OpenShift Container Platform 4.1

アプリケーション

OpenShift Container Platform 4.1 でのアプリケーションの作成および管理

Last Updated: 2020-05-15

OpenShift Container Platform 4.1 アプリケーション

OpenShift Container Platform 4.1 でのアプリケーションの作成および管理

法律上の通知

概要

本書では、OpenShift Container Platform で実行されるユーザーによってプロビジョニングされたアプリケーションのインスタンスを作成し、管理する各種の方法について説明します。さらに本書では、Open Service Broker API を使用したアプリケーションのプロビジョニングや、Operator Framework を使用したアプリケーションのビルドに関連する概念およびタスクについて説明します。

目次

第1章 プロジェクト **4**
1.1. プロジェクトの使用 4
1.2. 別のユーザーとしてのプロジェクトの作成 6
1.3. プロジェクト作成の設定 7

第2章 OPERATOR **12**
2.1. OPERATOR について 12
2.2. OPERATOR LIFECYCLE MANAGER について 14
2.3. OPERATORHUB について 26
2.4. OPERATOR のクラスターへの追加 28
2.5. クラスターからの OPERATOR の削除 34
2.6. インストールされた OPERATOR からのアプリケーションの作成 37
2.7. カスタムリソース定義からのリソースの管理 39

第3章 アプリケーションライフサイクル管理 **43**
3.1. アプリケーションの作成 43

第4章 サービスブローカー **51**
4.1. サービスカタログのインストール 51
4.2. テンプレートサービスブローカーのインストール 53
4.3. テンプレートアプリケーションのプロビジョニング 55
4.4. テンプレートサービスブローカーのアンインストール 56
4.5. OPENSHIFT ANSIBLE BROKER のインストール 57
4.6. OPENSHIFT ANSIBLE BROKER の設定 62
4.7. サービスバンドルのプロビジョニング 65
4.8. OPENSHIFT ANSIBLE BROKER のアンインストール 66

第5章 DEPLOYMENT **68**
5.1. DEPLOYMENT および DEPLOYMENTCONFIG について 68
5.2. デプロイメントプロセスの管理 74
5.3. DEPLOYMENTCONFIG ストラテジーの使用 81
5.4. ルートベースのデプロイメントストラテジーの使用 88

第6章 CRD **96**
6.1. カスタムリソース定義による KUBERNETES API の拡張 96
6.2. カスタムリソース定義からのリソースの管理 101

第7章 クォータ **104**
7.1. プロジェクトごとのリソースクォータ 104
7.2. 複数のプロジェクト間のリソースクォータ 115

第8章 アプリケーションの正常性のモニタリング **119**
8.1. ヘルスチェックについて 119
8.2. ヘルスチェックの設定 121

第9章 アプリケーションのアイドリング **125**
9.1. アプリケーションのアイドリング 125
9.2. アプリケーションのアイドリング解除 125

第10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 **127**
10.1. プルーニングの基本操作 127
10.2. グループのプルーニング 127
10.3. デプロイメントのプルーニング 128
10.4. ビルドのプルーニング 128

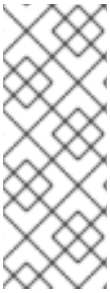
10.5. イメージのプルーニング 129
10.6. レジストリーのハードプルーニング 135
10.7. CRON ジョブのプルーニング 138

第11章 OPERATOR SDK 139
11.1. OPERATOR SDK の使用を開始する 139
11.2. ANSIBLE ベース OPERATOR の作成 152
11.3. HELM ベース OPERATOR の作成 172
11.4. CLUSTERSERVICEVERSION (CSV) の生成 182
11.5. PROMETHEUS による組み込みモニタリングの設定 192
11.6. リーダー選択の設定 194
11.7. OPERATOR SDK CLI リファレンス 196
11.8. 付録 204

第1章 プロジェクト

1.1. プロジェクトの使用

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

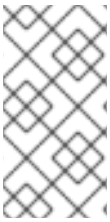


注記

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

1.1.1. Web コンソールを使用したプロジェクトの作成

クラスター管理者が許可する場合、新規プロジェクトを作成できます。



注記

openshift- および **kube-** で始まる名前のプロジェクトは OpenShift Container Platform によって重要 (Critical) と見なされます。そのため、OpenShift Container Platform では、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することはできません。

手順

1. **Home** → **Projects** に移動します。
2. **Create Project** をクリックします。
3. プロジェクトの詳細を入力します。
4. **Create** をクリックします。

1.1.2. CLI を使用したプロジェクトの作成

クラスター管理者が許可する場合、新規プロジェクトを作成できます。

注記

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

手順

1. 以下を実行します。

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

例:

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

注記

作成できるプロジェクトの数は、システム管理者によって制限される場合があります。上限に達すると、新規プロジェクトを作成できるように既存プロジェクトを削除しなければならない場合があります。

1.1.3. Web コンソールを使用したプロジェクトの表示

手順

1. **Home → Projects** に移動します。
2. 表示するプロジェクトを選択します。
   このページで、**Resources** ボタンをクリックしてプロジェクトのワークロードを確認し、**Dashboard** ボタンをクリックしてプロジェクトについてのメトリクスおよび詳細情報を確認します。

1.1.4. 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. プロジェクトの一覧を表示するには、以下を実行します。

   ```
   $ oc get projects
   ```

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

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

1.1.5. プロジェクトへの追加

手順

1. **Home → Projects** に移動します。
2. プロジェクトを選択します。
3. **Project Status** メニューの右上にある **Add** をクリックし、指定のオプションから選択します。

1.1.6. Web コンソールを使用したプロジェクトステータスの確認

手順

1. **Home → Projects** に移動します。
2. ステータスを確認するプロジェクトを選択します。

1.1.7. CLI を使用したプロジェクトステータスの確認

手順

1. 以下を実行します。

```
$ oc status
```

このコマンドは、コンポーネントとそれらの各種の関係を含む現在のプロジェクトの概要を示します。

1.1.8. Web コンソールを使用したプロジェクトの削除

手順

1. **Home → Projects** に移動します。
2. プロジェクトの一覧から削除するプロジェクトを見つけます。
3. プロジェクト一覧の右端にあるメニューから **Delete Project** を選択します。プロジェクトを削除するパーミッションがない場合は、**Delete Project** オプションがグレーアウトするため、オプションをクリックすることができません。

1.1.9. CLI を使用したプロジェクトの削除

プロジェクトを削除する際に、サーバーはプロジェクトのステータスを **Active** から **Terminating** に更新します。次に、サーバーは **Terminating** 状態のプロジェクトからすべてのコンテンツをクリアしてから、最終的にプロジェクトを削除します。プロジェクトのステータスが **Terminating** の場合、新規のコンテンツをプロジェクトに追加することはできません。プロジェクトは CLI または Web コンソールから削除できます。

手順

1. 以下を実行します。

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

1.2. 別のユーザーとしてのプロジェクトの作成

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

1.2.1. API の権限借用

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

1.2.2. プロジェクト作成時のユーザー権限の借用

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

手順

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

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

1.3. プロジェクト作成の設定

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

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

1.3.1. プロジェクト作成について

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

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

表1.1 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。

1.3.2. 新規プロジェクトのテンプレートの変更

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

独自のカスタムプロジェクトテンプレートを作成するには、以下を実行します。

手順

1. **cluster-admin** 権限を持つユーザーとしてのログイン。
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、または既存オブジェクトを変更することにより、テキストエディターで生成される **template.yaml** ファイルを変更します。
4. プロジェクトテンプレートは、**openshift-config** namespace に作成される必要があります。変更したテンプレートを読み込みます。

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

5. Web コンソールまたは CLI を使用し、プロジェクト設定リソースを編集します。
   - Web コンソールの使用
     i. **Administration → Cluster Settings** ページに移動します。
     ii. **Global 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。

1.3.3. プロジェクトのセルフプロビジョニングの無効化

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

手順

1. **cluster-admin** 権限を持つユーザーとしてのログイン。
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

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

self-provisionersセクションのサブジェクトを確認します。

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

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

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

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

4. ロールへの自動更新を防ぐには、**self-provisioners** クラスターロールバインディングを編集します。自動更新により、クラスターロールがデフォルトの状態にリセットされます。
   - CLI を使用してロールバインディングを更新するには、以下を実行します。
     i. 以下のコマンドを実行します。

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

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

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

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

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

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

```
$ oc new-project test

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

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

1.3.4. プロジェクト要求メッセージのカスタマイズ

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

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

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

- 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. **Global 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のログイン。

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>
```

例:

```
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2章 OPERATOR

2.1. OPERATOR について

概念的に、**Operator** は人間の運用上のナレッジを使用し、これをコンシューマーと簡単に共有できるソフトウェアにエンコードします。

Operator は、ソフトウェアの他の部分を実行する運用上の複雑さを軽減するソフトウェアの特定の部分で構成されます。Operator はソフトウェアベンダーのエンジニアリングチームの拡張機能のように動作し、(OpenShift Container Platform などの) Kubernetes 環境を監視し、その最新状態に基づいてリアルタイムの意思決定を行います。高度な Operator はアップグレードをシームレスに実行し、障害に自動的に対応するように設計されており、時間の節約のためにソフトウェアのバックアッププロセスを省略するなどのショートカットを実行することはありません。

技術的には、**Operator** は Kubernetes アプリケーションをパッケージ化し、デプロイし、管理する方法です。

Kubernetes アプリケーションは、Kubernetes にデプロイされ、Kubernetes API および **kubectl** または **oc** ツールを使用して管理されるアプリケーションです。Kubernetes を最大限に活用するには、Kubernetes 上で実行されるアプリケーションを提供し、管理するために拡張できるように一連の総合的な API が必要です。Operator は、Kubernetes 上でこのタイプのアプリケーションを管理するランタイムと見なすことができます。

2.1.1. Operator を使用する理由

Operator は以下を提供します。

- インストールおよびアップグレードの反復性。
- すべてのシステムコンポーネントの継続的なヘルスチェック。
- OpenShift コンポーネントおよび ISV コンテンツの OTA (Over-the-air) 更新。
- フィールドエンジニアからの知識をカプセル化し、1 または 2 ユーザーだけでなく、すべてのユーザーに展開する場所。

Kubernetes にデプロイする理由

Kubernetes (延長線上で考えると OpenShift Container Platform も含まれる) には、シークレットの処理、負荷分散、サービスの検出、自動スケーリングなどの、オンプレミスおよびクラウドプロバイダーで機能する、複雑な分散システムをビルドするために必要なすべてのプリミティブが含まれます。

アプリケーションを Kubernetes API および kubectl ツールで管理する理由

これらの API は機能的に充実しており、すべてのプラットフォームのクライアントを持ち、クラスターのアクセス制御/監査機能にプラグインします。Operator は Kubernetes の拡張メカニズム、カスタムリソース定義 (CRD、Custom Resource Definition) を使用するので、 **MongoDB** などのカスタムオブジェクトはビルトインされた、ネイティブ Kubernetes オブジェクトのように表示され、機能します。

Operator とサービスブローカーとの比較

サービスブローカーは、アプリケーションのプログラムによる検出およびデプロイメントを行うための 1 つの手段です。ただし、これは長期的に実行されるプロセスではないため、アップグレード、フェイルオーバー、またはスケーリングなどの Day 2 オペレーションを実行できません。カスタマイズおよびチューニング可能なパラメーターはインストール時に提供されるのに対し、Operator は

クラスターの最新の状態を常に監視します。クラスター外のサービスを使用する場合は、これらをサービスブローカーで使用できますが、Operator もこれらのクラスター外のサービスに使用できます。

2.1.2. Operator Framework

Operator Framework は、上記のカスタマーエクスペリエンスに関連して提供されるツールおよび機能のファミリーです。これは、コードを作成するためだけにあるのではなく、Operator のテスト、実行、および更新などの重要な機能を実行します。Operator Framework コンポーネントは、これらの課題に対応するためのオープンソースツールで構成されています。

Operator SDK

Kubernetes API の複雑性を把握していなくても、それぞれの専門知識に基づいて独自の Operator のブートストラップ、ビルド、テストおよびパッケージ化を実行できるよう Operator の作成者を支援します。

Operator Lifecycle Manager

クラスター内の Operator のインストール、アップグレード、ロールベースのアクセス制御 (RBAC) を制御します。OpenShift Container Platform 4.1 ではデフォルトでデプロイされます。

Operator Metering

クラスター上で Day 2 管理についての Operator の運用上のメトリクスを収集し、使用状況のメトリクスを集計します。

OperatorHub

クラスター上で Operator を検出し、インストールするための Web コンソールです。OpenShift Container Platform 4.1 ではデフォルトでデプロイされます。

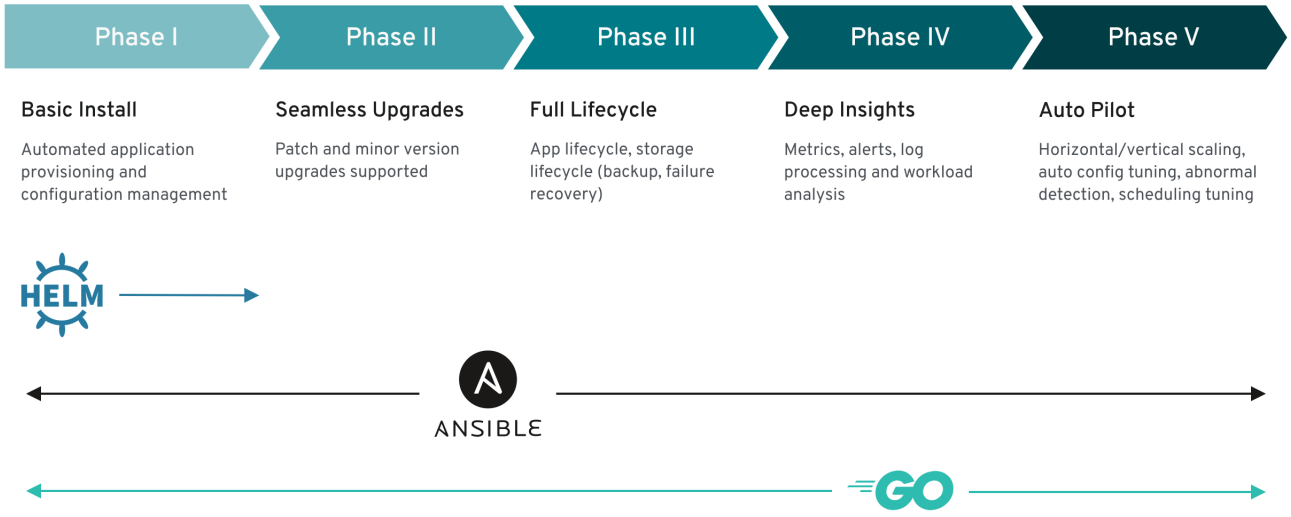
これらのツールは組み立て可能なツールとして設計されているため、役に立つと思われるツールを使用できます。

2.1.3. Operator 成熟度モデル

Operator 内にカプセル化されている管理ロジックの複雑さのレベルはさまざまです。また、このロジックは通常 Operator によって表されるサービスのタイプによって大きく変わります。

ただし、大半の Operator に含まれる特定の機能セットについては、Operator のカプセル化された操作の成熟度を一般化することができます。このため、以下の Operator 成熟度モデルは、 Operator の一般的な Day 2 オペレーションについての 5 つのフェーズの成熟度を定義しています。

図2.1 Operator 成熟度モデル



上記のモデルでは、これらの機能を Operator SDK の Helm、Go、および Ansible 機能で最適に開発する方法も示します。

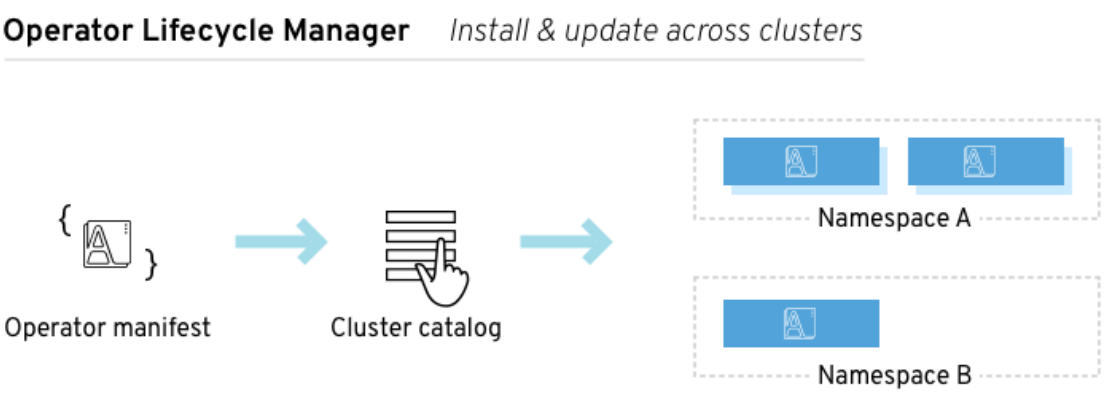
2.2. OPERATOR LIFECYCLE MANAGER について

以下では、OpenShift Container Platform における Operator Lifecycle Manager (OLM) のワークフローおよびアーキテクチャーの概要を説明します。

2.2.1. Operator Lifecycle Manager の概要

OpenShift Container Platform 4.1 では、 **Operator Lifecycle Manager** (OLM) を使用することにより、ユーザーはすべての Operator およびクラスター全体で実行される関連サービスをインストールし、更新し、管理することができます。これは、Kubernetes のネイティブアプリケーション (Operator) を効果的かつ自動化された拡張可能な方法で管理するために設計されたオープンソースツールキットの Operator Framework の一部です。

図2.2 Operator Lifecycle Manager ワークフロー



OLM は OpenShift Container Platform 4.1 でデフォルトで実行されます。OpenShift Container

Platform Web コンソールは、クラスター管理者が Operator をインストールしたり、クラスターで利用可能な Operator のカタログを使用できるように特定のプロジェクトアクセスを付与したりするのに使用する管理画面を提供します。

開発者の場合には、セルフサービスを使用することで、専門的な知識がなくてもデータベースのインスタンスのプロビジョニングや設定、またモニタリング、ビッグデータサービスなどを実行できます。Operator にそれらに関するナレッジが織り込まれているためです。

2.2.2. ClusterServiceVersion (CSV)

ClusterServiceVersion (CSV) は、Operator Lifecycle Manager (OLM) のクラスターでの Operator の実行を支援する Operator メタデータから作成される YAML マニフェストです。これは、ユーザーインターフェースにロゴ、説明、およびバージョンなどの情報を設定するために使用される Operator コンテナーイメージを伴うメタデータです。また、これは Operator が必要とする RBAC ルールやそれが管理したり、依存したりするカスタムリース(Custom Resource、CR) などの、Operator を実行するために必要な技術情報の情報源にもなります。

CSV は以下で構成されます。

メタデータ

- アプリケーションメタデータ:
  - 名前、説明、バージョン (semver 準拠)、リンク、ラベル、アイコンなど

インストールストラテジー

- タイプ: Deployment
  - サービスアカウントおよび必要なパーミッションのセット
  - Deployment のセット。

Custom Resource Definitions (CRDs)

- タイプ
- Owned: サービスで管理されます。
- Required: サービスが実行されるためにクラスターに存在する必要があります。
- Resources: Operator が対話するリソースの一覧です。
- Descriptors: 意味情報を提供するために CRD 仕様およびステータスフィールドにアノテーションを付けます。

2.2.3. Operator Lifecycle Manager アーキテクチャー

Operator Lifecycle Manager (OLM) は、OLM Operator およびカタログ Operator の 2 つの Operator で構成されています。

これらの Operator はそれぞれ OLM フレームワークのベースとなるカスタムリソース定義(CRD) を管理します。

表2.1 OLM およびカタログ Operator で管理される CRD

リソース	短縮名	所有する Operator	説明
ClusterServiceVersion	**csv**	OLM	アプリケーションのメタデータ: 名前、バージョン、アイコン、必須リソース、インストールなど。
InstallPlan	**ip**	カタログ	CSV を自動的にインストールするか、またはアップグレードするために作成されるリソースの計算された一覧。
CatalogSource	**catsrc**	カタログ	CSV、CRD、およびアプリケーションを定義するパッケージのリポジトリー。
Subscription	**sub**	カタログ	パッケージのチャネルを追跡して CSV を最新の状態に保ちます。
OperatorGroup	**org**	OLM	同じ namespace にデプロイされたすべての Operator を OperatorGroup オブジェクトとして設定し、namespace の一覧またはクラスター全体でカスタムリソース (CR) を監視します。

これらの Operator はそれぞれリソースも作成します。

表2.2 OLM およびカタログ Operator によって作成されるリソース

リソース	所有する Operator
Deployment	OLM
ServiceAccount	
(Cluster)Role	
(Cluster)RoleBinding	
Custom Resource Definition (CRD)	カタログ
ClusterServiceVersion (CSV)	

2.2.3.1. OLM Operator

OLM Operator は、CSV で指定された必須リソースがクラスター内にあることが確認された後に CSV リソースで定義されるアプリケーションをデプロイします。

OLM Operator は必須リソースの作成には関与せず、ユーザーが CLI を使用してこれらのリソースを手動で作成したり、カタログ Operator を使用してこれらのリソースを作成することを選択することができます。このタスクの分離により、アプリケーションに OLM フレームワークをどの程度活用するかに関連してユーザーによる追加機能の購入を可能にします。

OLM Operator はすべての namespace を監視するように設定されることが多い一方で、それらすべてが別々の namespace を管理する限り、他の OLM Operator と並行して操作することができます。

OLM Operator のワークフロー

- namespace で ClusterServiceVersion (CSV) の有無を確認し、要件を満たしていることを確認します。その場合、CSV のインストールストラテジーを実行します。

注記

CSV は、インストールストラテジーの実行を可能にするには、OperatorGroup のアクティブなメンバーである必要があります。

2.2.3.2. カタログ Operator

カタログ Operator は CSV およびそれらが指定する必須リソースを解決し、インストールします。また、 CatalogSource でチャネル内のパッケージへの更新の有無を確認し、それらを利用可能な最新バージョンに (オプションで自動的に) アップグレードします。

チャネル内のパッケージを追跡する必要のあるユーザーは、必要なパッケージ、チャネル、および更新のプルに使用する CatalogSource を設定する Subscription リソースを作成します。 更新が見つかると、ユーザーに代わって適切な InstallPlan の namespace への書き込みが行われます。

また、ユーザーは必要な CSV および承認ストラテジーの名前を含む InstallPlan リソースを直接作成でき、カタログ Operator はすべての必須リソースの作成の実行計画を作成します。これが承認されると、カタログ Operator はすべてのリソースを InstallPlan に作成します。 その後、これが単独で OLM Operator の要件を満たすと、CSV のインストールに移行します。

カタログ Operator のワークフロー

- 名前でインデックス化される CRD および CSV のキャッシュがあることを確認します。
- ユーザーによって作成された未解決の InstallPlan の有無を確認します。
  - 要求される名前に一致する CSV を検索し、これを解決済みリソースとして追加します。
  - 管理対象または必須の CRD のそれぞれについて、これを解決済みリソースとして追加します。
  - 必須 CRD のそれぞれについて、これを管理する CSV を検索します。
- 解決済みの InstallPlan の有無を確認し、それについての検出されたすべてのリソースを作成します (ユーザーによって、または自動的に承認される場合)。
- CatalogSource および Subscription の有無を確認し、それらに基づいて InstallPlan を作成します。

2.2.3.3. カタログレジストリー

カタログレジストリーは、クラスター内での作成用に CSV および CRD を保存し、パッケージおよびチャネルについてのメタデータを保存します。

パッケージマニフェスト は、パッケージアイデンティティーを CSV のセットに関連付けるカタログレジストリー内のエントリーです。パッケージ内で、チャネルは特定の CSV を参照します。CSV は置き換え対象の CSV を明示的に参照するため、パッケージマニフェストはカタログ Operator に対し、CSV

をチャネル内の最新バージョンに更新するために必要なすべての情報を提供します (各中間バージョンをステップスルー)。

2.2.4. OperatorGroup

OperatorGroup は、マルチテナント設定を OLM でインストールされた Operator に提供する OLM リソースです。OperatorGroup は、そのメンバー Operator に必要な RBAC アクセスを生成する際に使用するターゲット namespace のセットを選択します。ターゲット namespace のセットは、CSV の olm.targetNamespaces アノテーションに保存されるカンマ区切りの文字列によって指定されます。このアノテーションは、メンバー Operator の CSV インスタンスに適用され、それらのデプロインメントに展開されます。

2.2.4.1. OperatorGroup メンバーシップ

Operator は、以下の条件が true の場合に OperatorGroup の **メンバー** とみなされます。

- Operator の CSV が OperatorGroup と同じ namespace にある。
- Operator の CSV の InstallMode は OperatorGroup がターゲットに設定する namespace のセットをサポートする。

InstallMode は **InstallModeType** フィールドおよびブール値の **Supported** フィールドで構成される。CSV の仕様には、4 つの固有の **InstallModeTypes** の InstallMode のセットを含めることができます。

表2.3 InstallMode およびサポートされる OperatorGroup

InstallMode タイプ	説明
OwnNamespace	Operator は、独自の namespace を選択する OperatorGroup のメンバーにすることができます。
SingleNamespace	Operator は 1 つの namespace を選択する OperatorGroup のメンバーにすることができます。
MultiNamespace	Operator は複数の namespace を選択する OperatorGroup のメンバーにすることができます。
AllNamespaces	Operator はすべての namespace を選択する OperatorGroup のメンバーにすることができます (ターゲット namespace 設定は空の文字列 **""** です)。

注記

CSV の仕様が **InstallModeType** のエントリーを省略する場合、そのタイプは暗黙的にこれをサポートする既存エントリーによってサポートが示唆されない限り、サポートされないものとみなされます。

2.2.4.1.1. OperatorGroup メンバーシップのトラブルシューティング

- 複数の OperatorGroup が単一の namespace にある場合、その namespace で作成されるすべての CSV は**TooManyOperatorGroups** の理由で失敗状態に切り替わります。この理由で失敗状態になる CSV は、それらの namespace の OperatorGroup 数が 1 になると保留状態に切り替わります。

- CSV の InstallMode がその namespace で OperatorGroup のターゲット namespace 選択をサポートしない場合、CSV は **UnsupportedOperatorGroup** の理由で失敗状態に切り替わります。この理由で失敗した状態にある CSV は、 OperatorGroup のターゲット namespace の選択がサポートされる設定に変更されるか、または CSV の InstallMode が OperatorGroup の target namespace 選択をサポートするように変更される場合に保留状態に切り替わります。

2.2.4.2. ターゲット namespace の選択

spec.selector フィールドでラベルセレクターを使用して OperatorGroup の namespace のセットを指定します。

