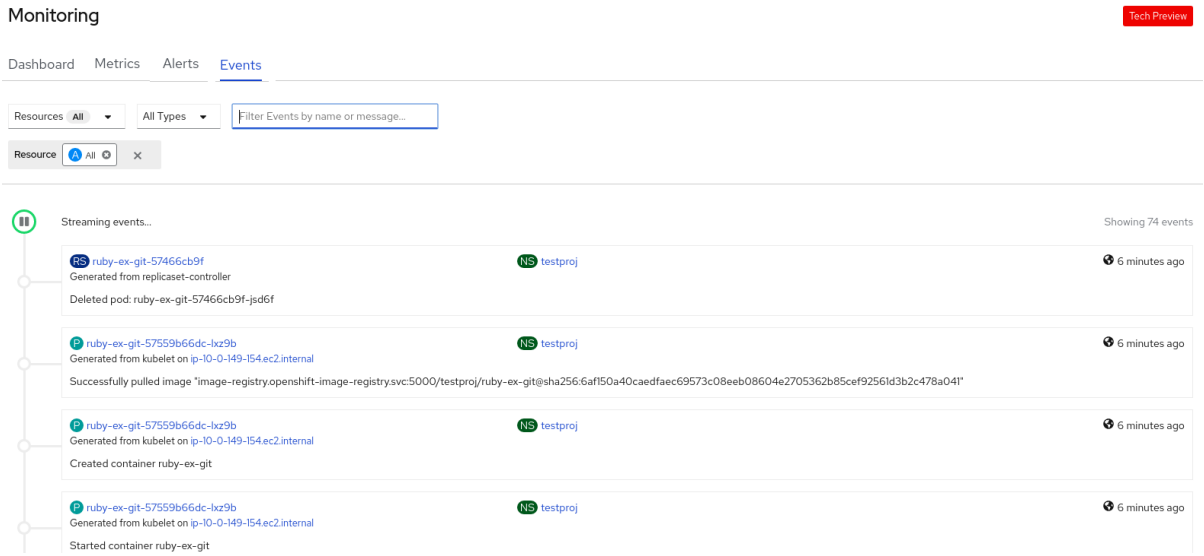


詳細は、以下のオプションを使用します。

- **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9.3 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。

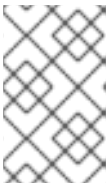
9.3. アプリケーションメトリクスのモニタリング

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** ペースペクティブで **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** ビューのワークロードノードに示されます。

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。

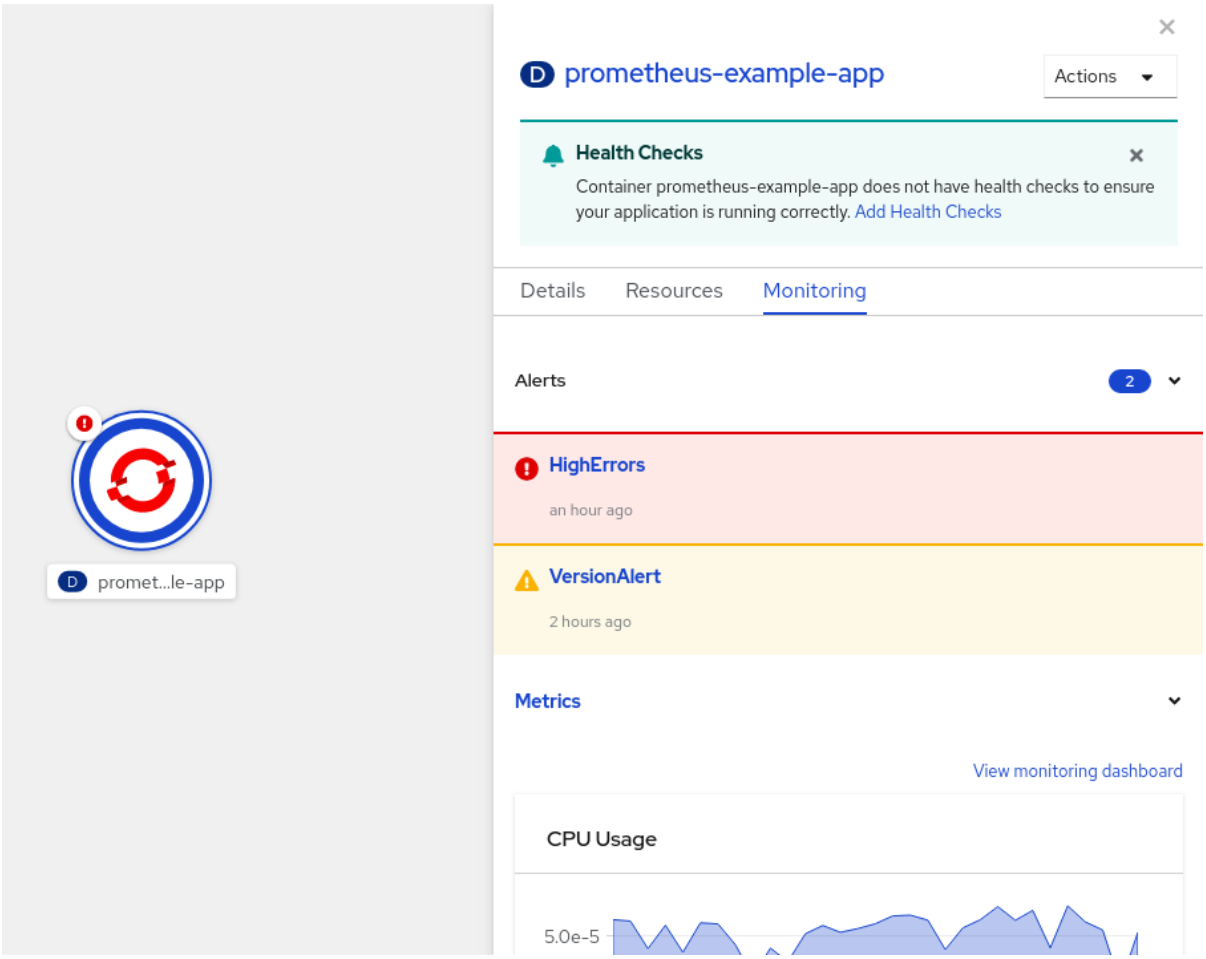
1. **Topology** ビューで、ワークロードをクリックし、ワークロードの詳細を右側のパネルに表示します。
2. **Observe** タブをクリックして、アプリケーションの重大な問題および警告のアラート、CPU、メモリー、および帯域幅の使用状況などのメトリクスのグラフ、およびアプリケーションのすべてのイベントを表示します。



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9.4 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。

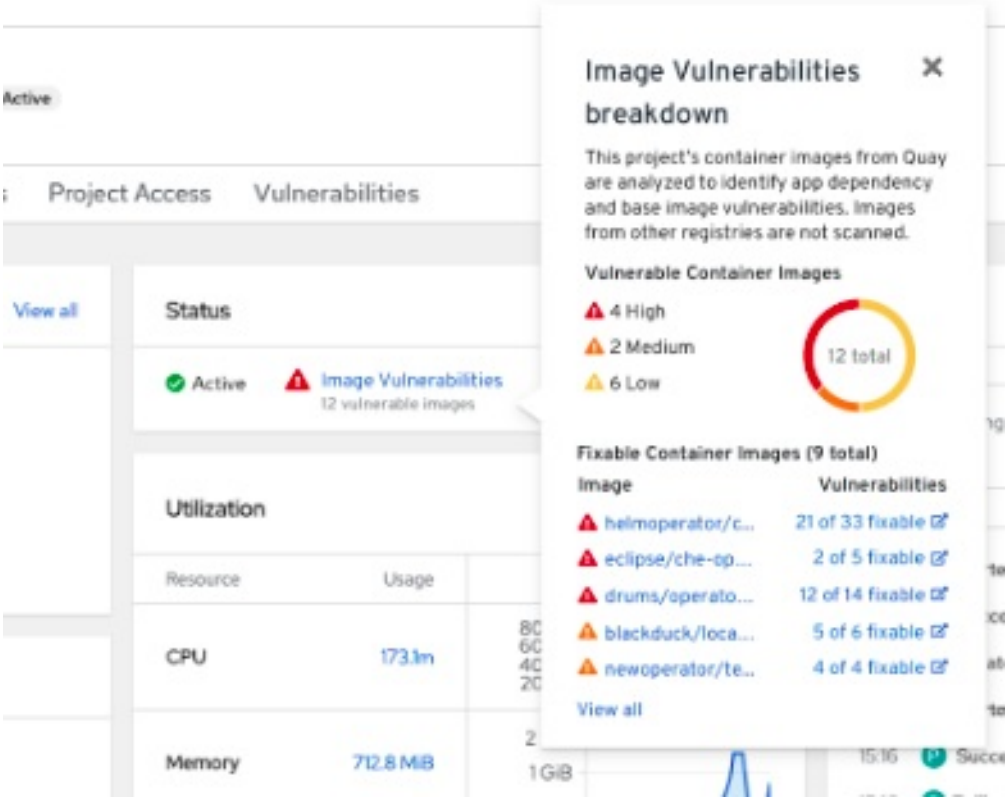
9.4. イメージの脆弱性の内訳

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9.5 イメージ脆弱性の表示



9.5. 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 Security Operator をインストールした。