```
apiVersion: operators.coreos.com/v1
kind: OperatorGroup
metadata:
  name: my-group
  namespace: my-namespace
  spec:
    selector:
      matchLabels:
        cool.io/prod: "true"
```

さらに、**spec.targetNamespaces** フィールドを使用してターゲット namespace に名前を明示的に指定することもできます。

```
apiVersion: operators.coreos.com/v1
kind: OperatorGroup
metadata:
  name: my-group
  namespace: my-namespace
spec:
  targetNamespaces:
  - my-namespace
  - my-other-namespace
  - my-other-other-namespace
```

注記

spec.targetNamespaces と **spec.selector** の両方が定義されている場合、**spec.selector** は無視されます。

または、**spec.selector** と **spec.targetNamespaces** の両方を省略し、**global** OperatorGroup を指定できます。 これにより、すべての namespace が選択されます。

```
apiVersion: operators.coreos.com/v1
kind: OperatorGroup
metadata:
  name: my-group
  namespace: my-namespace
```

解決済みの一覧の選択された namespace は OperatorGroup の **status.namespaces** フィールドに表示されます。グローバル OperatorGroup の **status.namespace** には空の文字列 ("") が含まれます。 これは、消費する Operator に対し、すべての namespace を監視するように示唆します。

2.2.4.3. OperatorGroup CSV アノテーション

OperatorGroup のメンバー CSV には以下のアノテーションがあります。

アノテーション	説明
olm.operatorGroup=<group_name>	OperatorGroup の名前が含まれます。
olm.operatorGroupNamespace=<group_namespace>	OperatorGroup の namespace が含まれます。
olm.targetNamespaces=<target_namespaces>	OperatorGroup のターゲット namespace 選択を一覧表示するカンマ区切りの文字列が含まれます。

注記

olm.targetNamespaces 以外のすべてのアノテーションがコピーされた CSV と共に含まれます。**olm.targetNamespaces** アノテーションをコピーされた CSV で省略すると、テナント間のターゲット namespace の重複が回避されます。

2.2.4.4. 提供される API アノテーション

OperatorGroup によって提供される **GroupVersionKinds** (GVK) についての情報が **olm.providedAPIs** アノテーションに表示されます。アノテーションの値は、カンマで区切られた **<kind>.<version>.<group>** で構成される文字列です。OperatorGroup のすべてのアクティブメンバーの CSV によって提供される CRD および APIService の GVK が含まれます。

PackageManifest リソースを提供する単一のアクティブメンバー CSV を含む OperatorGroup の以下の例を確認してください。

```
apiVersion: operators.coreos.com/v1
kind: OperatorGroup
metadata:
  annotations:
    olm.providedAPIs: PackageManifest.v1alpha1.packages.apps.redhat.com
  name: olm-operators
  namespace: local
  ...
spec:
  selector: {}
  serviceAccount:
    metadata:
      creationTimestamp: null
  targetNamespaces:
  - local
status:
  lastUpdated: 2019-02-19T16:18:28Z
  namespaces:
  - local
```

2.2.4.5. ロールベースのアクセス制御

OperatorGroup の作成時に、3 つの ClusterRole が生成されます。それぞれには、以下の示すように ClusterRoleSelector がラベルに一致するように設定された単一の AggregationRule が含まれます。

ClusterRole	一致するラベル
<operatorgroup_name>-admin	**olm.opgroup.permissions/aggregate-to-admin: <operatorgroup_name>**
<operatorgroup_name>-edit	**olm.opgroup.permissions/aggregate-to-edit: <operatorgroup_name>**
<operatorgroup_name>-view	**olm.opgroup.permissions/aggregate-to-view: <operatorgroup_name>**

以下の RBAC リソースは、CSV が **AllNamespaces** InstallMode のあるすべての namespace を監視しており、理由が **InterOperatorGroupOwnerConflict** の失敗状態にない限り、CSV が OperatorGroup のアクティブメンバーになる際に生成されます。

- CRD からの各 API リソースの ClusterRole
- APIService からの各 API リソースの ClusterRole
- 追加のロールおよびロールバインディング

表2.4 CRD からの各 API リソース用に生成された ClusterRole

ClusterRole	設定
<kind>.<group>-<version>-admin	**<kind>** の動詞 • * 集計ラベル: • **rbac.authorization.k8s.io/aggregate-to-admin: true** • **olm.opgroup.permissions/aggregate-to-admin: <operatorgroup_name>**

ClusterRole	設定
<kind>.<group>-<version>-edit	**<kind>** の動詞 • **create** • **update** • **patch** • **delete** 集計ラベル: • **rbac.authorization.k8s.io/aggregate-to-edit: true** • **olm.opgroup.permissions/aggregate-to-edit: <operatorgroup_name>**
<kind>.<group>-<version>-view	**<kind>** の動詞 • **get** • **list** • **watch** 集計ラベル: • **rbac.authorization.k8s.io/aggregate-to-view: true** • **olm.opgroup.permissions/aggregate-to-view: <operatorgroup_name>**
<kind>.<group>-<version>-view-crdview	Verbs on **apiextensions.k8s.io customresourcedefinitions <crd-name>**: • **get** 集計ラベル: • **rbac.authorization.k8s.io/aggregate-to-view: true** • **olm.opgroup.permissions/aggregate-to-view: <operatorgroup_name>**

表2.5 APIService からの各 API リソース用に生成された ClusterRole

ClusterRole	設定

ClusterRole	設定
<kind>.<group>-<version>-admin	**<kind>** の動詞 • ***** 集計ラベル: • **rbac.authorization.k8s.io/aggregate-to-admin: true** • **olm.opgroup.permissions/aggregate-to-admin: <operatorgroup_name>**
<kind>.<group>-<version>-edit	**<kind>** の動詞 • **create** • **update** • **patch** • **delete** 集計ラベル: • **rbac.authorization.k8s.io/aggregate-to-edit: true** • **olm.opgroup.permissions/aggregate-to-edit: <operatorgroup_name>**
<kind>.<group>-<version>-view	**<kind>** の動詞 • **get** • **list** • **watch** 集計ラベル: • **rbac.authorization.k8s.io/aggregate-to-view: true** • **olm.opgroup.permissions/aggregate-to-view: <operatorgroup_name>**

追加のロールおよびロールバインディング

- CSV が * が含まれる 1 つのターゲット namespace を定義する場合、ClusterRole と対応する ClusterRoleBinding が CSV のパーミッションフィールドに定義されるパーミッションごとに生成されます。生成されたすべてのリソースには **olm.owner: <csv_name>** および **olm.owner.namespace: <csv_namespace>** ラベルが付与されます。
- CSV が * が含まれる 1 つのターゲット namespace を定義 **しない** 場合、**olm.owner:**

<csv_name> および **olm.owner.namespace: <csv_namespace>** ラベルの付いた Operator namespace にあるすべてのロールおよびロールバインディングがターゲット namespace にコピーされます。

2.2.4.6. コピーされる CSV

OLM は、それぞれの OperatorGroup のターゲット namespace に、OperatorGroup のすべてのアクティブな CSV のコピーを作成します。コピーされる CSV の目的は、ユーザーに対して、特定の Operator が作成されるリソースを監視するように設定されたターゲット namespace について通知することにあります。コピーされる CSV にはステータスの理由 **Copied** があり、それらのソース CSV のステータスに一致するように更新されます。**olm.targetNamespaces** アノテーションは、クラスター上でコピーされる CSV が作成される前に取られます。ターゲット namespace 選択を省略すると、テナント間のターゲット namespace の重複が回避されます。コピーされる CSV はそれらのソース CSV が存在しなくなるか、またはそれらのソース CSV が属する OperatorGroup がコピーされた CSV の namespace をターゲットに設定しなくなると削除されます。

2.2.4.7. 静的 OperatorGroup

OperatorGroup はその **spec.staticProvidedAPIs** フィールドが **true** に設定されると **静的** になります。その結果、OLM は OperatorGroup の **olm.providedAPIs** アノテーションを変更しません。つまり、これを事前に設定することができます。これは、ユーザーが OperatorGroup を使用して namespace のセットでリソースの競合を防ぐ必要がある場合で、それらのリソースの API を提供するアクティブなメンバーの CSV がない場合に役立ちます。

以下は、**something.cool.io/cluster-monitoring: "true"** アノテーションのある、すべての namespace の Prometheus リソースを保護する OperatorGroup の例です。