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。

9.6. 関連情報

- Red Hat OpenShift Service on AWS モニタリングについて

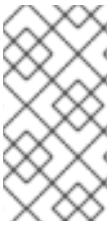
第10章 ヘルスチェックの使用によるアプリケーションの正常性の監視

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。Red Hat OpenShift Service on AWS 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。

10.1. 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。

Startup プローブ

startup 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。startup 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、startup プローブを liveness プローブに追加し、**failureThreshold** を失敗 30 回、**periodSeconds** を 10 秒 (30 x 10 秒 = 300 秒) に設定すると、最大 5 分間を確保できます。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。

10.1.1. 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. startup 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

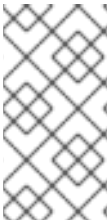
```
kind: Deployment
apiVersion: apps/v1
metadata:
  labels:
    test: health-check
  name: my-application
spec:
# ...
  template:
    spec:
      containers:
        - resources: {}
          readinessProbe: 1
            tcpSocket:
              port: 8080
            timeoutSeconds: 1
            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。

10.2. 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: Startup 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関する 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
```

10.3. 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。

10.4. 開発者パースペクティブを使用したヘルスチェックの追加

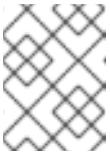
Topology ビューを使用して、デプロイされたアプリケーションにヘルスチェックを追加できます。

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。
- **Developer** パースペクティブを使用して Red Hat OpenShift Service on AWS でアプリケーションを作成し、デプロイしている。

手順

1. **Topology** ビューで、アプリケーションノードをクリックし、サイドパネルを表示します。コンテナーにヘルスチェックが追加されていない場合は、ヘルスチェックを追加するためのリンクを含む **Health Checks** 通知が表示されます。
2. 表示された通知で、**Add Health Checks** リンクをクリックします。
3. または、**Actions** リストをクリックし、**Add Health Checks** を選択します。コンテナーにヘルスチェックがすでにある場合は、add オプションの代わりに **Edit Health Checks** オプションが表示されます。
4. **Add Health Checks** フォームで複数のコンテナーをデプロイしている場合は、**Container** リストを使用して適切なコンテナーが選択されていることを確認します。
5. 必要なヘルスプローブのリンクをクリックして、それらをコンテナーに追加します。ヘルスチェックのデフォルトデータは事前に設定されています。デフォルトデータでプローブを追加するか、値をさらにカスタマイズしてから追加できます。たとえば、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 Readiness プローブを追加するには、以下を実行します。
    a. **Add Readiness 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. **Type** リストをクリックし、追加する要求タイプを選択します。たとえば、この場合は **Container Command** を選択し、コンテナー内で実行されるコマンドを選択します。
    c. **Command** フィールドで、引数 **cat** を追加することもできます。同様に、チェック用に複数の引数を追加したり、別の引数 **/tmp/healthy** を追加したりすることができます。
    d. 必要に応じて、他のパラメーターのデフォルト値を保持するか、変更します。



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。

e. 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。**Readiness Probe Added** メッセージが表示されます。

6. **Add** をクリックしてヘルスチェックを追加します。**Topology** ビューにリダイレクトされ、コンテナーが再起動します。

7. 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。

8. **Pod Details** ページで、**Containers** セクションに一覧表示されているコンテナーをクリックします。

9. **Container Details** ページで、Readiness probe - **Exec Command cat /tmp/healthy** がコンテナーに追加されていることを確認します。

10.5. 開発者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。
- **Developer** パースペクティブを使用して Red Hat OpenShift Service on AWS でアプリケーションを作成し、デプロイしている。
- 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

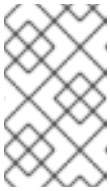
1. **Topology** ビューでアプリケーションを右クリックし、**Edit Health Checks** を選択します。または、サイドパネルで **Actions** ドロップダウンリストをクリックし、**Edit Health Checks** を選択します。

2. **Edit Health Checks** ページで以下を行います。

   - 追加されている正常性プローブを削除するには、その隣にある **Remove** アイコンをクリックします。
   - 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** セクションに一覧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。

10.6. 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。
- **Developer** パースペクティブを使用して Red Hat OpenShift Service on AWS でアプリケーションを作成し、デプロイしている。
- 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

- 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11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。

11.1. 前提条件

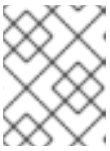
- Developer パースペクティブを使用して Red Hat OpenShift Service on AWS でアプリケーションを作成し、デプロイしている。
- Web コンソールにログインしており、Developer パースペクティブに切り替えている。

11.2. 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。



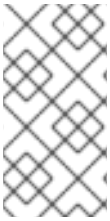
注記

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。

11.3. 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。

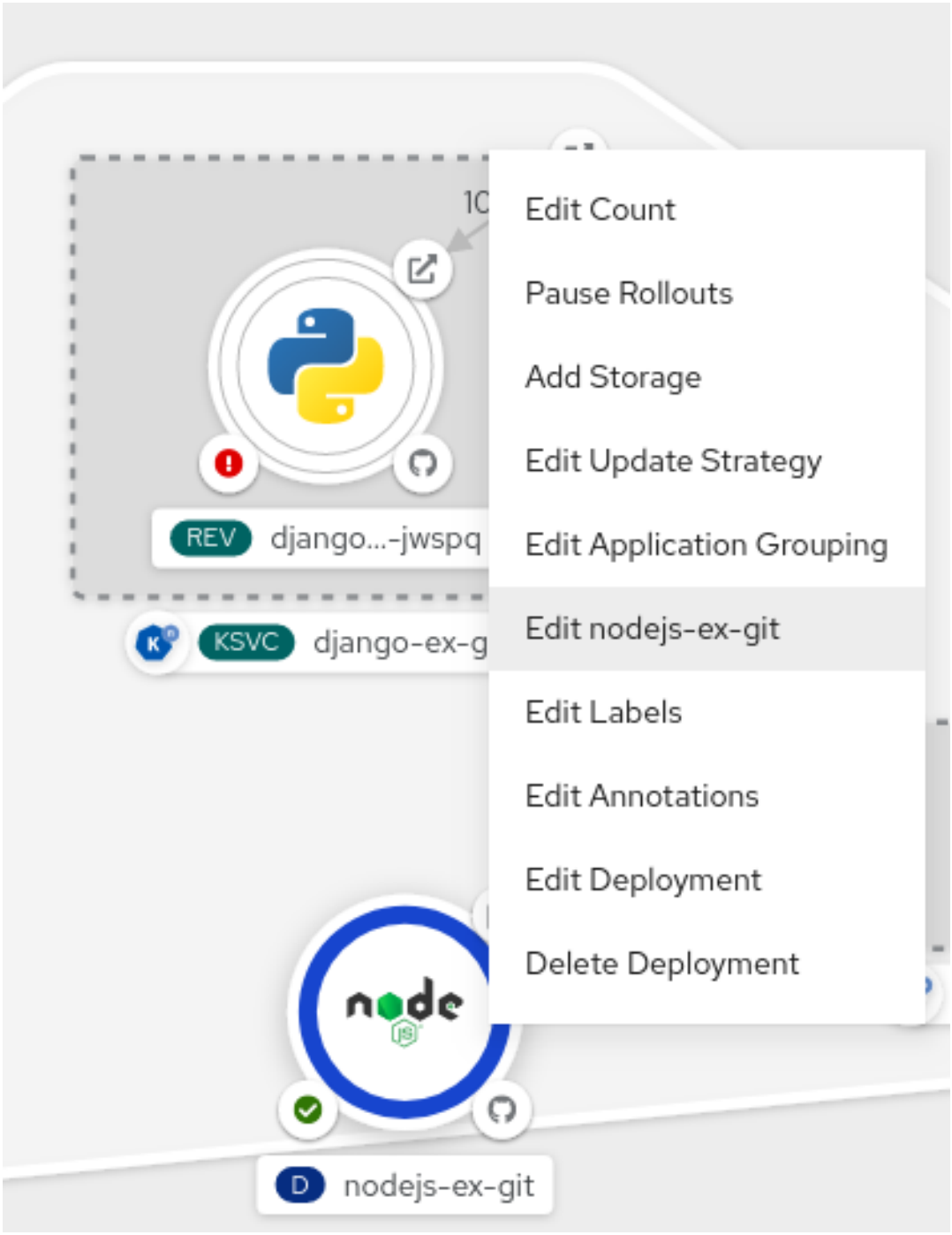
前提条件

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11.1 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。