```
apiVersion: operators.coreos.com/v1
kind: OperatorGroup
metadata:
  name: cluster-monitoring
  namespace: cluster-monitoring
  annotations:
    olm.providedAPIs:
Alertmanager.v1.monitoring.coreos.com,Prometheus.v1.monitoring.coreos.com,PrometheusRule.v1.mo
nitoring.coreos.com,ServiceMonitor.v1.monitoring.coreos.com
spec:
  staticProvidedAPIs: true
  selector:
    matchLabels:
      something.cool.io/cluster-monitoring: "true"
```

2.2.4.8. OperatorGroup の交差部分

2 つの OperatorGroup は、それらのターゲット namespace セットの交差部分が空のセットではなく、**olm.providedAPIs** アノテーションで定義されるそれらの指定 API セットの交差部分が空のセットではない場合に、 **交差部分のある指定 API** があると見なされます。

これによって生じ得る問題として、交差部分のある指定 API を持つ複数の OperatorGroup は、一連の交差部分のある namespace で同じリソースに関して競合関係になる可能性があります。

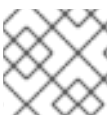
注記

交差ルールを確認すると、OperatorGroup の namespace は常に選択されたターゲット namespace の一部として組み込まれます。

2.2.4.8.1. 交差のルール

アクティブメンバーの CSV が同期する際はいつでも、OLM はクラスターで、CSV の OperatorGroup とそれ以外のすべての OperatorGroup 間に交差部分のある指定 API のセットについてクエリーします。その後、OLM はそのセットが空のセットであるかどうかを確認します。

- **true** であり、CSV の指定 API が OperatorGroup のサブセットである場合:
  - 移行を継続します。
- **true** であり、CSV の指定 API が Operator Group のサブセット **ではない** 場合:
  - OperatorGroup が静的である場合:
    - CSV に属するすべてのデプロイメントをクリーンアップします。
    - ステータスの理由 **CannotModifyStaticOperatorGroupProvidedAPIs** のある失敗状態に CSV を移行します。
  - OperatorGroup が静的 **ではない** 場合:
    - OperatorGroup の **olm.providedAPIs** アノテーションを、それ自体と CSV の指定 API の集合に置き換えます。
- **false** であり、CSV の指定 API が OperatorGroupt のサブセット **ではない** 場合:
  - CSV に属するすべてのデプロイメントをクリーンアップします。
  - ステータスの理由 **InterOperatorGroupOwnerConflict** のある失敗状態に CSV を移行します。
- **false** であり、CSV の提供された API が OperatorGroup のサブセットである場合:
  - OperatorGroup が静的である場合:
    - CSV に属するすべてのデプロイメントをクリーンアップします。
    - ステータスの理由 **CannotModifyStaticOperatorGroupProvidedAPIs** のある失敗状態に CSV を移行します。
  - OperatorGroup が静的 **ではない** 場合:
    - OperatorGroup の **olm.providedAPIs** アノテーションを、それ自体と CSV の指定 API 間の差異部分に置き換えます。



注記

OperatorGroup によって生じる失敗状態は非終了状態です。

以下のアクションは、OperatorGroup が同期するたびに実行されます。

- アクティブメンバーの CSV の指定 API のセットは、クラスターから計算されます。コピーされた CSV は無視されることに注意してください。

- クラスターセットは **olm.providedAPIs** と比較され、**olm.providedAPIs** に追加の API が含まれる場合は、それらの API がプルーニングされます。
- すべての namespace で同じ API を提供するすべての CSV は再びキューに入れられます。これにより、交差部分のあるグループ間の競合する CSV に対して、それらの競合が競合する CSV のサイズ変更または削除のいずれかによって解決されている可能性があることが通知されます。

2.2.5. メトリクス

OLM は、Prometheus ベースの OpenShift Container Platform クラスターモニタリングスタックで使用される特定の OLM 固有のリソースを公開します。

表2.6 OLM によって公開されるメトリクス

名前	説明
csv_count	正常に登録された CSV の数。
install_plan_count	InstallPlan の数。
subscription_count	サブスクリプションの数。
csv_upgrade_count	CatalogSource の単調 (monotonic) カウント。

2.3. OPERATORHUB について

以下では、OperatorHub のアーキテクチャーについて説明します。

2.3.1. OperatorHub の概要

OperatorHub は OpenShift Container Platform Web コンソールで利用でき、クラスター管理者が Operator を検出し、インストールするために使用するインターフェースです。1 回のクリックで、Operator はクラスター外のソースからプルでき、クラスター上でインストールされ、サブスクライブされ、エンジニアリングチームが Operator Lifecycle Manager (OLM) を使用してデプロイメント環境で製品をセルフサービスで管理される状態にすることができます。

クラスター管理者は、以下のカテゴリーにグループ化された OperatorSource から選択することができます。

カテゴリー	説明
Red Hat Operator	Red Hat によってパッケージ化され、出荷される Red Hat 製品。Red Hat によってサポートされます。
認定 Operator	大手独立系ソフトウェアベンダー (ISV) の製品。Red Hat は ISV とのパートナーシップにより、パッケージ化および出荷を行います。ISV によってサポートされます。

カテゴリー	説明
コミュニティー Operator	operator-framework/community-operators GitHub リポジトリーで関連するエンティティーによってメンテナンスされる、オプションで表示可能になるソフトウェア。正式なサポートはありません。
カスタム Operator	各自でクラスターに追加する Operator。カスタム Operator を追加しない場合、カスタムカテゴリーは Web コンソールの OperatorHub 上に表示されません。

OperatorHub コンポーネントは、デフォルトで OpenShift Container Platform の **openshift-marketplace** namespace に Operator としてインストールされ、実行されます。

2.3.2. OperatorHub アーキテクチャー

OperatorHub コンポーネントの Operator は、OperatorSource および CatalogSourceConfig の 2 つのカスタムリソース定義 (CRD) を管理します。

注記

一部の OperatorSource および CatalogSourceConfig 情報は OperatorHub ユーザーインターフェースで公開されますが、それらのファイルは独自の Operator を作成するユーザーによってのみ直接使用されます。

2.3.2.1. OperatorSource

それぞれの Operator について、OperatorSource は Operator バンドルを保存するために使用される外部データストアを定義するために使用されます。単純な OperatorSource には以下が含まれます。

フィールド	Description
type	データストアをアプリケーションレジストリーとして識別するために、**type** は **appregistry** に設定されます。
endpoint	現時点で、Quay は OperatorHub によって使用される外部データストアであるため、エンドポイントは Quay.io **appregistry** について **https://quay.io/cnr** に設定されます。
registryNamespace	コミュニティー Operator の場合、これは **community-operator** に設定されます。
displayName	オプションとして、Operator の OperatorHub ユーザーインターフェースに表示される名前に設定されます。
publisher	オプションとして、Operator をパブリッシュする個人または組織に設定され、これが OperatorHub に表示されるようにします。

2.3.2.2. CatalogSourceConfig

Operator の CatalogSourceConfig は、クラスター上の OperatorSource にある Operator を有効にするために使用されます。

単純な CatalogSourceConfig は以下を特定します。

フィールド	説明
targetNamespace	**openshift-operators** などの Operator がデプロイされる場所です。これは OLM が監視する namespace です。
packages	Operator の内容を構成するパッケージのカンマ区切りの一覧です。

2.4. OPERATOR のクラスターへの追加

以下では、クラスター管理者を対象に、Operator の OpenShift Container Platform クラスターへのインストールについて説明します。

2.4.1. OperatorHub からの Operator のインストール

クラスター管理者は、OpenShift Container Platform Web コンソールまたは CLI を使用して OperatorHub から Operator をインストールできます。その後、Operator を 1 つまたは複数の namespace にサブスクライブし、クラスター上で開発者が使用できるようにできます。

インストール時に、Operator の以下の初期設定を判別する必要があります。

インストールモード

All namespaces on the cluster (default)を選択して Operator をすべての namespace にインストールするか、または (利用可能な場合は) 個別の namespace を選択し、選択された namespace のみに Operator をインストールします。この例では、**All namespaces...** を選択し、Operator がすべてのユーザーおよびプロジェクトで利用可能な状態にします。

更新チャネル

Operator が複数のチャネルで利用可能な場合、サブスクライブするチャネルを選択できます。たとえば、(利用可能な場合に) **stable** チャネルからデプロイするには、これを一覧から選択します。

承認ストラテジー

自動 (Automatic) または手動 (Manual) のいずれかの更新を選択します。インストールされた Operator について自動更新を選択する場合、Operator の新規バージョンが利用可能になると、Operator Lifecycle Manager (OLM) は人の介入なしに、Operator の実行中のインスタンスを自動的にアップグレードします。手動更新を選択する場合、Operator の新規バージョンが利用可能になると、OLM は更新要求を作成します。クラスター管理者は、Operator が新規バージョンに更新されるように更新要求を手動で承認する必要があります。

2.4.1.1. Web コンソールを使用した OperatorHub からのインストール

この手順では、Couchbase Operator をサンプルとして使用し、OpenShift Container Platform Web コンソールを使用して、OperatorHub から Operator をインストールし、これにサブスクライブします。

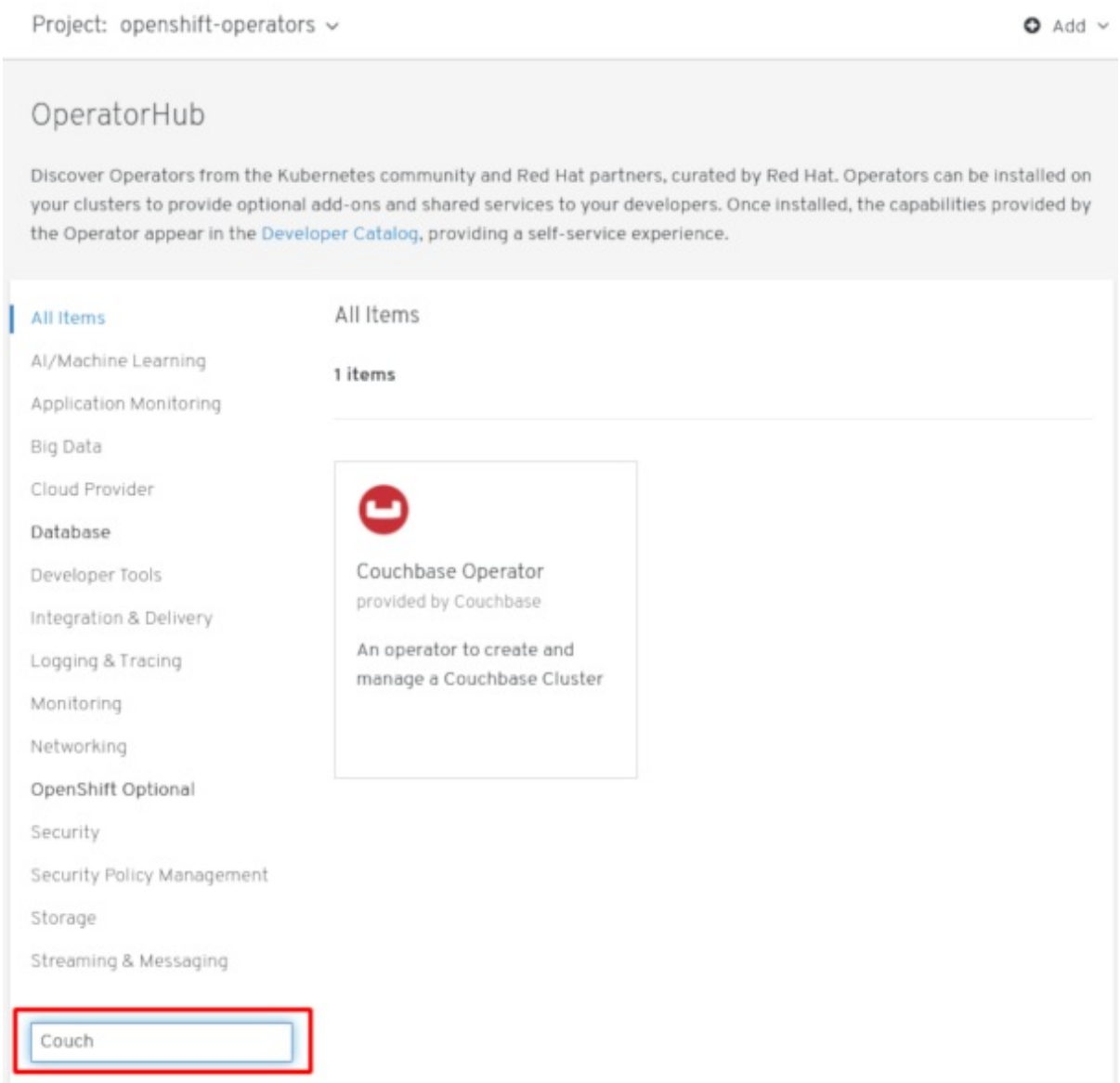
前提条件

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

手順

1. Web コンソールで、**Catalog → OperatorHub** ページに移動します。
2. スクロールするか、またはキーワードを **Filter by keyword** ボックスに入力し (この場合は **Couchbase**)、必要な Operator を見つけます。

   図2.3 キーワードによる Operator のフィルター



3. Operator を選択します。コミュニティー Operator の場合、Red Hat がそれらの Operator を認定していないことについての警告が出されます。作業を継続する前に、この警告を確認してください。Operator についての情報が表示されます。
4. Operator についての情報を確認してから、**Install** をクリックします。
5. **Create Operator Subscription** ページで以下を実行します。
   a. 以下のいずれかを選択します。
      - **All namespaces on the cluster (default)**は、デフォルトの **openshift-operators** namespace で Operator をインストールし、クラスターのすべての namespace を監視し、Operator をこれらの namespace に対して利用可能にします。このオプションは常に選択可能です。
      - **A specific namespace on the cluster**では、Operator をインストールする特定の単

- **A specific namespace on the cluster** では、Operator をインストールする特定の単一 namespace を選択できます。Operator は監視のみを実行し、この単一 namespace で使用されるように利用可能になります。

b. **Update Channel** を選択します (複数を選択できる場合)。

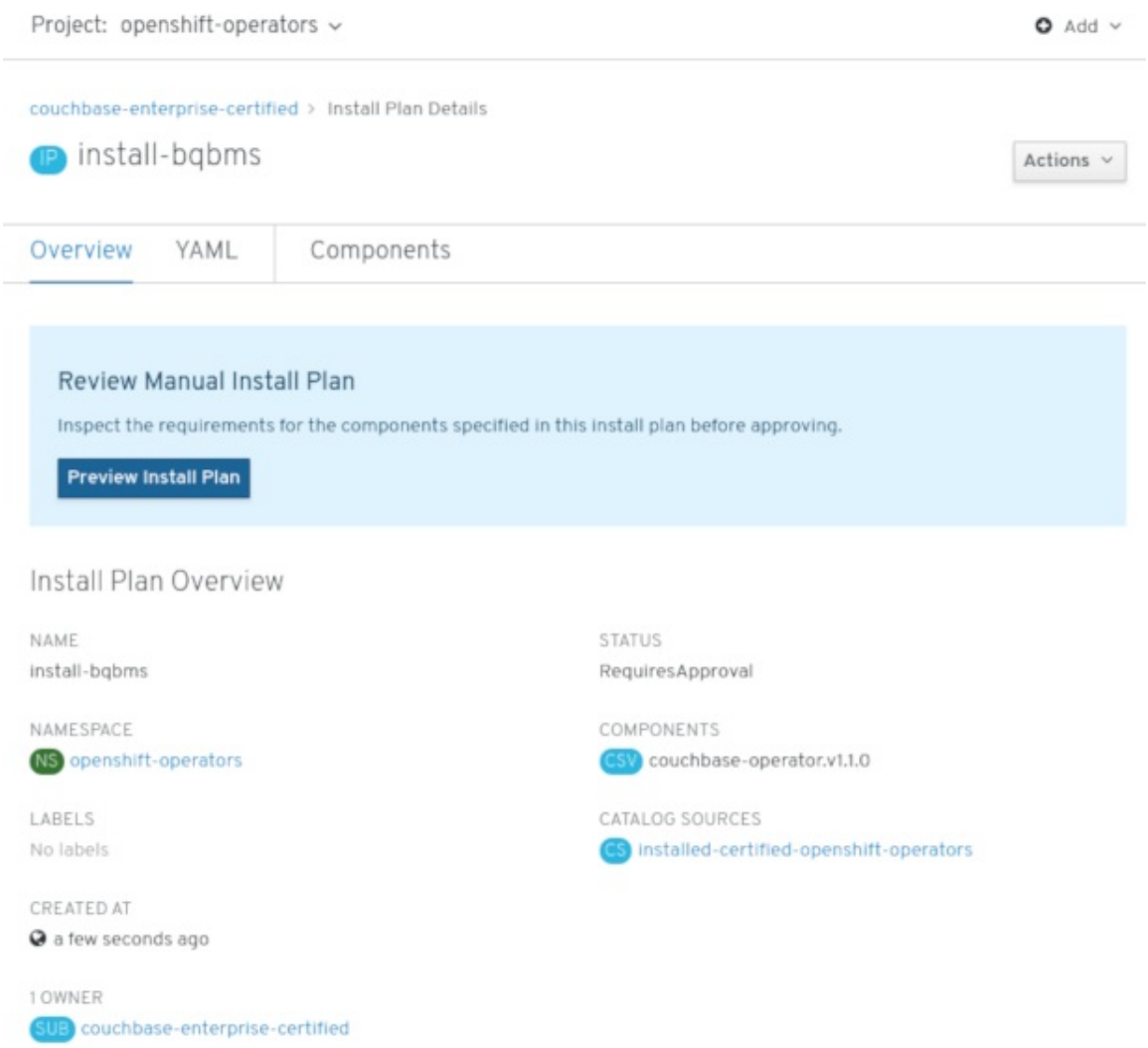
c. 前述のように、**自動 (Automatic)** または **手動 (Manual)** の承認ストラテジーを選択します。

6. **Subscribe** をクリックし、Operator をこの OpenShift Container Platform クラスターの選択した namespace で利用可能にします。

7. **Catalog → Operator Management** ページから、Operator サブスクリプションのインストールおよびアップグレードの進捗をモニターできます。

a. 手動の承認ストラテジーを選択している場合、Subscription のアップグレードステータスは、その Install Plan を確認し、承認するまで **Upgrading** のままになります。

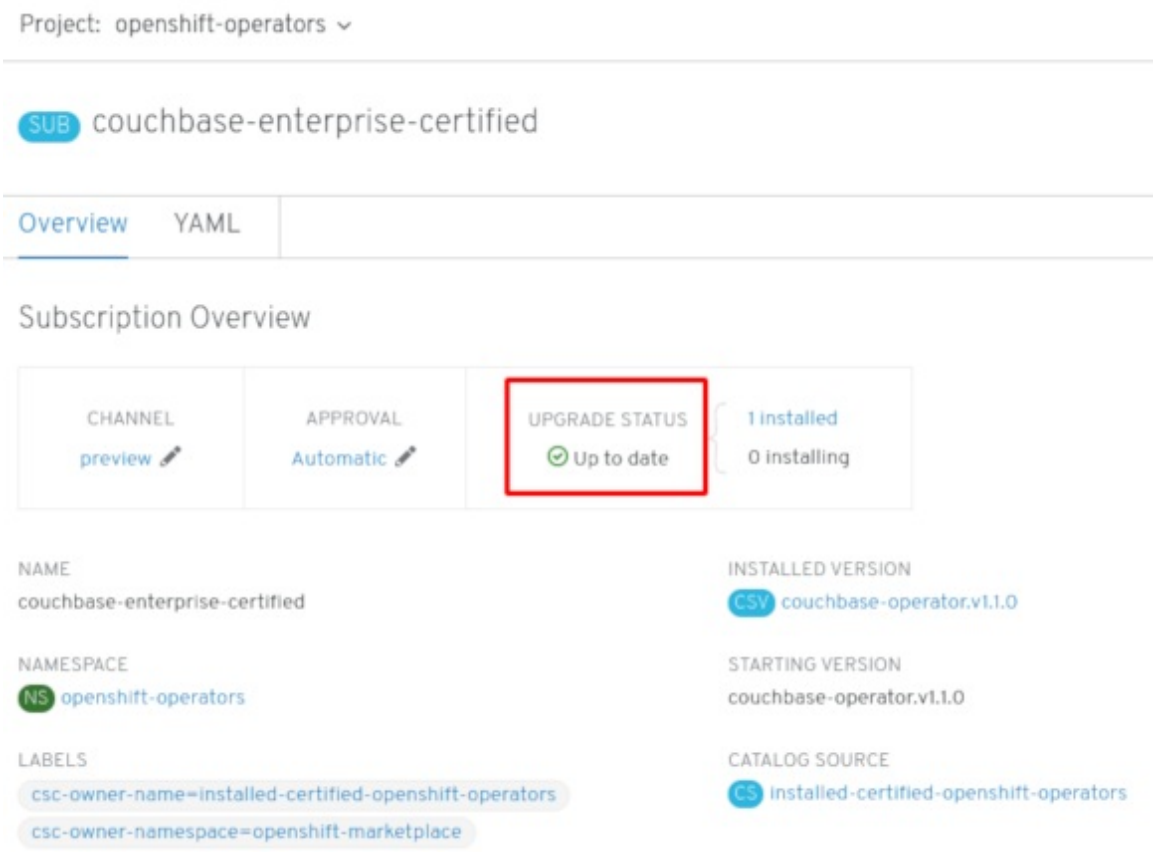
図2.4 Install Plan ページからの手動による承認



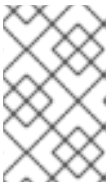
Install Plan ページでの承認後に、Subscription のアップグレードステータスは **Up to date** に移行します。

b. 自動承認ストラテジーを選択している場合、アップグレードステータスは、介入なしに **Up to date** に解決するはずです。

図2.5 Subscription のアップグレードステータス「Up to date」



8. サブスクリプションのアップグレードステータスが **Up to date** になった後に、**Catalog → Installed Operators** を選択して **Couchbase** ClusterServiceVersion (CSV) が表示され、その **ステータス** が最終的に関連する namespace で **InstallSucceeded** に解決することを確認します。



注記

All namespaces... インストールモードの場合、ステータスは **openshift-operators** namespace で **InstallSucceeded** になりますが、他の namespace でチェックする場合、ステータスは **Copied** になります。

上記通りにならない場合:

a. **Catalog → Operator Management** ページに切り替え、**Operator Subscriptions** および **Install Plans** タブで、 **Status** の下に失敗またはエラーがあるかどうかを確認します。

b. Pod のログを **openshift-operators** プロジェクトで確認します (または、**A specific namespace...** インストールモードが選択されている場合は別の関連する namespace) で確認します。これは、さらにトラブルシューティングする問題を報告する **Workloads → Pods** ページから実行します。

2.4.1.2. CLI を使用した OperatorHub からのインストール

OpenShift Container Platform Web コンソールを使用する代わりに、CLI を使用して OperatorHub から Operator をインストールできます。**oc** コマンドを使用して CatalogSourceConfig オブジェクトを作成または更新してから、Subscription オブジェクトを追加します。

注記

この手順の Web コンソールバージョンは、 CatalogSourceConfig および Subscription オブジェクトの作成を、1 つのステップで実行されるかのように背後で処理します。

前提条件

- **cluster-admin** パーミッションを持つアカウントを使用して OpenShift Container Platform クラスターにアクセスできること。
- **oc** コマンドをローカルシステムにインストールする。

手順

1. OperatorHub からクラスターで利用できる Operator の一覧を表示します。

```
$ oc get packagemanifests -n openshift-marketplace
NAME                  AGE
amq-streams           14h
packageserver         15h
couchbase-enterprise    14h
mongodb-enterprise      14h
etcd                  14h
myoperator            14h
...
```

2. クラスターを有効にする Operator を特定するには、 CatalogSourceConfig オブジェクト YAML ファイル (**csc.cr.yaml** など) を作成します。直前の手順 (couchbase-enterprise または etcd など) に記載されている 1 つ以上のパッケージを含めます。以下は例になります。

CatalogSourceConfig のサンプル

```
apiVersion: operators.coreos.com/v1
kind: CatalogSourceConfig
metadata:
  name: example
  namespace: openshift-marketplace
spec:
  targetNamespace: openshift-operators (1)
  packages: myoperator (2)
```

(1) Operator を利用可能にする必要のある namespace を特定するために **targetNamespace** を設定します。**openshift-operators** namespace は Operator Lifecycle Manager (OLM) によって監視されます。

(2) **packages** を、サブスクライブする必要のある Operator のカンマ区切りの一覧に設定します。

Operator は、**targetNamespace** に指定される namespace で CatalogSourceConfig から CatalogSource を生成します。

3. CatalogSourceConfig を作成し、選択された namespace で指定した Operator を有効にします。

```
$ oc apply -f csc.cr.yaml
```

4. namesapace を Operator にサブスクライブするためにサブスクリプションオブジェクト YAML ファイル (**myoperator-sub.yaml** など) を作成します。選択する namesapce には installMode (AllNamespaces または SingleNamespace モード) に一致する OperatorGroup がなければならないことに注意してください。

   Subscription の例

```
apiVersion: operators.coreos.com/v1alpha1
kind: Subscription
metadata:
  name: myoperator
  namespace: openshift-operators
spec:
  channel: alpha
  name: myoperator (1)
  source: example (2)
  sourceNamespace: openshift-operators
```

   (1) サブスクライブする Operator の名前。

   (2) 作成された CatalogSource の名前。

5. Subscription オブジェクトを作成します。

```
$ oc apply -f myoperator-sub.yaml
```

   この時点で、OLM は選択した Operator を認識します。Operator の ClusterServiceVersion (CSV) はターゲット namespace に表示され、Operator で指定される API は作成用に利用可能になります。

6. 後に追加の Operator をインストールする必要がある場合、以下を実行します。

   a. CatalogSourceConfig ファイル (この例では **csc.cr.yaml**) を追加のパッケージで更新します。以下は例になります。

      更新された CatalogSourceConfig のサンプル

```
apiVersion: operators.coreos.com/v1
kind: CatalogSourceConfig
metadata:
  name: example
  namespace: openshift-marketplace
spec:
  targetNamespace: global
  packages: myoperator,another-operator (1)
```

      (1) 既存のパッケージリストに新規パッケージを追加します。

   b. CatalogSourceConfig オブジェクトを更新します。

```
$ oc apply -f csc.cr.yaml
```

c. 新規 Operator 用に追加の Subscription オブジェクトを作成します。

追加リソース

- OperatorHub を使用してカスタム Operator をクラスターにインストールするには、まず Operator アーティファクトを Quay.io にアップロードし、次に独自の **OperatorSource** をクラスターに追加します。オプションで、シークレットを Operator に追加して認証を指定できます。その後、他の Operator と同様に Operator をクラスターで管理できます。これらの手順については、「Testing Operators」を参照してください。

2.5. クラスターからの OPERATOR の削除

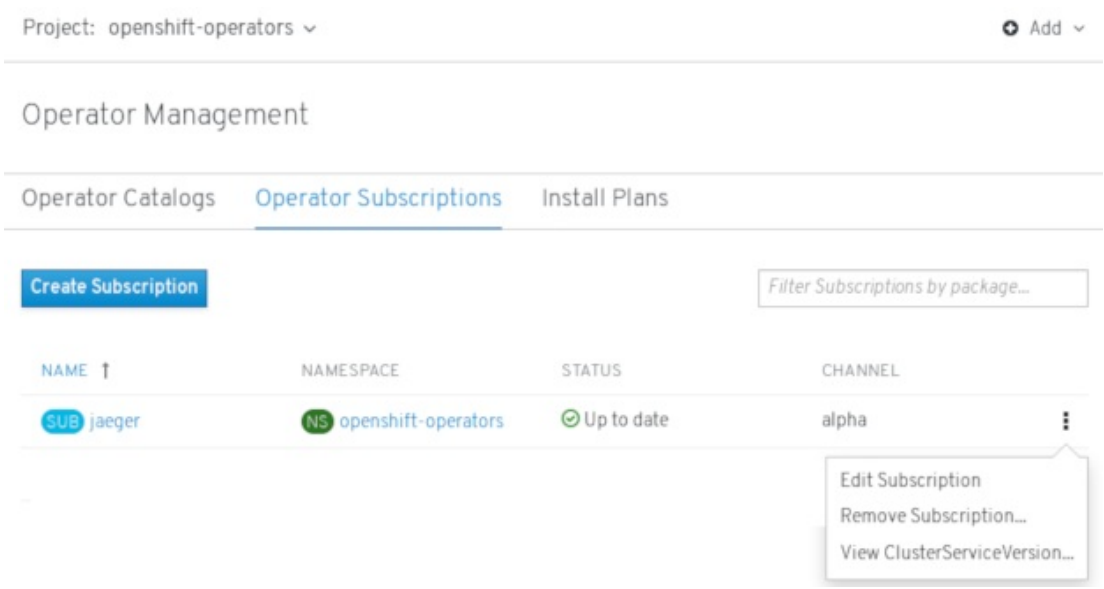
クラスターから Operator を削除 (アンインストール) するには、サブスクライブされた namespace からこれを削除するためにサブスクリプションを削除できます。クリーンな状態にする必要がある場合は、Operator CSV およびデプロイメントを削除してから CatalogSourceConfig で Operator のエントリーを削除することもできます。以下では、Web コンソールまたはコマンドラインのいずれかを使用してクラスターから Operator を削除する方法について説明しています。

2.5.1. Web コンソールの使用によるクラスターからの Operator の削除

Web コンソールでインストールされた Operator を選択された namespace から削除するには、以下の手順を実行します。

手順

1. 削除する Operator を選択します。これを実行するために使用できる 2 つのパスがあります。
   - **Catalog → OperatorHub** ページの使用:
     1. スクロールするか、またはキーワードを **Filter by keyword box** に入力し (この場合は **jaeger**)、必要な Operator を見つけてこれをクリックします。
     2. **Uninstall** をクリックします。
   - **Catalog → Operator Management** ページから:
     1. **Project** 一覧から Operator がインストールされている namespace を選択します。クラスター全体の Operator の場合、デフォルトは **openshift-operators** です。
     2. **Operator Subscriptions** タブから、削除する必要のある Operator を見つけ (この例では **jaeger**)、 Options メニュー 



3. **Remove Subscription** をクリックします。

2. **Remove Subscription** 画面でプロンプトが出されたら、インストールに関連するすべてのコンポーネントを削除する必要がある場合には、**Also completely remove the jaeger Operator from the selected namespace** チェックボックスをオプションで選択します。これにより CSV が削除され、次に Operator に関連付けられた Pod、Deployment、CRD および CR が削除されます。

3. **Remove** を選択します。この Operator は実行を停止し、更新を受信しなくなります。

注記

Operator がインストールされなくなるか、更新を受信しなくなっても、その Operator は依然として Operator Catalogs 一覧に表示され、再度サブスクライブできる状態になります。Operator をこの一覧から削除するには、コマンドラインで CatalogSourceConfig の Operator のエントリーを削除できます (「CLI の使用によるクラスターからの Operator の削除」の最後の手順を参照)。

2.5.2. CLI の使用によるクラスターからの Operator の削除

OpenShift Container Platform Web コンソールを使用する代わりに、CLI を使用して Operator をクラスターから削除できます。これは、Subscription および ClusterServiceVersion を **targetNamespace** から削除し、 CatalogSourceConfig を編集して Operator のパッケージ名を削除することで実行できます。

前提条件

- **cluster-admin** パーミッションを持つアカウントを使用して OpenShift Container Platform クラスターにアクセスできること。
- **oc** コマンドをローカルシステムにインストールすること。

手順

この例では、2 つの Operator (Jaeger および Descheduler) が **openshift-operators** namespace にインストールされています。ここでは、Descheduler を削除させずに Jaeger を削除することを目標とします。

1. サブスクライブされた Operator (例: **jaeger**) の現行バージョンを **currentCSV** フィールドで確認します。

```
$ oc get subscription jaeger -n openshift-operators -o yaml | grep currentCSV
  currentCSV: jaeger-operator.v1.8.2
```

2. Operator の Subscription (例: **jaeger**) を削除します。

```
$ oc delete subscription jaeger -n openshift-operators
subscription.operators.coreos.com "jaeger" deleted
```

3. 直前の手順で **currentCSV** 値を使用し、ターゲット namespace の Operator の CSV を削除します。

```
$ oc delete clusterserviceversion jaeger-operator.v1.8.2 -n openshift-operators
clusterserviceversion.operators.coreos.com "jaeger-operator.v1.8.2" deleted
```

4. **CatalogSourceConfig** リソースの内容を表示し、**spec** セクションでパッケージの一覧を確認します。

```
$ oc get catalogsourceconfig -n openshift-marketplace \
    installed-community-openshift-operators -o yaml
```

たとえば、spec セクションは以下のように表示されるかもしれません。

CatalogSourceConfig のサンプル

```
spec:
  csDisplayName: Community Operators
  csPublisher: Community
  packages: jaeger,descheduler
  targetNamespace: openshift-operators
```

5. 以下の 2 つの方法のいずれかで Operator を CatalogSourceConfig から削除します。

- 複数の Operator がある場合、CatalogSourceConfig リソースを編集し、Operator のパッケージを削除します。

```
$ oc edit catalogsourceconfig -n openshift-marketplace \
    installed-community-openshift-operators
```

以下のようにパッケージを **packages** 行から削除します。

CatalogSourceConfig で変更されたパッケージの例

```
  packages: descheduler
```

変更を保存すると、**marketplace-operator** が CatalogSourceConfig を調整します。

- CatalogSourceConfig に 1 つの Operator のみがある場合、以下のように CatalogSourceConfig 全体を削除してこれを削除することができます。

```
$ oc delete catalogsourceconfig -n openshift-marketplace \
    installed-community-openshift-operators
```

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

以下では、開発者を対象に、OpenShift Container Platform 4.1 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。

2.6.1. Operator を使用した etcd クラスターの作成

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

前提条件

- OpenShift Container Platform 4.1 クラスターへのアクセス
- 管理者によってクラスターにすでにインストールされている etcd Operator

手順

1. この手順を実行するために OpenShift Container Platform Web コンソールで新規プロジェクトを作成します。この例では、**my-etcd** というプロジェクトを使用します。
2. **Catalogs → Installed Operators** ページに移動します。クラスター管理者によってクラスターにインストールされ、使用可能にされた Operator が ClusterServiceVersion (CSV) の一覧としてここに表示されます。CSV は Operator によって提供されるソフトウェアを起動し、管理するために使用されます。

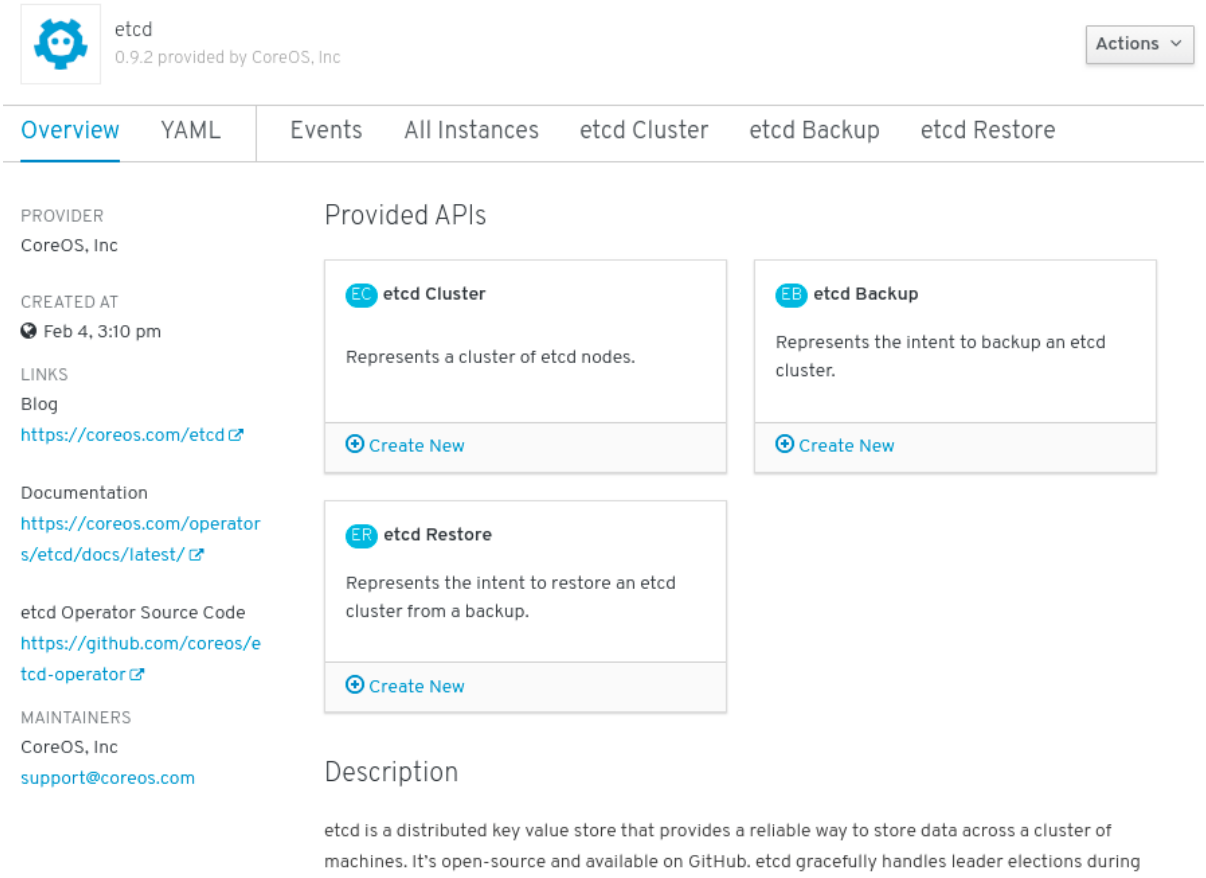
   ヒント

   以下を使用して、CLI でこの一覧を取得できます。

   ```
   $ oc get csv
   ```

3. **Installed Operators** ページで、**Copied** をクリックしてから、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。

図2.6 etcd Operator の概要



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、 **Deployments** または **ReplicaSets** 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。

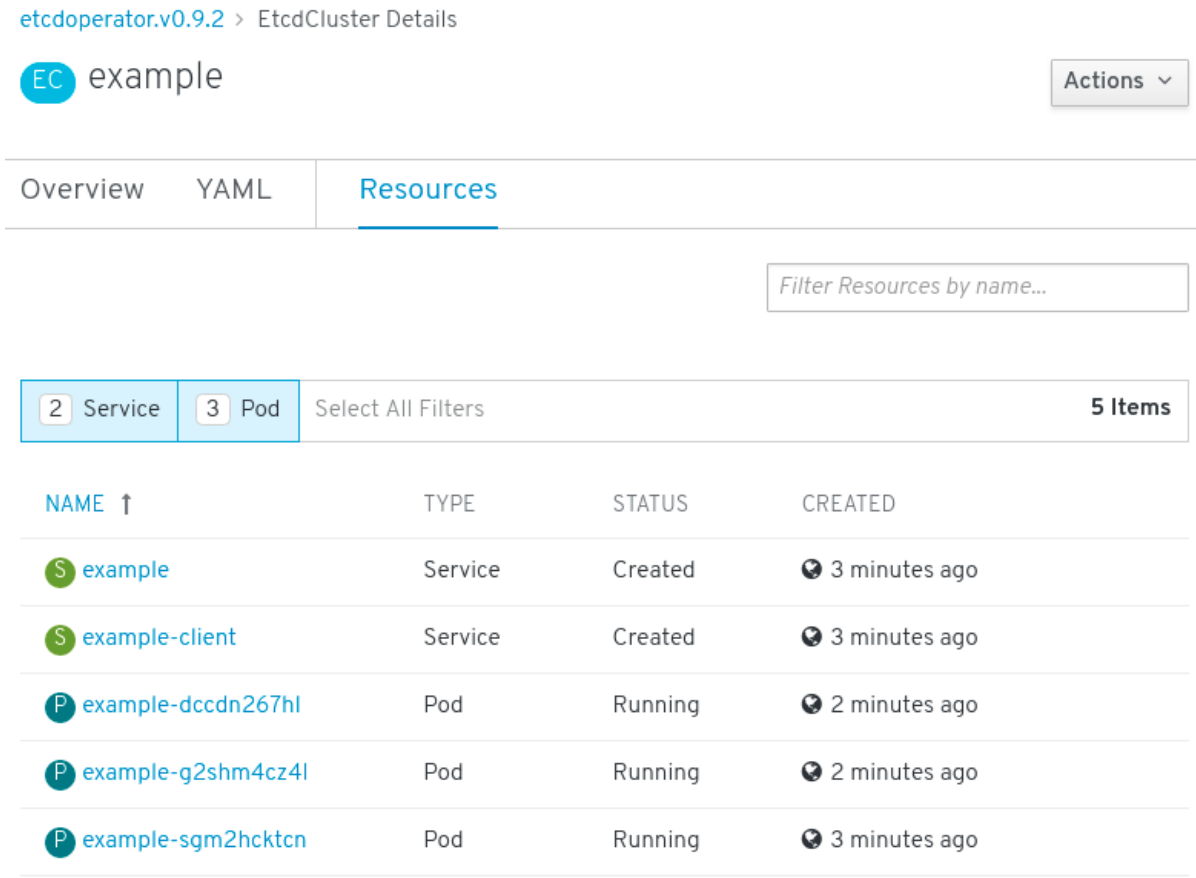
4. 新規 etcd クラスターを作成します。

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

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

5. **Resources** タブをクリックして、プロジェクトに Operator によって自動的に作成され、設定された数多くのリソースが含まれることを確認します。

図2.7 etcd Operator リソース



Kubernetes サービスが作成され、プロジェクトの他の Pod からデータベースにアクセスできることを確認します。

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

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

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

2.7. カスタムリソース定義からのリソースの管理

以下では、開発者がカスタムリソース定義 (CRD) にあるカスタムリソース (CR) をどのように管理できるかについて説明します。

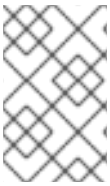
2.7.1. カスタムリソース定義

Kubernetes API では、リソースは特定の種類の API オブジェクトのコレクションを保管するエンドポイントです。たとえば、ビルトインされた Pod リソースには Pod オブジェクトのコレクションが含まれます。

カスタムリソース定義 (CRD) オブジェクトは、クラスター内に新規の固有オブジェクト **Kind** を定義し、Kubernetes API サーバーにそのライフサイクル全体を処理させます。

カスタムリソース (CR) オブジェクトは、クラスター管理者によってクラスターに追加された CRD から作成され、すべてのクラスターユーザーが新規リソースタイプをプロジェクトに追加できるようにします。

Operator はとりわけ CRD を必要な RBAC ポリシーおよび他のソフトウェア固有のロジックでパッケージ化することで CRD を利用します。またクラスター管理者は、Operator のライフサイクル外にあるクラスターに CRD を手動で追加でき、これらをすべてのユーザーに利用可能にすることができます。



注記

クラスター管理者のみが CRD を作成できる一方で、開発者は CRD への読み取りおよび書き込みパーミッションがある場合には、既存の CRD から CR を作成することができます。

2.7.2. ファイルからのカスタムリソースの作成

カスタムリソース定義 (CRD) がクラスターに追加された後に、クラスターリソース (CR) は CR 仕様を使用するファイルを使って CLI で作成できます。

前提条件

- CRD がクラスター管理者によってクラスターに追加されている。

手順

1. CR の YAML ファイルを作成します。以下の定義例では、**cronSpec** と **image** のカスタムフィールドが**Kind: CronTab** のCR に設定されます。この **Kind** は、CRD オブジェクトの **spec.kind** フィールドから取得します。

   CR の YAML ファイルサンプル

   ```
   apiVersion: "stable.example.com/v1" 1
   kind: CronTab 2
   metadata:
     name: my-new-cron-object 3
     finalizers: 4
     - finalizer.stable.example.com
   spec: 5
     cronSpec: "* * * * /5"
     image: my-awesome-cron-image
   ```

   1 カスタムリソース定義からグループ名および API バージョン (名前/バージョン) を指定します。

   2 CRD にタイプを指定します。

   3 オブジェクトの名前を指定します。

   4 オブジェクトのファイナライザーを指定します (ある場合)。ファイナライザーは、コントローラーがオブジェクトの削除前に完了する必要のある条件を実装できるようにします。

5 オブジェクトのタイプに固有の条件を指定します。

2. ファイルの作成後に、オブジェクトを作成します。

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

2.7.3. カスタムリソースの検査

CLI を使用してクラスターに存在するカスタムリソース (CR) オブジェクトを検査できます。

前提条件

- CR オブジェクトがアクセスできる namespace にあること。

手順

1. CR の特定の **Kind** についての情報を取得するには、以下を実行します。

```
$ oc get <kind>
```

例:

```
$ oc get crontab

NAME                 KIND
my-new-cron-object   CronTab.v1.stable.example.com
```

リソース名では大文字と小文字が区別されず、CRD で定義される単数形または複数形のいずれか、および任意の短縮名を指定できます。例:

```
$ oc get crontabs
$ oc get crontab
$ oc get ct
```

2. CR の未加工の YAML データを確認することもできます。

```
$ oc get <kind> -o yaml
```

```
$ oc get ct -o yaml

apiVersion: v1
items:
- apiVersion: stable.example.com/v1
  kind: CronTab
  metadata:
    clusterName: ""
    creationTimestamp: 2017-05-31T12:56:35Z
    deletionGracePeriodSeconds: null
    deletionTimestamp: null
    name: my-new-cron-object
    namespace: default
    resourceVersion: "285"
```

```
  selfLink: /apis/stable.example.com/v1/namespaces/default/crontabs/my-new-cron-object
  uid: 9423255b-4600-11e7-af6a-28d2447dc82b
spec:
  cronSpec: '* * * * /5' 1
  image: my-awesome-cron-image 2
```

1 2 オブジェクトの作成に使用した YAML からのカスタムデータが表示されます。

第3章 アプリケーションライフサイクル管理

3.1. アプリケーションの作成

OpenShift Container Platform CLI を使用して、ソースまたはバイナリーコード、イメージおよびテンプレートを含むコンポーネントから OpenShift Container Platform アプリケーションを作成できます。

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

3.1.1. CLI を使用したアプリケーションの作成

3.1.1.1. 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、デプロイメント設定を作成して新規のイメージをデプロイするほか、サービスを作成してイメージを実行するデプロイメントへの負荷分散したアクセスを提供します。

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

3.1.1.1.1. ローカル

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

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

注記

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

3.1.1.1.2. 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、**BuildConfig** に挿入され、リポジトリーにアクセスできるようになります。

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

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

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

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

3.1.1.1.3. ビルドストラテジーの検出

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

それ以外の場合は、ソースビルドストラテジーが生成されます。

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

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

注記

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

3.1.1.1.4. 言語の検出

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

表3.1 new-app が検出する言語

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

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

言語の検出後、**new-app** は OpenShift Container Platform サーバーで、検出言語と一致する **supports** アノテーションを持つイメージストリームタグを検索するか、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、特定のソースリポジトリーにビルダーが使用するイメージを上書きすることができます。この方法を使用すると、ビルドストラテジーの検出および言語の検出は実行されない点に留意してください。

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

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

ローカルリポジトリーのソースを使用して `openshift/ruby-20-centos7:latest ` コンテナーのイメージストリームを作成する場合は、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、または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。

3.1.1.2. イメージからアプリケーションを作成する方法

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

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

注記

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

3.1.1.2.1. DockerHub MySQL イメージ

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

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

3.1.1.2.2. 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
```