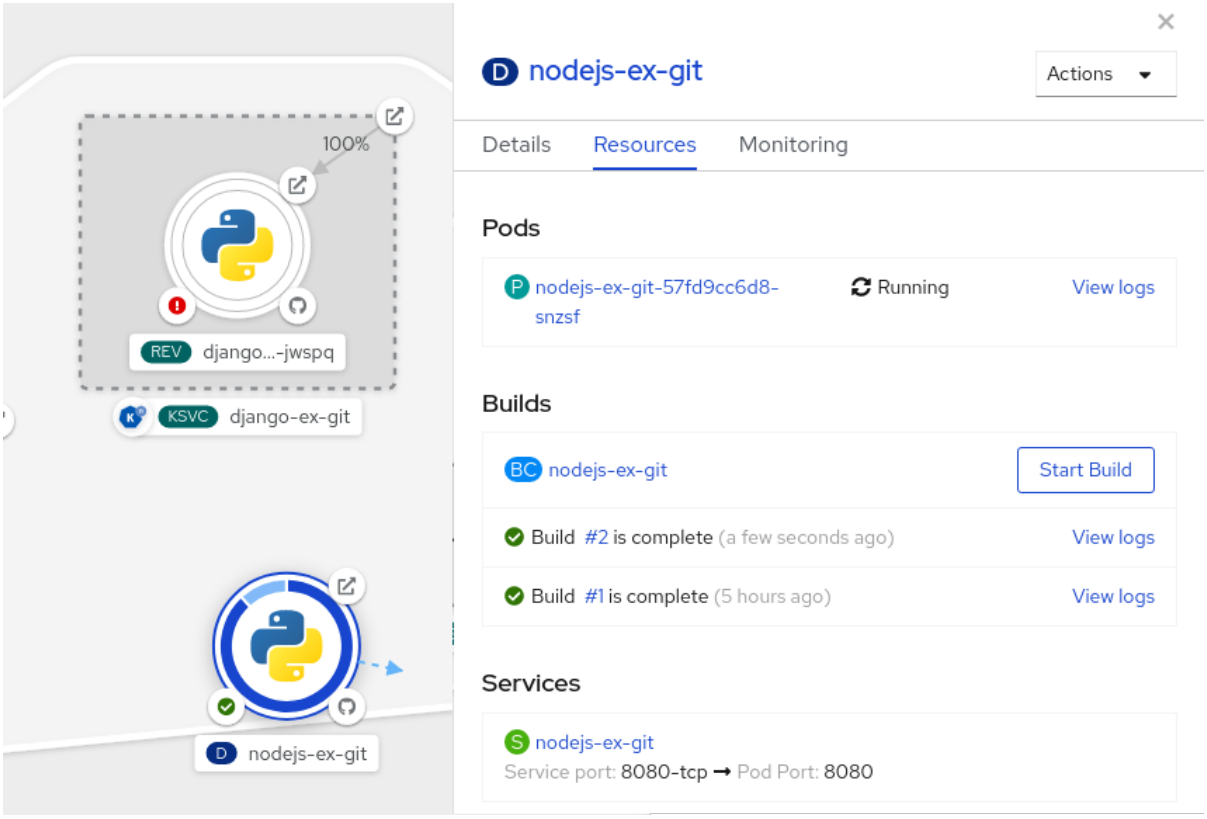
3. アプリケーションに必要な値を編集します。

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11.2 アプリケーションの編集および再デプロイ



第12章 クォータの使用

ResourceQuota オブジェクトで定義される **リソースクォータ** は、プロジェクトごとにリソース消費量の総計を制限する制約を指定します。これは、タイプ別にプロジェクトで作成できるオブジェクトの数量を制限すると共に、そのプロジェクトのリソースが消費できるコンピュートリソースおよびストレージの合計量を制限することができます。

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。リソースクォータの使用時に、オブジェクトがサーバーストレージにある場合、そのオブジェクトは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ストレージ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。