3.1.1.2.3. 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オプション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
```

3.1.1.3. テンプレートからのアプリケーションの作成

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

保存したテンプレートからアプリケーションを作成します。以下は例になります。

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

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

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

3.1.1.3.1. 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場合は、以下のように**--param-file=-** を使用します。

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

3.1.1.4. アプリケーション作成の変更

new-app コマンドは、OpenShift Container Platform オブジェクトを生成します。このオブジェクトにより、作成されるアプリケーションがビルドされ、デプロイされ、実行されます。通常、これらのオブジェクトは現在のプロジェクトに作成され、これらのオブジェクトには入力ソースリポジトリーまたはインプットイメージから派生する名前が割り当てられます。ただし、**new-app** でこの動作を変更することができます。

表3.2 new-app 出力オブジェクト

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

3.1.1.4.1. 環境変数の指定

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

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

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

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

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

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

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。

3.1.1.4.2. ビルド環境変数の指定

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

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

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

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

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

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

3.1.1.4.3. 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
```

3.1.1.4.4. 作成前の出力の表示

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

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

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

3.1.1.4.5. 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
```

3.1.1.4.6. 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
```

3.1.1.4.7. 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。

3.1.1.4.8. 単一 Pod でのイメージとソースのグループ化

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

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

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

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

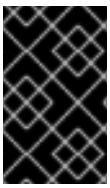
3.1.1.4.9. イメージ、テンプレート、および他の入力の検索

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

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

第4章 サービスブローカー

4.1. サービスカタログのインストール



重要

サービスカタログは OpenShift Container Platform 4 では非推奨になっています。同等または強化された機能は Operator Framework および Operator Lifecycle Manager (OLM) で提供されます。

4.1.1. サービスカタログについて

マイクロサービスベースのアプリケーションを開発して、クラウドネイティブのプラットフォームで実行する場合には、サービスプロバイダーやプラットフォームに合わせて、異なるリソースをプロビジョニングし、その位置情報 (coordinate)、認証情報および設定を共有する数多くの方法を利用できます。

開発者がよりシームレスに作業できるように、OpenShift Container Platform には Kubernetes 向けの Open Service Broker API (OSB API) の実装である **サービスカタログ** が含まれています。これにより、OpenShift Container Platform にデプロイされているアプリケーションをさまざまな種類のサービスブローカーに接続できます。

サービスカタログでは、クラスター管理者が 1 つの API 仕様を使用して、複数のプラットフォームを統合できます。OpenShift Container Platform Web コンソールは、サービスカタログにサービスブローカーによって提供されるクラスターサービスカタログを表示するので、ユーザーはこれらのサービスをそれぞれのアプリケーションで使用できるようにサービスの検出やインスタンス化を実行できます。

結果として、サービスのユーザーは異なるプロバイダーが提供する異なるタイプのサービスを簡単かつ一貫して使用できるという利点が得られます。また、サービスプロバイダーは、複数のプラットフォームにアクセスできる統合ポイントという利点を得られます。

サービスカタログは、OpenShift Container Platform 4 ではデフォルトでインストールされません。

4.1.2. サービスカタログのインストール

OpenShift Ansible Broker またはテンプレートサービスブローカーからサービスを使用することを計画する場合、以下の手順を実行してサービスカタログをインストールする必要があります。

サービスカタログの API サーバーおよびコントローラーマネージャーのカスタムリソースは OpenShift Container Platform にデフォルトで作成されますが、初期の状態は **managementState** が**Removed** になります。サービスカタログをインストールするには、それらのリソースの **managementState** を **Managed** に変更する必要があります。

手順

1. サービスカタログ API サーバーを有効にします。
   a. 以下のコマンドを使用して、サービスカタログ API サーバーリソースを編集します。

      ```
      $ oc edit servicecatalogapiservers
      ```

   b. **spec** の下で、**managementState** フィールドを **Managed** に設定します。

```
spec:
  logLevel: Normal
  managementState: Managed
```

c. 変更を適用するためにファイルを保存します。
Operator はサービスカタログ API サーバーコンポーネントをインストールします。
OpenShift Container Platform 4 の時点では、このコンポーネントは **openshift-service-catalog-apiserver** namespace にインストールされます。

2. サービスカタログコントローラーマネージャーを有効にします。

a. 以下のコマンドを使用して、サービスカタログコントローラーマネージャーのリソースを編集します。

```
$ oc edit servicecatalogcontrollermanagers
```

b. **spec** の下で、**managementState** フィールドを **Managed** に設定します。

```
spec:
  logLevel: Normal
  managementState: Managed
```

c. 変更を適用するためにファイルを保存します。
Operator はサービスカタログコントローラーマネージャーをインストールします。
OpenShift Container Platform 4 の時点で、このコンポーネントは **openshift-service-catalog-controller-manager** namespace にインストールされます。

4.1.3. サービスカタログのインストール

サービスカタログをアンインストールするには、サービスカタログの API サーバーの **managementState** を変更し、コントローラーマネージャーのリソース **Managed** から **Removed** に変更する必要があります。

手順

1. サービスカタログ API サーバーを無効にします。

a. 以下のコマンドを使用して、サービスカタログ API サーバーリソースを編集します。

```
$ oc edit servicecatalogapiservers
```

b. **spec** の下で、**managementState** フィールドを **Removed** に設定します。

```
spec:
  logLevel: Normal
  managementState: Removed
```

c. 変更を適用するためにファイルを保存します。

2. サービスカタログコントローラーマネージャーを無効にします。

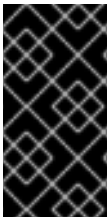
a. 以下のコマンドを使用して、サービスカタログコントローラーマネージャーのリソースを編集します。

```
$ oc edit servicecatalogcontrollermanagers
```

b. **spec** の下で、**managementState** フィールドを **Removed** に設定します。

```
spec:
  logLevel: Normal
  managementState: Removed
```

c. 変更を適用するためにファイルを保存します。



重要

サービスカタログを無効にした後に削除しようとする際に、プロジェクトが「Terminatin」状態になることに関連する既知の問題があります。回避策については、『OpenShift Container Platform 4.1 Release Notes 』を参照してください。(**BZ#1746174**)

4.2. テンプレートサービスブローカーのインストール

テンプレートサービスブローカーをインストールし、これが提供するテンプレートアプリケーションへのアクセスを取得します。

重要

テンプレートサービスブローカーは OpenShift Container Platform 4 では非推奨になっています。同等または強化された機能は Operator Framework および Operator Lifecycle Manager (OLM) で提供されます。

前提条件

- サービスカタログのインストール

4.2.1. テンプレートサービスブローカーについて

テンプレートサービスブローカー は、サービスカタログに対し、初期リリース以降 OpenShift Container Platform に同梱されるデフォルトのインスタントアプリケーションおよびクイックスタートテンプレートを可視化します。さらにテンプレートサービスブローカーは、Red Hat、クラスター管理者、またはユーザーないしはサードパーティーベンダーのいずれかが作成する OpenShift Container Platform テンプレートのいずれのコンテンツもサービスとして利用可能にすることができます。

デフォルトで、テンプレートサービスブローカーは **openshift** プロジェクトからグローバルに利用できるオブジェクトを表示します。また、これはクラスター管理者が選択する他のプロジェクトを監視するように設定することもできます。

テンプレートサービスブローカーは、OpenShift Container Platform 4 ではデフォルトでインストールされません。

4.2.2. テンプレートサービスブローカー Operator のインストール

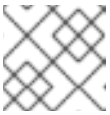
前提条件

- サービスカタログがインストールされていること。

手順

以下の手順では、Web コンソールを使用してテンプレートサービスブローカー Operator をインストールします。

1. namespace を作成します。
   a. Web コンソールで **Administration → Namespaces** に移動し、**Create Namespace** をクリックします。
   b. **Name** フィールドに **openshift-template-service-broker** を入力し、**Create** をクリックします。



注記

namespace は **openshift-** で開始する必要があります。

2. **Catalog → OperatorHub** ページに移動します。**openshift-template-service-broker** プロジェクトが選択されていることを確認します。
3. **Template Service Broker Operator** を選択します。
4. Operator についての情報を確認してから、**Install** をクリックします。
5. デフォルトの選択を確認し、**Subscribe** をクリックします。

次に、テンプレートサービスブローカーを起動し、これが提供するテンプレートアプリケーションへのアクセスを取得します。

4.2.3. テンプレートサービスブローカーの起動

テンプレートサービスブローカー Operator のインストール後に、以下の手順でテンプレートサービスブローカーを起動します。

前提条件

- サービスカタログがインストールされていること。
- テンプレートサービスブローカー Operator がインストールされていること。

手順

1. Web コンソールで **Catalog → Installed Operators** に移動し、**openshift-template-service-broker** プロジェクトを選択します。
2. **Template Service Broker Operator** を選択します。
3. **Provided APIs** で、**Template Service Broker** について **Create New** をクリックします。
4. デフォルトの YAML を確認し、**Create** をクリックします。
5. テンプレートサービスブローカーが起動していることを確認します。
   テンプレートサービスブローカーの起動後に、**Catalog → Developer Catalog** に移動し、**Service Class** チェックボックスを選択して利用可能なテンプレートアプリケーションを表示できます。テンプレートサービスブローカーが起動し、テンプレートアプリケーションが利用可能になるまで数分の時間がかかる場合があります。

これらのサービスクラスが表示されない場合は、以下の項目のステータスを確認できます。

- テンプレートサービスブローカー Pod のステータス
  - **openshift-template-service-broker** プロジェクトの **Workloads → Pods** ページから、**apiserver-** で起動する Pod のステータスが **Running** であり、**Ready** の準備状態であることを確認します。
- クラスターサービスブローカーのステータス
  - **Catalog → Broker Management → Service Brokers** ページから、**template-service-broker** サービスブローカーのステータスが **Ready** であることを確認します。
- サービスカタログコントローラーマネージャー Pod のログ
  - **openshift-service-catalog-controller-manager** プロジェクトの **Workloads → Pods** ページから、それぞれの Pod のログを確認し、**Successfully fetched catalog entries from broker** のメッセージと共にログエントリーが表示されていることを確認します。

4.3. テンプレートアプリケーションのプロビジョニング

4.3.1. テンプレートアプリケーションのプロビジョニング

以下の手順では、テンプレートサービスブローカーによって利用可能にされたサンプル PostgreSQL テンプレートアプリケーションをプロビジョニングします。

前提条件

- サービスカタログがインストールされていること。
- テンプレートサービスブローカーがインストールされていること。

手順

1. プロジェクトを作成します。
   a. Web コンソールで、**Home → Projects** に移動し、**Create Project** をクリックします。
   b. **test-postgresql** を **Name** フィールドに入力し、**Create** をクリックします。
2. サービスインスタンスを作成します。
   a. **Catalog → Developer Catalog** ページに移動します。
   b. **PostgreSQL (Ephemeral)** テンプレートアプリケーションを選択し、**Create Service Instance** をクリックします。
   c. デフォルトの選択を確認し、それ以外の必要なフィールドを設定してから **Create** をクリックします。
   d. **Catalog → Provisioned Services** に移動し、**postgresql-ephemeral** サービスインスタンスが作成され、ステータスが **Ready** であることを確認します。
   Home → Events ページで進捗を確認できます。しばらくすると、**postgresql-ephemeral** のイベントが「The instance was provisioned successfully」というメッセージと共に表示されるはずです。

3. サービスバインディングを作成します。

    a. **Provisioned Services** ページから、**postgresql-ephemeral** をクリックし、**Create Service Binding** をクリックします。

    b. デフォルトのサービスバインディング名を確認し、**Create** をクリックします。
    これにより、指定された名前を使用してバインディングの新規シークレットが作成されます。

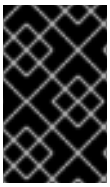
4. 作成されたシークレットを確認します。

    a. **Workloads → Secrets** に移動し、**postgresql-ephemeral** という名前のシークレットが作成されていることを確認します。

    b. **postgresql-ephemeral** をクリックし、他のアプリへのバインディングに使用されるキーと値のペアを **Data** セクションで確認します。

4.4. テンプレートサービスブローカーのアンインストール

テンプレートサービスブローカーは、これが提供するテンプレートアプリケーションへのアクセスを必要としなくなった場合にアンインストールできます。



重要

テンプレートサービスブローカーは OpenShift Container Platform 4 では非推奨になっています。同等または強化された機能は Operator Framework および Operator Lifecycle Manager (OLM) で提供されます。

4.4.1. テンプレートサービスブローカーのアンインストール

以下の手順では、Web コンソールを使用してテンプレートサービスブローカーおよび Operator をアンインストールします。

警告

クラスターにテンプレートサービスブローカーからプロビジョニングされたサービスがある場合には、これをアンインストールしないでください。アンインストールすると、サービスを管理しようとする際にエラーが生じる可能性があります。

前提条件

- テンプレートサービスブローカーがインストールされていること。

手順

この手順では、テンプレートサービスブローカーが **openshift-template-service-broker** プロジェクトにインストールされていることを前提とします。

1. テンプレートサービスブローカーのアンインストール

a. **Catalog → Installed Operators** に移動し、ドロップダウンメニューから **openshift-template-service-broker** プロジェクトを選択します。

b. **Template Service Broker Operator**をクリックします。

c. **Template Service Broker**タブを選択します。

d. **template-service-broker** をクリックします。

e. **Actions** ドロップダウンメニューから、**Delete Template Service Broker**を選択します。

f. 確認ポップアップ画面から **Delete** をクリックします。
テンプレートサービスブローカーのアンインストールが終了し、テンプレートアプリケーションは Developer Catalog からすぐに削除されます。

2. テンプレートサービスブローカー Operator のアンインストール

a. **Catalog → Operator Management** に移動し、ドロップダウンメニューから **openshift-template-service-broker** プロジェクトを選択します。

b. **Template Service Broker Operator**の **View subscription** をクリックします。

c. **templateservicebroker** を選択します。

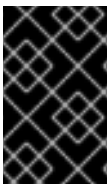
d. **Actions** ドロップダウンメニューから、**Remove Subscription** を選択します。

e. **Also completely remove the templateservicebroker Operator from the selected namespace** の横にあるチェックボックスにチェックが付けられていることを確認し、**Remove** をクリックします。
テンプレートサービスブローカー Operator がクラスターにインストールされていない状態になります。

テンプレートサービスブローカーのアンインストール後に、ユーザーはテンプレートサービスブローカーによって提供されるテンプレートアプリケーションにアクセスできなくなります。

4.5. OPENSHIFT ANSIBLE BROKER のインストール

OpenShift Ansible Broker をインストールし、これが提供するサービスバンドルへのアクセスを取得します。



重要

OpenShift Ansible Broker は OpenShift Container Platform 4 では非推奨になっています。同等または強化された機能は Operator Framework および Operator Lifecycle Manager (OLM) で提供されます。

前提条件

- サービスカタログのインストール

4.5.1. OpenShift Ansible Broker について

OpenShift Ansible Broker は、**Ansible playbook bundles** (APB) で定義されるアプリケーションを管理する Open Service Broker (OSB) API の実装です。APB は、OpenShift Container Platform のコンテナーアプリケーションを定義し、配信する方法を提供し、Ansible ランタイムと共にコンテナーイメー

ジに組み込まれた Ansible Playbook のバンドルで構成されています。APB は Ansible を活用し、複雑なデプロイメントを自動化する標準メカニズムを構築します。

OpenShift Ansible Brokerは、以下の基本的なワークフローに従います。

1. ユーザーは、OpenShift Container Platform Web コンソールを使用してサービスカタログから利用可能なアプリケーションの一覧を要求します。
2. サービスカタログは、OpenShift Ansible Broker から利用可能なアプリケーションの一覧を要求します。
3. OpenShift Ansible Broker は定義されたコンテナーイメージレジストリーと通信し、利用可能な APB の情報を得ます。
4. ユーザーは特定の APB をプロビジョニングする要求を実行します。
5. OpenShift Ansible Broker は、APB でプロビジョニングメソッドを呼び出して、ユーザーのプロビジョニング要求に対応します。

OpenShift Ansible Broker は、OpenShift Container Platform 4 ではデフォルトでインストールされません。

4.5.1.1. Ansible Playbook Bundle

Ansible Playbook Bundle (APB) は、Ansible ロールおよび Playbook の既存の投資を活用できるようにする軽量アプリケーション定義です。

APB は、名前の付いた Playbook が含まれる単純なディレクトリーを使用し、プロビジョニングやバインドなどの OSB API アクションを実行します。**apb.yml** ファイルで定義するメタデータには、デプロイメント時に使用する必須/任意のパラメーターの一覧が含まれています。

追加リソース

- Ansible Playbook Bundle リポジトリー

4.5.2. OpenShift Ansible Service Broker Operator のインストール

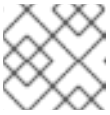
前提条件

- サービスカタログがインストールされていること。

手順

以下の手順では、Web コンソールを使用して OpenShift Ansible Service Broker Operator をインストールします。

1. namespace を作成します。
   a. Web コンソールで **Administration → Namespaces** に移動し、**Create Namespace** をクリックします。
   b. **openshift-ansible-service-broker** を **Name** フィールドに、**openshift.io/cluster-monitoring=true** を **Labels** フィールドに入力し、**Create** をクリックします。



注記

namespace は **openshift-** で開始する必要があります。

2. クラスターのロールバインディングを作成します。
   a. **Administration → Role Bindings** に移動し、**Create Binding** をクリックします。
   b. **Binding Type** については、**Cluster-wide Role Binding (ClusterRoleBinding)** を選択します。
   c. **Role Binding** については、**ansible-service-broker** を **Name** フィールドに入力します。
   d. **Role** については、**admin** を選択します。
   e. **Subject** については、**Service Account** オプションを選択し、**openshift-ansible-service-broker** namespace を選択して **openshift-ansible-service-broker-operator** を **Subject Name** フィールドに入力します。
   f. **Create** をクリックします。
3. Red Hat Container Catalog に接続するためにシークレットを作成します。
   a. **Workloads → Secrets** に移動します。**openshift-ansible-service-broker** プロジェクトが選択されていることを確認します。
   b. **Create → Key/Value Secret** をクリックします。
   c. **asb-registry-auth** を **シークレット名** として入力します。
   d. **username**の **Key** および Red Hat Container Catalog ユーザー名の **Value** を追加します。
   e. **Add Key/Value** をクリックし、**password** の **Key** および Red Hat Container Catalog パスワードの **Value** を追加します。
   f. **Create** をクリックします。
4. **Catalog → OperatorHub** ページに移動します。**openshift-ansible-service-broker** プロジェクトが選択されていることを確認します。
5. **OpenShift Ansible Service Broker Operator** を選択します。
6. Operator についての情報を確認してから、**Install** をクリックします。
7. デフォルトの選択を確認し、**Subscribe** をクリックします。

次に、OpenShift Ansible Broker を起動し、これが提供するサービスバンドルへのアクセスを取得します。

4.5.3. OpenShift Ansible Broker の起動

OpenShift Ansible Broker Operator のインストール後に、以下の手順で OpenShift Ansible Broker を起動します。

前提条件

- サービスカタログがインストールされていること。

- OpenShift Ansible Service Broker Operator がインストールされていること。

手順

1. Web コンソールで **Catalog → Installed Operators** に移動し、**openshift-ansible-service-broker** プロジェクトを選択します。
2. **OpenShift Ansible Service Broker Operator** を選択します。
3. **Provided APIs** で、**Automation Broker** について **Create New** をクリックします。
4. 以下を、提供されているデフォルトの YAML の **spec** フィールドに追加します。

```
registry:
  - name: rhcc
    type: rhcc
    url: https://registry.redhat.io
    auth_type: secret
    auth_name: asb-registry-auth
```

これは、OpenShift Ansible Service Broker Operator のインストール時に作成されたシークレットを参照します。これにより、Red Hat Container Catalog に接続できます。

5. 追加の OpenShift Ansible Broker 設定オプションを設定し、**Create** をクリックします。
6. OpenShift Ansible Broker が起動したことを確認します。
OpenShift Ansible Broker の起動後に、**Catalog → Developer Catalog** に移動し、**Service Class** チェックボックスを選択して利用可能なサービスバンドルを表示できます。OpenShift Ansible Broker が起動し、サービスバンドルが利用可能になるまで数分の時間がかかる場合があります。

これらのサービスクラスが表示されない場合は、以下の項目のステータスを確認できます。

- OpenShift Ansible Broker Pod のステータス
  - **openshift-ansible-service-broker** プロジェクトの **Workloads → Pods** ページから、**asb-** で起動する Pod のステータスが **Running** であり、**Ready** の準備状態であることを確認します。
- クラスターサービスブローカーのステータス
  - **Catalog → Broker Management → Service Brokers** ページから、**ansible-service-broker** サービスブローカーのステータスが **Ready** であることを確認します。
- サービスカタログコントローラーマネージャー Pod のログ
  - **openshift-service-catalog-controller-manager** プロジェクトの **Workloads → Pods** ページから、それぞれの Pod のログを確認し、**Successfully fetched catalog entries from broker** のメッセージと共にログエントリーが表示されていることを確認します。

4.5.3.1. OpenShift Ansible Broker 設定オプション

OpenShift Ansible Broker の以下のオプションを設定できます。

表4.1 OpenShift Ansible Broker 設定オプション

YAML キー	説明	デフォルト値
brokerName	Broker インスタンスを特定するために使用される名前。	**ansible-service-broker**
brokerNamespace	Broker が置かれている namespace。	**openshift-ansible-service-broker**
brokerImage	Broker に使用されている完全修飾イメージ。	**docker.io/ansibleplaybookbundle/origin-ansible-service-broker:v4.0**
brokerImagePullPolicy	Broker イメージ自体に使用されるプルポリシー。	**IfNotPresent**
brokerNodeSelector	Broker のデプロイメントに使用されるノードセレクター文字列。	**''**
registries	Broker レジストリー設定の yaml 一覧として表現される。これにより、ユーザーは Broker が検出し、APB の取得に使用するイメージレジストリーを設定できます。	デフォルトレジストリー配列 を参照してください。
logLevel	Broker のログに使用されるログレベル。	**info**
apbPullPolicy	APB Pod に使用されるプルポリシー。	**IfNotPresent**
sandboxRole	APB を実行するために使用されるサービスアカウントに付与されるロール。	**edit**
keepNamespace	APB の完了後に APB を実行するために作成された一時 namespace が削除されたかどうか (結果の如何は問わない)。	**false**
keepNamespaceOnError	結果がエラーの場合のみ、APB の完了後に APB を実行するために作成された一時 namespace が削除されたかどうか。	**false**
bootstrapOnStartup	Broker が起動時にブートストラップルーチンを実行する必要があるかどうか。	**true**
refreshInterval	APB のインベントリーを更新する Broker ブートストラップ間の間隔。	**600s**
launchApbOnBind	**Experimental:** バインド操作時に APB を実行する Broker を切り替えます。	**false**

YAML キー	説明	デフォルト値
autoEscalate	APB の実行中に、Broker がユーザーのパーミッションをエスカレートするかどうか。これは、Broker が起点となるユーザー承認を実行してユーザーが APB サンドボックスに付与されたパーミッションを持つようにするため、通常は **false** のままになります。	**false**
outputRequest	Broker が受信する低レベル HTTP 要求を出力するかどうか。	**false**

registries のデフォルト配列

```
- type: rhcc
  name: rhcc
  url: https://registry.redhat.io
  white_list:
  - ".*-apb$"
  auth_type: secret
  auth_name: asb-registry-auth
```

4.6. OPENSHIFT ANSIBLE BROKER の設定

重要

OpenShift Ansible Broker は OpenShift Container Platform 4 では非推奨になっています。同等または強化された機能は Operator Framework および Operator Lifecycle Manager (OLM) で提供されます。

4.6.1. OpenShift Ansible Broker の設定

以下の手順では、OpenShift Ansible Broker の設定をカスタマイズします。

前提条件

- OpenShift Ansible Broker がインストールされていること。

手順

この手順では、**ansible-service-broker** を OpenShift Ansible Broker 名前とインストール先のプロジェクトの両方に使用していることを前提とします。

1. Web コンソールで **Catalog → Installed Operators** に移動し、**ansible-service-broker** プロジェクトを選択します。
2. **OpenShift Ansible Service Broker Operator** を選択します。
3. **Automation Broker** タブで、**ansible-service-broker** を選択します。

4. **YAML** タブの **spec** フィールドの下で OpenShift Ansible Broker 設定オプションを追加するか、または更新します。
以下は例になります。

```
spec:
  keepNamespace: true
  sandboxRole: edit
```

5. **Save** をクリックして変更を適用します。

4.6.1.1. OpenShift Ansible Broker 設定オプション

OpenShift Ansible Broker の以下のオプションを設定できます。

表4.2 OpenShift Ansible Broker 設定オプション

YAML キー	説明	デフォルト値
brokerName	Broker インスタンスを特定するために使用される名前。	**ansible-service-broker**
brokerNamespace	Broker が置かれている namespace。	**openshift-ansible-service-broker**
brokerImage	Broker に使用されている完全修飾イメージ。	**docker.io/ansibleplaybookbundle/origin-ansible-service-broker:v4.0**
brokerImagePullPolicy	Broker イメージ自体に使用されるプルポリシー。	**IfNotPresent**
brokerNodeSelector	Broker のデプロイメントに使用されるノードセレクター文字列。	**''**
registries	Broker レジストリー設定の yaml 一覧として表現される。これにより、ユーザーは Broker が検出し、APB の取得に使用するイメージレジストリーを設定できます。	デフォルトレジストリー配列を参照してください。
logLevel	Broker のログに使用されるログレベル。	**info**
apbPullPolicy	APB Pod に使用されるプルポリシー。	**IfNotPresent**
sandboxRole	APB を実行するために使用されるサービスアカウントに付与されるロール。	**edit**
keepNamespace	APB の完了後に APB を実行するために作成された一時 namespace が削除されたかどうか (結果の如何は問わない)。	**false**

YAML キー	説明	デフォルト値
keepNamespaceOnError	結果がエラーの場合のみ、APB の完了後に APB を実行するために作成された一時 namespace が削除されたかどうか。	**false**
bootstrapOnStartup	Broker が起動時にブートストラップルーチンを実行する必要があるかどうか。	**true**
refreshInterval	APB のインベントリーを更新する Broker ブートストラップ間の間隔。	**600s**
launchApbOnBind	**Experimental:** バインド操作時に APB を実行する Broker を切り替えます。	**false**
autoEscalate	APB の実行中に、Broker がユーザーのパーミッションをエスカレートするかどうか。これは、Broker が起点となるユーザー承認を実行してユーザーが APB サンドボックスに付与されたパーミッションを持つようにするため、通常は **false** のままになります。	**false**
outputRequest	Broker が受信する低レベル HTTP 要求を出力するかどうか。	**false**

registries のデフォルト配列

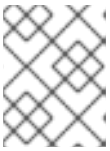
```
- type: rhcc
  name: rhcc
  url: https://registry.redhat.io
  white_list:
  - ".*-apb$"
  auth_type: secret
  auth_name: asb-registry-auth
```

4.6.2. OpenShift Ansible Broker のモニタリング設定

Prometheus が OpenShift Ansible Broker をモニターできるようにするには、以下のリソースを作成して、OpenShift Ansible Broker がインストールされている namespace にアクセスできるように Prometheus にパーミッションを付与する必要があります。

前提条件

- OpenShift Ansible Broker がインストールされていること。



注記

この手順では、OpenShift Ansible Broker が **openshift-ansible-service-broker** namespace にインストールされていることを前提とします。

手順

1. ロールを作成します。
   a. **Administration → Roles** に移動し、**Create Role** をクリックします。
   b. エディターで YAML を以下に置き換えます。

```
apiVersion: rbac.authorization.k8s.io/v1
kind: Role
metadata:
  name: prometheus-k8s
  namespace: openshift-ansible-service-broker
rules:
- apiGroups:
  - ""
  resources:
  - services
  - endpoints
  - pods
  verbs:
  - get
  - list
  - watch
```

   c. **Create** をクリックします。
2. ロールバインディングを作成します。
   a. **Administration → Role Bindings** に移動し、**Create Binding** をクリックします。
   b. **Binding Type** について、**Namespace Role Binding (RoleBinding)**を選択します。
   c. **Role Binding** について、**prometheus-k8s** を Name フィールドに、**openshift-ansible-service-broker** を Namespace フィールドに入力します。
   d. **Role** について、**prometheus-k8s** を選択します。
   e. **Subject** について、**Service Account** オプションを選択し、**openshift-monitoring** namespace を選択してから **prometheus-k8s** を **Subject Name** フィールドに入力します。
   f. **Create** をクリックします。

Prometheus は OpenShift Ansible Broker メトリクスにアクセスできるようになります。

4.7. サービスバンドルのプロビジョニング

4.7.1. サービスバンドルのプロビジョニング

以下の手順では、OpenShift Ansible Broker で利用可能にされている PostgreSQL サービスバンドル (APB) のサンプルをプロビジョニングします。

前提条件

- サービスカタログがインストールされていること。
- OpenShift Ansible Broker がインストールされていること。

手順

1. プロジェクトを作成します。
   a. Web コンソールで、**Home → Projects** に移動し、**Create Project** をクリックします。
   b. **test-postgresql-apb** を **Name** フィールドに入力し、**Create** をクリックします。
2. サービスインスタンスを作成します。
   a. **Catalog → Developer Catalog** ページに移動します。
   b. **PostgreSQL (APB)** サービスバンドルを選択し、**Create Service Instance** をクリックします。
   c. デフォルトの選択を確認し、それ以外の必要なフィールドを設定してから **Create** をクリックします。
   d. **Catalog → Provisioned Services** に移動し、**dh-postgresql-apb** サービスインスタンスが作成され、ステータスが **Ready** であることを確認します。
   Home → Events ページで進捗を確認できます。しばらくすると、**dh-postgresql-apb** のイベントが「The instance was provisioned successfully」というメッセージと共に表示されるはずです。
3. サービスバインディングを作成します。
   a. **Provisioned Services** ページから、**dh-postgresql-apb** をクリックし、**Create Service Binding** をクリックします。
   b. デフォルトのサービスバインディング名を確認し、**Create** をクリックします。
   これにより、指定された名前を使用してバインディングの新規シークレットが作成されます。
4. 作成されたシークレットを確認します。
   a. **Workloads → Secrets** に移動し、**dh-postgresql-apb** という名前のシークレットが作成されていることを確認します。
   b. **dh-postgresql-apb** をクリックし、他のアプリへのバインディングに使用されるキーと値のペアを **Data** セクションで確認します。

4.8. OPENSHIFT ANSIBLE BROKER のアンインストール

OpenShift Ansible Broker は、これが提供するサービスバンドルへのアクセスが必要なくなった場合にアンインストールできます。

重要

OpenShift Ansible Broker は OpenShift Container Platform 4 では非推奨になっています。同等または強化された機能は Operator Framework および Operator Lifecycle Manager (OLM) で提供されます。

4.8.1. OpenShift Ansible Broker のアンインストール

以下の手順では、Web コンソールを使用して OpenShift Ansible Broker およびその Operator をアンインストールします。

警告

クラスターに OpenShift Ansible Broker からプロビジョニングされたサービスがある場合には、これをアンインストールしないでください。アンインストールすると、サービスを管理しようとする際にエラーが生じる可能性があります。

前提条件

- OpenShift Ansible Broker がインストールされていること。

手順

この手順では、OpenShift Ansible Broker が **openshift-ansible-service-broker** プロジェクトにインストールされていることを前提とします。

1. OpenShift Ansible Broker をアンインストールします。
   a. **Catalog → Installed Operators** に移動し、ドロップダウンメニューから **openshift-ansible-service-broker** プロジェクトを選択します。
   b. **OpenShift Ansible Service Broker Operator** をクリックします。
   c. **Automation Broker** タブを選択します。
   d. **ansible-service-broker** をクリックします。
   e. **Actions** ドロップダウンメニューから、**Delete Automation Broker** を選択します。
   f. 確認ポップアップ画面から **Delete** をクリックします。
   OpenShift Ansible Broker のアンインストールが終了し、サービスバンドルは Developer Catalog からすぐに削除されます。
2. OpenShift Ansible Service Broker Operator のアンインストール
   a. **Catalog → Operator Management** に移動し、ドロップダウンメニューから **openshift-ansible-service-broker** プロジェクトを選択します。
   b. **OpenShift Ansible Broker Operator** の **View subscription** をクリックします。
   c. **automationbroker** を選択します。
   d. **Actions** ドロップダウンメニューから、**Remove Subscription** を選択します。
   e. **Also completely remove the automationbroker Operator from the selected namespace** の横にあるチェックボックスにチェックが付けられていることを確認し、**Remove** をクリックします。
   OpenShift Ansible Service Broker Operator がクラスターにインストールされていない状態になります。

OpenShift Ansible Broker のアンインストール後に、ユーザーは OpenShift Ansible Broker で提供されるサービス バンドルにアクセスできなくなります。

第5章 DEPLOYMENT

5.1. DEPLOYMENT および DEPLOYMENTCONFIG について

OpenShift Container Platform の **Deployment** および **DeploymentConfig** 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 DeploymentConfig または Deployment。
- DeploymentConfig には 1 つまたは複数の **ReplicationController** が使用され、これには Pod テンプレートとしての DeploymentConfig の特定の時点の状態のレコードが含まれます。同様に、Deployment には ReplicationController を継承する 1 つ以上の **ReplicaSet** が使用されます。
- アプリケーションの特定バージョンのインスタンスを表す 1 つ以上の Pod。

5.1.1. デプロイメントのビルディングブロック

Deployment および DeploymentConfig 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 ReplicationController および ReplicaSet の使用によって有効にされます。

ユーザーは、DeploymentConfig または Deployment によって所有される ReplicationController、ReplicaSet、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。

5.1.1.1. ReplicationController

ReplicationController は、Pod の指定された数のレプリカが常時実行されるようにします。Pod が終了するか、または削除される場合、ReplicationController 定義された数を満たすように追加のインスタンス化を行います。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。

ReplicationController 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 を特定するためのセレクター。

セレクターは、ReplicationController 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、ReplicationController がインスタンス化します。
ReplicationController は、必要に応じて調整できるよう、セレクターを使用してすでに実行中の Pod 数

を判別します。

ReplicationController は、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せず、追跡も実行しません。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。

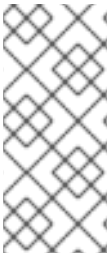
以下は ReplicationController 定義の例です。

```
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。
2. 実行する Pod のラベルセレクター。
3. コントローラーが作成する Pod のテンプレート。
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。

5.1.1.2. ReplicaSet

ReplicationController と同様に、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。ReplicaSet と ReplicationController の相違点は、ReplicaSet 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。



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 ReplicaSet を使用します。それ以外は Deployment を使用します。ReplicaSet は個別に使用できますが、Pod の作成/削除/更新のオーケストレーションを実行するためにデプロイメントで使用されます。Deployment は ReplicaSet を自動的に管理し、Pod に宣言型の更新を加えるので、作成する ReplicaSet 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。

5.1.2. DeploymentConfig

ReplicationController ベースにビルドされた OpenShift Container Platform は、**DeploymentConfig** の概念に基づいてソフトウェア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。最も単純なケースでは、DeploymentConfig は新規の ReplicationController を作成し、これに Pod を起動させます。

ただし、DeploymentConfig の OpenShift Container Platform デプロイメントは、イメージの既存デプロイメントから新規デプロイメントに移行する機能を提供し、ReplicationController の作成前後に実行されるフックも定義します。

DeploymentConfig デプロイメントシステムは以下の機能を提供します。

- アプリケーションを実行するためのテンプレートである DeploymentConfig。
- 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。ストラテジーは通常デプロイメントプロセスと呼ばれ、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 を作成すると、ReplicationController が、DeploymentConfig の Pod テンプレートとして作成されます。DeploymentConfig が変更されると、最新の Pod テンプレートで新しい ReplicationController 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 ReplicationController のスケールダウン、および新規 ReplicationController のスケールアップが行われます。

アプリケーションのインスタンスは、作成時にサービスローダーバランサーやルーターに対して自動的に追加/削除されます。アプリケーションが正常なシャットダウン機能をサポートしている限り、アプリケーションが **TERM** シグナルを受け取ると、実行中のユーザー接続が通常通り完了できるようにすることができます。

OpenShift Container Platform **DeploymentConfig** オブジェクトは以下の詳細を定義します。

1. **ReplicationController** 定義の要素。
2. 新規デプロイメントの自動作成のトリガー。
3. デプロイメント間の移行ストラテジー。
4. ライフサイクルフック。

デプロイヤー Pod は、デプロイメントがトリガーされるたびに、手動または自動であるかを問わず、(古い ReplicationController のスケールダウン、新規 ReplicationController のスケールアップ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、Deployment のログを維持するために Deployment 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 ReplicationController 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: 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) **ConfigChange** トリガーにより、新規 Deployment は ReplicationController テンプレートが変更されるたびに作成されます。

(2) **ImageChange** トリガーにより、新規 Deployment は、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たびに作成されます。

(3) デフォルトの **Rolling** ストラテジーにより、Deployment 間のダウンタイムなしの移行が行われます。

5.1.3. Deployment

Kubernetes は、**Deployment** という OpenShift Container Platform のファーストクラスのネイティブ API オブジェクトを提供します。Deployment は、OpenShift Container Platform 固有の DeploymentConfig として機能します。

DeploymentConfig の様に、Deployment は Pod テンプレートとして、アプリケーションの特定コンポーネントの必要な状態を記述します。Deployment は、Pod のライフサイクルをオーケストレーションする ReplicaSet を作成します。

たとえば、以下の Deployment 定義は ReplicaSet を作成し、1 つの **hello-openshift** Pod を起動します。

Deployment 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
```

5.1.4. Deployment および DeploymentConfig の比較

Kubernetes Deployment および OpenShift Container Platform でプロビジョニングされる DeploymentConfig の両方が OpenShift Container Platform でサポートされていますが、

DeploymentConfig で提供される特定の機能または動作が必要でない場合、Deployment を使用することが推奨されます。

以下のセクションでは、使用するタイプの決定に役立つ 2 つのオブジェクト間の違いを詳述します。

5.1.4.1. 設計

Deployment と DeploymentConfig 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 は整合性を優先しますが、Deployment は整合性よりも可用性を優先します。

DeploymentConfig 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、または手動で削除されます。ノードを手動で削除すると、対応する Pod も削除されます。つまり、kubelet は関連付けられた Pod も削除するため、Pod を削除してロールアウトの固定解除を行うことはできません。

一方、Deployment ロールアウトはコントローラーマネージャーから実行されます。コントローラーマネージャーはマスター上で高可用性モードで実行され、リーダー選択アルゴリズムを使用して可用性を整合性よりも優先するように設定します。障害の発生時には、他の複数のマスターが同時に同じ Deployment 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。

5.1.4.2. DeploymentConfig 固有の機能

自動ロールバック
現時点で、Deployment では、問題の発生時の最後に正常にデプロイされた ReplicaSet への自動ロールバックをサポートしていません。

トリガー
Deployment の場合、デプロイメントの Pod テンプレートに変更があるたびに新しいロールアウトが自動的にトリガーされるので、暗黙的な **ConfigChange** トリガーが含まれます。Pod テンプレートの変更時に新たなロールアウトが不要な場合には、デプロイメントを以下のように停止します。

```
$ oc rollout pause deployments/<name>
```

ライフサイクルフック
Deployment ではライフサイクルフックをサポートしていません。

カスタムストラテジー
Deployment 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。

5.1.4.3. Deployment 固有の機能

ロールオーバー
Deployment 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 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 は任意の数のアクティブな ReplicaSet を指定することができ、デプロイメントコントローラーがすべての古い ReplicaSet をスケールダウンし、最新の ReplicaSet をスケールアップします。

DeploymentConfig 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 ReplicationController 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 ReplicationController のみを一度にアクティブにできます。最終的には Deployment のロールアウトが加速します。

比例スケーリング

Deployment コントローラーのみが Deployment が所有する新旧の ReplicaSet のサイズについての信頼できる情報源であるため、継続中のロールアウトのスケーリングが可能です。追加のレプリカはそれぞれの ReplicaSet のサイズに比例して分散されます。

DeploymentConfig については、DeploymentConfig コントローラーが新規 ReplicationController のサイズに関してデプロイヤープロセスと競合するためにロールアウトが続行されている場合はスケーリングできません。

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。一方、デプロイヤー Pod は現時点で一時停止できないので、ロールアウト時に DeploymentConfig を一時停止しようとしても、デプロイヤープロセスはこの影響を受けず、完了するまで続行されます。

5.2. デプロイメントプロセスの管理

5.2.1. DeploymentConfig の管理

DeploymentConfig は、OpenShift Container Platform 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。

5.2.1.1. デプロイメントの開始

アプリケーションのデプロイメントプロセスを開始するために、**ロールアウト** を開始できます。

手順

1. 既存の DeploymentConfig から新規デプロイメントプロセスを開始するには、以下のコマンドを実行します。

```
$ oc rollout latest dc/<name>
```

注記

デプロイメントプロセスが進行中の場合には、このコマンドを実行すると、メッセージが表示され、新規 ReplicationController はデプロイされません。

5.2.1.2. デプロイメントの表示

アプリケーションの利用可能なすべてのリビジョンについての基本情報を取得するためにデプロイメントを表示できます。

手順

1. 現在実行中のデプロイメントプロセスを含む、指定した DeploymentConfig についての最近作成されたすべての ReplicationController についての詳細を表示するには、以下を実行します。

```
$ oc rollout history dc/<name>
```

2. リビジョンに固有の詳細情報を表示するには、**--revision** フラグを追加します。

```
$ oc rollout history dc/<name> --revision=1
```

3. デプロイメント設定およびその最新バージョンの詳細については、**oc describe** コマンドを使用します。

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

5.2.1.3. デプロイメントの再試行

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

手順

1. 失敗したデプロイメントプロセスを再起動するには、以下を実行します。

```
$ oc rollout retry dc/<name>
```

最新リビジョンのデプロイメントに成功した場合には、このコマンドによりメッセージが表示され、デプロイメントプロセスはこれ以上試行されません。

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 ReplicationController は、失敗したときと同じ設定を使用します。