このガイドでは、リソースクォータの仕組みや、開発者がリソースを使用し、表示する方法を説明します。

12.1. クォータの表示

Web コンソールでプロジェクトの **Quota** ページに移動し、プロジェクトのクォータで定義されるハード制限に関連する使用状況の統計情報を表示できます。

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. 関連するクォータを記述します。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
```

12.2. クォータで管理されるリソース

以下では、クォータで管理できる一連のコンピュートリソースとオブジェクトタイプを説明します。

注記

status.phase in (Failed, Succeeded) が true の場合、Pod は終了状態にあります。

表12.1 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12.2 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の合計数です。

リソース名	説明
<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永続ボリューム要求の合計数です。
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12.3 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。
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。

リソース名	説明
persistentvolumeclaims	プロジェクトに存在できる永続ボリューム要求の合計数です。
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。

12.3. クォータのスコープ

各クォータには **スコープ** のセットが関連付けられます。クォータは、列挙されたスコープの交差部分に一致する場合にのみリソースの使用状況を測定します。

スコープをクォータに追加すると、クォータが適用されるリソースのセットを制限できます。許可されるセット以外のリソースを設定すると、検証エラーが発生します。

スコープ	説明
BestEffort	**cpu** または **memory** のいずれかに関するサービスの QoS (Quality of Service) が Best Effort の Pod に一致します。
NotBestEffort	**cpu** および **memory** に関するサービスの 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**

12.4. クォータの実施

プロジェクトのリソースクォータが最初に作成されると、プロジェクトは、更新された使用状況の統計が計算されるまでクォータ制約の違反を引き起こす可能性のある新規リソースの作成機能を制限します。

クォータが作成され、使用状況の統計が更新されると、プロジェクトは新規コンテンツの作成を許可します。リソースを作成または変更する場合、クォータの使用量はリソースの作成または変更要求があるとすぐに増分します。

リソースを削除する場合、クォータの使用量は、プロジェクトのクォータ統計の次回の完全な再計算時に減分されます。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。

12.5. 要求と制限

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。

第13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、Red Hat OpenShift Service on AWS 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。

dedicated-admin ロールを持つユーザーは、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す。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。

13.1. 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>**。

13.2. 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13.1 oc 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
```

13.3. 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13.2 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。

オプション	説明
--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。 **重要** このフラグはテクノロジープレビュー機能です。

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 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
```

13.4. 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13.3 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、failed、error、または canceled のすべてのビルドをプルーニングします。
--keep-complete=<N>	ビルド設定に基づいて、ステータスが complete 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。

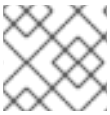
手順

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

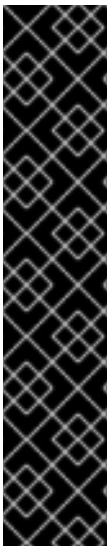
開発者は、ビルドの設定を変更して自動ビルドプルーニングを有効にできます。

13.5. 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、カスタムリソース (CR) のプルーニングを設定したり、保留にしたりできます。

前提条件

- **dedicated-admin** パーミッションを持つアカウントを使用して Red Hat OpenShift Service on AWS 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。



重要

プルーナーを管理するための Image Registry Operator の動作は、Image Registry Operator の **ClusterOperator** オブジェクトで指定された **managementState** とは無関係です。Image Registry 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、Image Registry 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、etcd のイメージメタデータのみプルーニングされます。

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 * * *
  suspend: false
```

```
    keepTagRevisions: 3
    keepYoungerThanDuration: 60m
    keepYoungerThan: 3600000000000
    resources: {}
    affinity: {}
    nodeSelector: {}
    tolerations: []
    successfulJobsHistoryLimit: 3
    failedJobsHistoryLimit: 3
  status:
    observedGeneration: 2
    conditions:
    - 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."
```

- **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 daily で午前 0 時でに設定されます。
- **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。
- **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。
- **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。
- **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。
- **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。
- **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。
- **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。
- **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。

- **successfulJobsHistoryLimit**: 保持する成功したジョブの最大数です。メトリクスがレポートされるようにするには、**1** 以上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。
- **failedJobsHistoryLimit**: 保持する失敗したジョブの最大数です。メトリクスがレポートされるようにするには、**1** 以上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。
- **observedGeneration**: Operator によって観察される生成です。
- **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。

13.6. 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

- 複数のプロジェクト間のリソースクォータ

第14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、Red Hat OpenShift Service on AWS 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構成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。

14.1. 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。

14.1.1. 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>
```

14.1.2. 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。

14.2. 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。

第15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。

15.1. 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。