5.2.1.4. デプロイメントのロールバック

ロールバックすると、アプリケーションを以前のリビジョンに戻します。この操作は、REST API、CLI または Web コンソールで実行できます。

手順

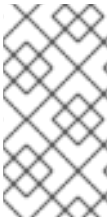
1. 最後にデプロイして成功した設定のリビジョンにロールバックするには以下を実行します。

```
$ oc rollout undo dc/<name>
```

DeploymentConfig 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 ReplicationController 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 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

DeploymentConfig 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。

5.2.1.5. コンテナー内でのコマンドの実行

コマンドをコンテナーに追加して、イメージの **ENTRYPOINT** を却下してコンテナーの起動動作を変更することができます。これは、指定したタイミングでデプロイメントごとに 1 回実行できるライフサイクルフックとは異なります。

手順

1. **command** パラメーターを、DeploymentConfig の **spec** フィールドを追加します。**command** コマンドを変更する **args** フィールドも追加できます (または **command** が存在しない場合には、**ENTRYPOINT**)。

```
spec:
  containers:
    -
    name: <container_name>
    image: 'image'
    command:
      - '<command>'
    args:
      - '<argument_1>'
      - '<argument_2>'
      - '<argument_3>'
```

たとえば、**-jar** および **/opt/app-root/springboots2idemo.jar** 引数を指定して、**java** コマンドを実行するには、以下を実行します。

```
spec:
  containers:
    -
    name: example-spring-boot
    image: 'image'
    command:
      - java
    args:
      - '-jar'
      - /opt/app-root/springboots2idemo.jar
```

5.2.1.6. デプロイメントログの表示

手順

1. 指定の DeploymentConfig 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 ReplicationController およびデプロイヤー 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>
```

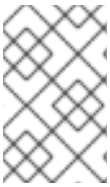
5.2.1.7. デプロイメントトリガー

DeploymentConfig 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 に定義されていない場合は、**ConfigChange** トリガーがデフォルトで追加されます。トリガーが空のフィールドとして定義されている場合には、デプロイメントは手動で起動する必要があります。

ConfigChange デプロイメントトリガー
ConfigChange トリガーにより、DeploymentConfig 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 ReplicationController が作成されます。



注記

ConfigChange トリガーが DeploymentConfig に定義されている場合は、DeploymentConfig 自体が作成された直後に、最初の ReplicationController が自動的に作成され、一時停止されません。

ConfigChange デプロイメントトリガー

```
triggers:
  - type: "ConfigChange"
```

ImageChange デプロイメントトリガー
ImageChange トリガーにより、イメージストリームタグの内容の変更時に常に新規 ReplicationController が生成されます (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)。

ImageChange デプロイメントトリガー

```
triggers:
  - type: "ImageChange"
    imageChangeParams:
      automatic: true (1)
      from:
        kind: "ImageStreamTag"
        name: "origin-ruby-sample:latest"
        namespace: "myproject"
      containerNames:
        - "helloworld"
```

(1) **imageChangeParams.automatic** フィールドが **false** に設定されると、トリガーが無効になります。

上記の例では、**origin-ruby-sample** イメージストリームの **latest** タグの値が変更され、新しいイメージの値が DeploymentConfig 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 ReplicationController が作成されます。

注記

ImageChange トリガーが DeploymentConfig (**ConfigChange** トリガーと **automatic=false**、または **automatic=true**) で定義されていて、**ImageChange** トリガーで参照されている **ImageStreamTag** がまだ存在していない場合には、ビルドにより、イメージが、**ImageStreamTag** にインポートまたはプッシュされた直後に初回のデプロイメントプロセスが自動的に開始されます。

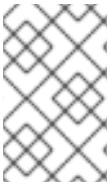
5.2.1.7.1. デプロイメントトリガーの設定

手順

1. **oc set triggers** コマンドを使用して、DeploymentConfig 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、**ImageChangeTrigger** 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>
```

5.2.1.8. デプロイメントリソースの設定



注記

このリソースはクラスター管理者が一時ストレージのテクノロジープレビュー機能を有効にしている場合にのみ利用できます。この機能はデフォルトでは無効にされています。

デプロイメントは、ノードでリソース (メモリー、CPU 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。

デプロイメントストラテジーの一部としてリソース制限を指定して、リソースの使用を制限することも可能です。デプロイメントリソースは、Recreate (再作成)、Rolling (ローリング) または Custom (カスタム) のデプロイメントストラテジーで使用できます。

手順

1. 以下の例では、**resources**、**cpu**、**memory**、および **ephemeral-storage** はそれぞれオプションです。

```
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。この項目は、クラスター管理者が一時ストレージのテクノロジープレビュー機能を有効にしている場合のみ該当します。

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:

```
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一覧に対応するリソース一覧を含みます。

- プロジェクトで定義される制限の範囲。**LimitRange** オブジェクトのデフォルト値が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。

5.2.1.9. 手動のスケーリング

ロールバック以外に、手動スケーリングにより、レプリカの数を詳細に管理できます。

注記

Pod は **oc autoscale** コマンドを使用して自動スケーリングすることも可能です。

手順

1. DeploymentConfig 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 のレプリカを **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 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。

5.2.1.10. DeploymentConfig 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 に追加し、プライベートリポジトリーからイメージにアクセスできるようにします。この手順では、OpenShift Container Platform Web コンソールを使用する方法を示します。

手順

1. 新規プロジェクトを作成します。
2. **Workloads** ページから、プライベートイメージリポジトリーにアクセスするための認証情報を含むシークレットを作成します。
3. DeploymentConfig を作成します。
4. DeploymentConfig エディターページで、**Pull Secret** を設定し、変更を保存します。

5.2.1.11. 特定のノードへの Pod の割り当て

ラベル付きのノードと合わせてノードセレクターを使用し、Pod の配置を制御することができます。

クラスター管理者は、プロジェクトに対して デフォルトのノードセレクターを設定して 特定のノードに Pod の配置を制限できます。開発者は、Pod 設定にノードセレクターを設定して、ノードをさらに制限することができます。

手順

1. Pod の作成時にノードセレクターを追加するには、Pod 設定を編集し、**nodeSelector** の値を追加します。これは、単一の Pod 設定や、Pod テンプレートに追加できます。

```
apiVersion: v1
kind: Pod
spec:
  nodeSelector:
    disktype: ssd
...
```

ノードセレクターが有効な場合に作成される Pod は指定されたラベルを持つノードに割り当てられます。ここで指定されるラベルは、クラスター管理者によって追加されるラベルと併用されます。

たとえば、プロジェクトに **type=user-node** と **region=east** のラベルがクラスター管理者により追加され、上記の **disktype: ssd** ラベルを Pod に追加した場合に、Pod は 3 つのラベルすべてが含まれるノードにのみスケジュールされます。

注記

ラベルには値を 1 つしか設定できないので、**region=east** が管理者によりデフォルト設定されている Pod 設定に **region=west** のノードセレクターを設定すると、Pod が全くスケジュールされなくなります。

5.2.1.12. 異なるサービスアカウントでの Pod の実行

デフォルト以外のサービスアカウントで Pod を実行できます。

手順

1. DeploymentConfig を編集します。

```
$ oc edit dc/<deployment_config>
```

2. **serviceAccount** と **serviceAccountName** パラメーターを **spec** フィールドに追加し、使用するサービスアカウントを指定します。

```
spec:
  securityContext: {}
  serviceAccount: <service_account>
  serviceAccountName: <service_account>
```

5.3. DEPLOYMENTCONFIG 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変更またはアップグレードする 1 つの方法です。この目的は、ユーザーには改善が加えられていることが分からないように、ダウンタイムなしに変更を加えることにあります。

エンドユーザーは通常ルーターによって処理されるルート経由でアプリケーションにアクセスするため、デプロイメントストラテジーは、DeploymentConfig 機能またはルーティング機能に重点を置きます。DeploymentConfig に重点を置くストラテジーは、アプリケーションを使用するすべてのルートに影響を与えます。ルーター機能を使用するストラテジーは個別のルートにターゲットを設定します。

デプロイメントストラテジーの多くは、DeploymentConfig でサポートされ、追加のストラテジーはルーター機能でサポートされます。このセクションでは、DeploymentConfig ストラテジーについて説明します。

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 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。

5.3.1. 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 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 check** 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

```
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 ReplicationController 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 ReplicationController をスケールダウンします。
4. 新しい ReplicationController が希望のレプリカ数に到達して、以前の ReplicationController の数がゼロになるまで、このスケーリングを繰り返します。
5. **post** ライフサイクルフックを実行します。

重要

スケールダウン時には、ローリングストラテジーは Pod の準備ができるまで待機し、スケーリングを行うことで可用性に影響が出るかどうかを判断します。Pod をスケールアップしたにもかかわらず、準備が整わない場合には、デプロイメントプロセスは最終的にタイムアウトして、デプロイメントに失敗します。

maxUnavailable パラメーターは、更新時に利用できない Pod の最大数です。**maxSurge** パラメーターは、元の Pod 数を超えてスケジュールできる Podの 最大数です。どちらのパラメーターも、パーセント (例: **10%**) または絶対値 (例: **2**) のいずれかに設定できます。両方のデフォルト値は **25%** です。

以下のパラメーターを使用して、デプロイメントの可用性やスピードを調整できます。以下は例になります。

- **maxUnavailable*=0** および **maxSurge*=20%** が指定されていると、更新時および急速なスケールアップ時に完全なキャパシティーが維持されるようになります。
- **maxUnavailable*=10%** および **maxSurge*=0** が指定されていると、追加のキャパシティーを使用せずに更新を実行します (インプレース更新)。
- **maxUnavailable*=10%** および **maxSurge*=10%** の場合は、キャパシティーが失われる可能性がありますが、迅速にスケールアップおよびスケールダウンします。

一般的に、迅速にロールアウトする場合は **maxSurge** を使用します。リソースのクォータを考慮して、一部に利用不可の状態が発生してもかまわない場合には、**maxUnavailable** を使用します。

5.3.1.1. カナリアデプロイメント

OpenShift Container Platform におけるすべてのローリングデプロイメントは **カナリアデプロイメント** です。新規バージョン (カナリア) はすべての古いインスタンスが置き換えられる前にテストされます。readiness チェックが成功しない場合、カナリアインスタンスは削除され、DeploymentConfig 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。

5.3.1.2. ローリングデプロイメントの作成

ローリングデプロイメントは OpenShift Container Platform のデフォルトタイプです。CLI を使用してローリングデプロイメントを作成できます。

手順

1. DockerHub にあるデプロイメントイメージの例を基にアプリケーションを作成します。

```
$ oc new-app openshift/deployment-example
```

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してください (または、サービス IP を直接使用してください)。

```
$ oc expose svc/deployment-example
```

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 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時に、新規 ReplicationController は徐々にスケールアップされます。(rediness チェックをパスした後に) 新規 Pod に **ready** のマークが付けられると、デプロイメントプロセスは継続されます。

Pod が準備状態にならない場合、プロセスは中止し、DeploymentConfig は直前のバージョンにロールバックします。

5.3.2. 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

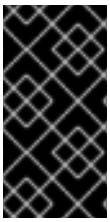
```
strategy:
  type: Recreate
  recreateParams: 1
    pre: {} 2
    mid: {}
    post: {}
```

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。

5.3.3. 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

```
strategy:
  type: Custom
  customParams:
    image: organization/strategy
    command: [ "command", "arg1" ]
    environment:
      - name: ENV_1
        value: VALUE_1
```

上記の例では、**organization/strategy** コンテナーイメージにより、デプロイメントの動作が提供されます。オプションの **command** 配列は、イメージの **Dockerfile** で指定した **CMD** ディレクティブを上書きします。指定したオプションの環境変数は、ストラテジープロセスの実行環境に追加されます。

さらに、OpenShift Container Platform は以下の環境変数をデプロイメントプロセスに提供します。

環境変数	説明
OPENSHIFT_DEPLOYMENT_NAME	新規デプロイメント名 (ReplicationController)
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** を使用して、カスタムのデプロイメントロジックを、既存のデプロイメントストラテジーに挿入します。カスタムのシェルスクリプトロジックを指定して、**openshift-deploy** バイナリーを呼び出します。カスタムのデプロイヤーコンテナーイメージを用意する必要はありません。ここでは、代わりにデフォルトの OpenShift Container Platform デプロイヤーイメージが使用されます。

```
strategy:
  type: Rolling
  customParams:
    command:
    - /bin/sh
    - -c
    - |
      set -e
      openshift-deploy --until=50%
      echo Halfway there
      openshift-deploy
      echo Complete
```

この設定により、以下のようなデプロイメントになります。

```
Started deployment #2
--> Scaling up custom-deployment-2 from 0 to 2, scaling down custom-deployment-1 from 2 to 0
(keep 2 pods available, don't exceed 3 pods)
    Scaling custom-deployment-2 up to 1
--> Reached 50% (currently 50%)
Halfway there
--> Scaling up custom-deployment-2 from 1 to 2, scaling down custom-deployment-1 from 2 to 0
(keep 2 pods available, don't exceed 3 pods)
    Scaling custom-deployment-1 down to 1
    Scaling custom-deployment-2 up to 2
    Scaling custom-deployment-1 down to 0
--> Success
Complete
```

カスタムデプロイメントストラテジーのプロセスでは、OpenShift Container Platform API または Kubernetes API へのアクセスが必要な場合には、ストラテジーを実行するコンテナーは、認証用のコンテナーで利用可能なサービスアカウントのトークンを使用できます。

5.3.4. 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 **failurePolicy** が含まれます。

Abort	フックに失敗すると、デプロイメントプロセスも失敗とみなされます。
Retry	フックの実行は、成功するまで再試行されます。
Ignore	フックの失敗は無視され、デプロイメントは続行されます。

フックには、フックの実行方法を記述するタイプ固有のフィールドがあります。現在、フックタイプとしてサポートされているのは Pod ベースのフックのみで、このフックは **execNewPod** フィールドで指定されます。

Pod ベースのライフサイクルフック

Pod ベースのライフサイクルフックは、DeploymentConfig 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 DeploymentConfig 例は簡素化されており、この例ではローリングストラテジーを使用します。簡潔にまとめられるように、トリガーおよびその他の詳細は省略しています。

```
kind: DeploymentConfig
apiVersion: v1
metadata:
  name: frontend
spec:
  template:
    metadata:
      labels:
        name: frontend
    spec:
      containers:
        - name: helloworld
          image: openshift/origin-ruby-sample
  replicas: 5
  selector:
    name: frontend
  strategy:
    type: Rolling
    rollingParams:
      pre:
        failurePolicy: Abort
        execNewPod:
          containerName: helloworld 1
          command: [ "/usr/bin/command", "arg1", "arg2" ] 2
          env: 3
            - name: CUSTOM_VAR1
              value: custom_value1
          volumes:
            - data 4
```

1 **helloworld** の名前は **spec.template.spec.containers[0].name** を参照します。

2 この **command** は、**openshift/origin-ruby-sample** イメージで定義される **ENTRYPOINT** を上書きします。

3 **env** は、フックコンテナーの環境変数です (任意)。

4 **volumes** は、フックコンテナーのボリューム参照です (任意)。

この例では、**pre** フックは、**helloworld** コンテナーからの **openshift/origin-ruby-sample** イメージを使用して新規 Pod で実行されます。フック Pod には以下のプロパティーが設定されます。

- フックコマンドは **/usr/bin/command arg1 arg2** 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 Pod から **data** ボリュームを継承します。

5.3.4.1. ライフサイクルフックの設定

CLI を使用して DeploymentConfig 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 \
    -v data --failure-policy=abort -- /usr/bin/command arg1 arg2
```

5.4. ルートベースのデプロイメントストラテジーの使用

デプロイメントストラテジーは、アプリケーションを進化させる手段として使用します。一部のストラテジーは DeploymentConfigs は、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 DeploymentConfig 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 (blue バージョン) を、テストと評価用に起動しつつ、安定版 (green バージョン) をユーザーが継続して使用します。準備が整ったら、blue バージョンに切り替えられます。問題が発生した場合には、green バージョンに戻すことができます。

一般的な別のストラテジーとして、**A/B バージョン** がいずれも、同時にアクティブな状態で、A バージョンを使用するユーザーも、B バージョンを使用するユーザーもいるという方法があります。これは、ユーザーインターフェースや他の機能の変更をテストして、ユーザーのフィードバックを取得するために使用できます。また、ユーザーに対する問題の影響が限られている場合に、実稼働のコンテキストで操作が正しく行われていることを検証するのに使用することもできます。

カナリアデプロイメントでは、新規バージョンをテストしますが、問題が検出されると、すぐに以前のバージョンにフォールバックされます。これは、上記のストラテジーどちらでも実行できます。

ルートベースのデプロイメントストラテジーでは、サービス内の Pod 数はスケーリングされません。希望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。

5.4.1. プロキシーシャードおよびトラフィック分割

実稼働環境で、特定のシャードに到達するトラフィックの分散を正確に制御できます。多くのインスタンスを扱う場合は、各シャードに相対的なスケールを使用して、割合ベースのトラフィックを実装できます。これは、他の場所で実行中の別のサービスやアプリケーションに転送または分割する **プロキシーシャード** とも適切に統合されます。

最も単純な設定では、プロキシーは要求を変更せずに転送します。より複雑な設定では、受信要求を複製して、別のクラスターだけでなく、アプリケーションのローカルインスタンスにも送信して、結果を比較することができます。他のパターンとしては、DR のインストールのキャッシュを保持したり、分析目的で受信トラフィックをサンプリングすることができます。

TCP (または UDP) のプロキシーは必要なシャードで実行できます。**oc scale** コマンドを使用して、プロキシーシャードで要求に対応するインスタンスの相対数を変更してください。より複雑なトラフィックを管理する場合には、OpenShift Container Platform ルーターを比例分散機能でカスタマイズすることを検討してください。

5.4.2. N-1 互換性

新規コードと以前のコードが同時に実行されるアプリケーションの場合は、新規コードで記述されたデータが、以前のバージョンのコードで読み込みや処理 (または正常に無視) できるように注意する必要があります。これは、**スキーマの進化**と呼ばれる複雑な問題です。

これは、ディスクに保存したデータ、データベース、一時的なキャッシュ、ユーザーのブラウザーセッションの一部など、多数の形式を取ることができます。多くの Web アプリケーションはローリングデプロイメントをサポートできますが、アプリケーションをテストし、設計してこれに対応させることが重要です。

アプリケーションによっては、新旧のコードが並行的に実行されている期間が短いため、バグやユーザーのトランザクションに失敗しても許容範囲である場合があります。別のアプリケーションでは失敗したパターンが原因で、アプリケーション全体が機能しなくなる場合もあります。

N-1 互換性を検証する 1 つの方法として、A/B デプロイメントを使用できます。 制御されたテスト環境で、以前のコードと新しいコードを同時に実行して、新規デプロイメントに流れるトラフィックが以前のデプロイメントで問題を発生させないかを確認します。

5.4.3. 正常な終了

OpenShift Container Platform および Kubernetes は、負荷分散のローテーションから削除する前にアプリケーションインスタンスがシャットダウンする時間を設定します。ただし、アプリケーションでは、終了前にユーザー接続が正常に中断されていることを確認する必要があります。

シャットダウン時に、OpenShift Container Platform はコンテナーのプロセスに **TERM** シグナルを送信します。**SIGTERM** を受信すると、アプリケーションコードは、新規接続の受け入れを停止します。これにより、ロードバランサーによって他のアクティブなインスタンスにトラフィックがルーティングされるようになります。アプリケーションコードは、開放されている接続がすべて終了する (または、次の機会に個別接続が正常に終了される) まで待機してから終了します。

正常に終了する期間が終わると、終了されていないプロセスに **KILL** シグナルが送信され、プロセスが即座に終了されます。Pod の **terminationGracePeriodSeconds** 属性または Pod テンプレート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。

5.4.4. Blue-Green デプロイメント

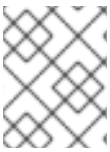
Blue-green デプロイメントでは、同時にアプリケーションの 2 つのバージョンを実行し、実稼働版 (green バージョン) からより新しいバージョン (blue バージョン) にトラフィックを移動します。ルートでは、ローリングストラテジーまたは切り替えサービスを使用できます。

多くのアプリケーションは永続データに依存するので、**N-1 互換性** をサポートするアプリケーションが必要です。つまり、データを共有して、データ層を 2 つ作成し、データベース、ストアまたはディスク間のライブマイグレーションを実装します。

新規バージョンのテストに使用するデータについて考えてみてください。実稼働データの場合には、新規バージョンのバグにより、実稼働版を破損してしまう可能性があります。

5.4.4.1. Blue-Green デプロイメントの設定

Blue-green デプロイメントでは 2 つの DeploymentConfig 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 DeploymentConfig 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 (blue) バージョンは有効になります。

必要に応じて以前のバージョンにサービスを切り替えて、以前の green バージョンにロールバックすることができます。

手順

1. アプリケーションサンプルの 2 つのコピーを作成します。

```
$ oc new-app openshift/deployment-example:v1 --name=example-green
$ oc new-app openshift/deployment-example:v2 --name=example-blue
```

   上記のコマンドにより、独立したアプリケーションコンポーネントが 2 つ作成されます。1 つは、**example-green** サービスで **v1** イメージを実行するコンポーネントと、もう 1 つは **example-blue** サービスで **v2** イメージを実行するコンポーネントです。

2. 以前のサービスを参照するルートを作成します。

```
$ oc expose svc/example-green --name=bluegreen-example
```

3. **example-green.<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-blue** に変更します。

```
$ oc patch route/bluegreen-example -p '{"spec":{"to":{"name":"example-blue"}}}'
```

5. ルートが変更されたことを確認するには、**v2** イメージが表示されるまで、ブラウザーを更新します。

5.4.5. A/B デプロイメント

A/B デプロイメントストラテジーでは、新しいバージョンのアプリケーションを実稼働環境での制限された方法で試すことができます。実稼働バージョンは、ユーザーの要求の大半に対応し、要求の一部が新しいバージョンに移動されるように指定できます。

各バージョンへの要求の割合を制御できるので、テストが進むにつれ、新しいバージョンへの要求を増やし、最終的に以前のバージョンの使用を停止することができます。各バージョン要求負荷を調整する際に、期待どおりのパフォーマンスを出せるように、各サービスの Pod 数もスケーリングする必要が生じる場合があります。

ソフトウェアのアップグレードに加え、この機能を使用してユーザーインターフェースのバージョンを検証することができます。以前のバージョンを使用するユーザーと、新しいバージョンを使用するユーザーが出てくるので、異なるバージョンに対するユーザーの反応を評価して、設計上の意思決定を知らせることができます。

このデプロイメントを有効にするには、以前のバージョンと新しいバージョンは同時に実行できるほど類似している必要があります。これは、バグ修正リリースや新機能が以前の機能と干渉しないようにする場合の一般的なポイントになります。これらのバージョンが正しく連携するには N-1 互換性が必要です。

OpenShift Container Platform は、Web コンソールと CLI で N-1 互換性をサポートします。

5.4.5.1. A/B テスト用の負荷分散

ユーザーは複数のサービスでルートを設定します。各サービスは、アプリケーションの 1 つのバージョンを処理します。

各サービスには **weight** が割り当てられ、各サービスへの要求の部分については **service_weight** を **sum_of_weights** で除算します。エンドポイントの **weights** の合計がサービスの **weight** になるように、サービスごとの **weight** がサービスのエンドポイントに分散されます。

ルートにはサービスを最大で 4 つ含めることができます。サービスの **weight** は、**0** から **256** の間で指定してください。**weight** が **0** の場合は、サービスはロードバランシングに参加せず、既存の持続する接続を継続的に提供します。サービスの **weight** が **0** でない場合は、エンドポイントの最小 **weight** は **1** となります。これにより、エンドポイントが多数含まれるサービスでは、最終的に **weight** は必要な値よりも大きくなる可能性があります。このような場合は、負荷分散の **weight** を必要なレベルに下げるために Pod の数を減らします。

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 DeploymentConfig を作成します。これらのアプリケーションは同じアプリケーションのバージョンであり、通常 1 つは現在の実稼働バージョンで、もう 1 つは提案される新規バージョンとなります。

```
$ oc new-app openshift/deployment-example --name=ab-example-a
$ oc new-app openshift/deployment-example --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。

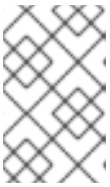
2. ルート経由でアプリケーションを外部から利用できるようにします。この時点でサービスを公

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-<project>.<router_domain> でアプリケーションを参照して、必要な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。
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。DeploymentConfig をスケーリングして、必要な負荷を処理できるように Pod 数を調整する必要がある場合があります。

ルートを編集するには、以下を実行します。

```
$ oc edit route <route_name>
...
metadata:
  name: route-alternate-service
  annotations:
    haproxy.router.openshift.io/balance: roundrobin
spec:
  host: ab-example.my-project.my-domain
  to:
    kind: Service
    name: ab-example-a
    weight: 10
  alternateBackends:
  - kind: Service
    name: ab-example-b
    weight: 15
...
```

5.4.5.1.1. Web コンソールを使用した重みの管理

手順

1. Route の詳細ページ (Applications/Routes) に移動します。
2. Actions メニューから **Edit** を選択します。
3. **Split traffic across multiple services** にチェックを入れます。
4. **Service Weights** スライダーで、各サービスに送信するトラフィックの割合を設定します。

3 つ以上のサービスにトラフィックを分割する場合には、各サービスに 0 から 256 の整数を使用して、相対的な重みを指定します。

トラフィックの重みは、トラフィックを分割したアプリケーションの行を展開すると **Overview** に表示されます。

5.4.5.1.2. 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、DeploymentConfig 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
NAME                KIND     TO          WEIGHT
routes/ab-example   Service  ab-example-a 198 (99%)
routes/ab-example   Service  ab-example-b 2   (1%)
```

3. **--adjust** フラグを使用すると、個別のサービスの重みを、それ自体に対して、または主要なサービスに対して相対的に変更できます。割合を指定すると、主要サービスまたは 1 番目の代用サービス (主要サービスを設定している場合) に対して相対的にサービスを調整できます。他にバックエンドがある場合には、重みは変更に比例した状態になります。
以下は例になります。

```
$ oc set route-backends ab-example --adjust ab-example-a=200 ab-example-b=10
$ oc set route-backends ab-example --adjust ab-example-b=5%
$ 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。

5.4.5.1.3.1 サービス、複数の DeploymentConfig

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
```

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 svc/ab-example-a --name=ab-example
```

3. **ab-example-<project>.<router_domain>** でアプリケーションを参照し、**v1** イメージが表示されることを確認します。

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"
```

5. この時点で、いずれの Pod もルートでサービスが提供されます。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 DeploymentConfig 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、以下を実行します。

```
$ oc edit dc/ab-example-b
```

第6章 CRD

6.1. カスタムリソース定義による KUBERNETES API の拡張

以下では、カスタムリソース定義 (CRD) を作成し、管理することで、クラスター管理者が OpenShift Container Platform クラスターをどのように拡張できるかについて説明します。

6.1.1. カスタムリソース定義

Kubernetes API では、リソースは特定の種類の API オブジェクトのコレクションを保管するエンドポイントです。たとえば、ビルトインされた Pod リソースには Pod オブジェクトのコレクションが含まれます。

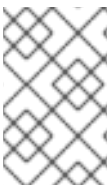
カスタムリソース定義 (CRD) オブジェクトは、クラスター内に新規の固有オブジェクト **Kind** を定義し、Kubernetes API サーバーにそのライフサイクル全体を処理させます。

カスタムリソース (CR) オブジェクトは、クラスター管理者によってクラスターに追加された CRD から作成され、すべてのクラスターユーザーが新規リソースタイプをプロジェクトに追加できるようにします。

クラスター管理者が新規 CRD をクラスターに追加する際に、Kubernetes API サーバーは、クラスター全体または単一プロジェクト (namespace) によってアクセスできる新規の RESTful リソースパスを作成することによって応答し、指定された CR を提供し始めます。

CRD へのアクセスを他のユーザーに付与する必要のあるクラスター管理者は、クラスターロールの集計を使用して **admin**、**edit**、または **view** のデフォルトクラスターロールを持つユーザーにアクセスを付与できます。また、クラスターロールの集計により、カスタムポリシールールをこれらのクラスターロールに挿入することができます。この動作は、新規リソースを組み込み型のインリソースであるかのようにクラスターの RBAC ポリシーに統合します。

Operator はとりわけ CRD を必要な RBAC ポリシーおよび他のソフトウェア固有のロジックでパッケージ化することで CRD を利用します。またクラスター管理者は、Operator のライフサイクル外にあるクラスターに CRD を手動で追加でき、これらをすべてのユーザーに利用可能にすることができます。



注記

クラスター管理者のみが CRD を作成できる一方で、開発者は CRD への読み取りおよび書き込みパーミッションがある場合には、既存の CRD から CR を作成することができます。

6.1.2. カスタムリソース定義の作成

カスタムリソース (CR) オブジェクトを作成するには、クラスター管理者はまずカスタムリソース定義 (CRD) を作成する必要があります。

前提条件

- **cluster-admin** ユーザー権限を使用した OpenShift Container Platform クラスターへのアクセス

手順

CRD を作成するには、以下を実行します。

1. 以下の例のようなフィールドタイプを含む YAML ファイルを作成します。

CRD の YAML ファイルサンプル

```
apiVersion: apiextensions.k8s.io/v1beta1 (1)
kind: CustomResourceDefinition
metadata:
  name: crontabs.stable.example.com (2)
spec:
  group: stable.example.com (3)
  version: v1 (4)
  scope: Namespaced (5)
  names:
    plural: crontabs (6)
    singular: crontab (7)
    kind: CronTab (8)
    shortNames:
    - ct (9)
```

1. **apiextensions.k8s.io/v1beta1** API を使用します。
2. 定義の名前を指定します。これは **group** および **plural** フィールドの値を使用する <plural-name>.<group> 形式である必要があります。
3. API のグループ名を指定します。API グループは、論理的に関連付けられるオブジェクトのコレクションです。たとえば、**Job** または **ScheduledJob** などのすべてのバッチオブジェクトはバッチ API グループ (batch.api.example.com など) である可能性があります。組織の完全修飾ドメイン名を使用することが奨励されます。
4. URL で使用されるバージョン名を指定します。それぞれの API グループは複数バージョンで存在させることができます。たとえば、**v1alpha**、**v1beta**、**v1** などが使用されます。
5. カスタムオブジェクトがクラスター (**Cluster**) の 1 つのプロジェクト (**Namespaced**) またはすべてのプロジェクトで利用可能であるかどうかを指定します。
6. URL で使用される複数形の名前を指定します。**plural** フィールドは API URL のリソースと同じになります。
7. CLI および表示用にエイリアスとして使用される単数形の名前を指定します。
8. 作成できるオブジェクトの種類を指定します。タイプは CamelCase にすることができます。
9. CLI でリソースに一致する短い文字列を指定します。

注記

デフォルトで、CRD のスコープはクラスターで設定され、すべてのプロジェクトで利用可能です。

2. CRD オブジェクトを作成します。

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

新規の RESTful API エンドポイントは以下のように作成されます。

```
/apis/<spec:group>/<spec:version>/<scope>/*/<names-plural>/...
```

たとえば、サンプルファイルを使用すると、以下のエンドポイントが作成されます。

```
/apis/stable.example.com/v1/namespaces/*/crontabs/...
```

このエンドポイント URL を使用して CR を作成し、管理できます。オブジェクトの **Kind** は、作成した CRD オブジェクトの **spec.kind** フィールドに基づいています。

6.1.3. カスタムリソース定義のクラスターロールの作成

クラスター管理者は、既存のクラスタースコープのカスタムリソース定義 (CRD) にパーミッションを付与できます。**admin**、**edit**、および **view** のデフォルトクラスターロールを使用する場合、これらのルールにクラスターロールの集計を利用します。

重要

これらのロールのいずれかにパーミッションを付与する際は、明示的に付与する必要があります。より多くのパーミッションを持つロールはより少ないパーミッションを持つロールからルールを継承しません。ルールをあるロールに割り当てる場合、より多くのパーミッションを持つロールにもその動詞を割り当てる必要もあります。たとえば、**get crontabs** パーミッションを表示ロールに付与する場合、これを **edit** および **admin** ロールにも付与する必要があります。**admin** または **edit** ロールは通常、プロジェクトテンプレートでプロジェクトを作成したユーザーに割り当てられます。

前提条件

- CRD を作成します。

手順

1. CRD のクラスターロール定義ファイルを作成します。クラスターロール定義は、各クラスターロールに適用されるルールが含まれる YAML ファイルです。OpenShift Container Platform Controller はデフォルトクラスターロールに指定するルールを追加します。

カスタムロール定義の YAML ファイルサンプル

```
kind: ClusterRole
apiVersion: rbac.authorization.k8s.io/v1 (1)
metadata:
  name: aggregate-cron-tabs-admin-edit (2)
  labels:
    rbac.authorization.k8s.io/aggregate-to-admin: "true" (3)
    rbac.authorization.k8s.io/aggregate-to-edit: "true" (4)
rules:
- apiGroups: ["stable.example.com"] (5)
  resources: ["crontabs"] (6)
  verbs: ["get", "list", "watch", "create", "update", "patch", "delete", "deletecollection"] (7)
---
kind: ClusterRole
apiVersion: rbac.authorization.k8s.io/v1
metadata:
  name: aggregate-cron-tabs-view (8)
```

```
  labels:
    # Add these permissions to the "view" default role.
    rbac.authorization.k8s.io/aggregate-to-view: "true" 9
    rbac.authorization.k8s.io/aggregate-to-cluster-reader: "true" 10
rules:
- apiGroups: ["stable.example.com"] 11
  resources: ["crontabs"] 12
  verbs: ["get", "list", "watch"] 13
```

1. **rbac.authorization.k8s.io/v1** API を使用します。
2. 8 定義の名前を指定します。
3. パーミッションを管理のデフォルトロールに付与するためにこのラベルを指定します。
4. パーミッションを編集のデフォルトロールに付与するためにこのラベルを指定します。
5. 11 CRD のグループ名を指定します
6. 12 これらのルールが適用される CRD の複数形の名前を指定します。
7. 13 ロールに付与されるパーミッションを表す動詞を指定します。たとえば、読み取りおよび書き込みパーミッションを **admin** および **edit** ロールに適用し、読み取り専用パーミッションを **view** ロールに適用します。
9. このラベルを指定して、パーミッションを **view** デフォルトロールに付与します。
10. このラベルを指定して、パーミッションを **cluster-reader** デフォルトロールに付与します。

2. クラスターロールを作成します。

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

6.1.4. ファイルからのカスタムリソースの作成

カスタムリソース定義 (CRD) がクラスターに追加された後に、クラスターリソース (CR) は CR 仕様を使用するファイルを使って CLI で作成できます。

前提条件

- CRD がクラスター管理者によってクラスターに追加されている。

手順

1. CR の YAML ファイルを作成します。以下の定義例では、**cronSpec** と **image** のカスタムフィールドが**Kind: CronTab** のCR に設定されます。この **Kind** は、CRD オブジェクトの **spec.kind** フィールドから取得します。

   CR の YAML ファイルサンプル

```
apiVersion: "stable.example.com/v1" 1
kind: CronTab 2
```

```
metadata:
  name: my-new-cron-object 3
  finalizers: 4
  - finalizer.stable.example.com
spec: 5
  cronSpec: "* * * * /5"
  image: my-awesome-cron-image
```

1. カスタムリソース定義からグループ名および API バージョン (名前/バージョン) を指定します。
2. CRD にタイプを指定します。
3. オブジェクトの名前を指定します。
4. オブジェクトのファイナライザーを指定します (ある場合)。ファイナライザーは、コントローラーがオブジェクトの削除前に完了する必要のある条件を実装できるようにします。
5. オブジェクトのタイプに固有の条件を指定します。

2. ファイルの作成後に、オブジェクトを作成します。

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

6.1.5. カスタムリソースの検査

CLI を使用してクラスターに存在するカスタムリソース (CR) オブジェクトを検査できます。

前提条件

- CR オブジェクトがアクセスできる namespace にあること。

手順

1. CR の特定の **Kind** についての情報を取得するには、以下を実行します。

```
$ oc get <kind>
```

例:

```
$ oc get crontab

NAME                 KIND
my-new-cron-object   CronTab.v1.stable.example.com
```

リソース名では大文字と小文字が区別されず、CRD で定義される単数形または複数形のいずれか、および任意の短縮名を指定できます。例:

```
$ oc get crontabs
$ oc get crontab
$ oc get ct
```

2. CR の未加工の YAML データを確認することもできます。

```
$ oc get <kind> -o yaml
```

```
$ oc get ct -o yaml

apiVersion: v1
items:
- apiVersion: stable.example.com/v1
  kind: CronTab
  metadata:
    clusterName: ""
    creationTimestamp: 2017-05-31T12:56:35Z
    deletionGracePeriodSeconds: null
    deletionTimestamp: null
    name: my-new-cron-object
    namespace: default
    resourceVersion: "285"
    selfLink: /apis/stable.example.com/v1/namespaces/default/crontabs/my-new-cron-object
    uid: 9423255b-4600-11e7-af6a-28d2447dc82b
  spec:
    cronSpec: '* * * * /5' 1
    image: my-awesome-cron-image 2
```

1 2 オブジェクトの作成に使用した YAML からのカスタムデータが表示されます。

6.2. カスタムリソース定義からのリソースの管理

以下では、開発者がカスタムリソース定義 (CRD) にあるカスタムリソース (CR) をどのように管理できるかについて説明します。

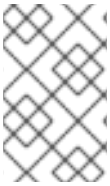
6.2.1. カスタムリソース定義

Kubernetes API では、リソースは特定の種類の API オブジェクトのコレクションを保管するエンドポイントです。たとえば、ビルトインされた Pod リソースには Pod オブジェクトのコレクションが含まれます。

カスタムリソース定義 (CRD) オブジェクトは、クラスター内に新規の固有オブジェクト **Kind** を定義し、Kubernetes API サーバーにそのライフサイクル全体を処理させます。

カスタムリソース (CR) オブジェクトは、クラスター管理者によってクラスターに追加された CRD から作成され、すべてのクラスターユーザーが新規リソースタイプをプロジェクトに追加できるようにします。

Operator はとりわけ CRD を必要な RBAC ポリシーおよび他のソフトウェア固有のロジックでパッケージ化することで CRD を利用します。またクラスター管理者は、Operator のライフサイクル外にあるクラスターに CRD を手動で追加でき、これらをすべてのユーザーに利用可能にすることができます。



注記

クラスター管理者のみが CRD を作成できる一方で、開発者は CRD への読み取りおよび書き込みパーミッションがある場合には、既存の CRD から CR を作成することができます。

6.2.2. ファイルからのカスタムリソースの作成

カスタムリソース定義 (CRD) がクラスターに追加された後に、クラスターリソース (CR) は CR 仕様を使用するファイルを使って CLI で作成できます。

前提条件

- CRD がクラスター管理者によってクラスターに追加されている。

手順

1. CR の YAML ファイルを作成します。以下の定義例では、**cronSpec** と **image** のカスタムフィールドが**Kind: CronTab** のCR に設定されます。この **Kind** は、CRD オブジェクトの **spec.kind** フィールドから取得します。

   CR の YAML ファイルサンプル

   ```
   apiVersion: "stable.example.com/v1" 1
   kind: CronTab 2
   metadata:
     name: my-new-cron-object 3
     finalizers: 4
     - finalizer.stable.example.com
   spec: 5
     cronSpec: "* * * * /5"
     image: my-awesome-cron-image
   ```

   1 カスタムリソース定義からグループ名および API バージョン (名前/バージョン) を指定します。

   2 CRD にタイプを指定します。

   3 オブジェクトの名前を指定します。

   4 オブジェクトのファイナライザーを指定します (ある場合)。ファイナライザーは、コントローラーがオブジェクトの削除前に完了する必要のある条件を実装できるようにします。

   5 オブジェクトのタイプに固有の条件を指定します。

2. ファイルの作成後に、オブジェクトを作成します。

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

6.2.3. カスタムリソースの検査

CLI を使用してクラスターに存在するカスタムリソース (CR) オブジェクトを検査できます。

前提条件

- CR オブジェクトがアクセスできる namespace にあること。

手順

1. CR の特定の **Kind** についての情報を取得するには、以下を実行します。

```
$ oc get <kind>
```

例:

```
$ oc get crontab

NAME                 KIND
my-new-cron-object   CronTab.v1.stable.example.com
```

リソース名では大文字と小文字が区別されず、CRD で定義される単数形または複数形のいずれか、および任意の短縮名を指定できます。例:

```
$ oc get crontabs
$ oc get crontab
$ oc get ct
```

2. CR の未加工の YAML データを確認することもできます。

```
$ oc get <kind> -o yaml
```

```
$ oc get ct -o yaml

apiVersion: v1
items:
- apiVersion: stable.example.com/v1
  kind: CronTab
  metadata:
    clusterName: ""
    creationTimestamp: 2017-05-31T12:56:35Z
    deletionGracePeriodSeconds: null
    deletionTimestamp: null
    name: my-new-cron-object
    namespace: default
    resourceVersion: "285"
    selfLink: /apis/stable.example.com/v1/namespaces/default/crontabs/my-new-cron-object
    uid: 9423255b-4600-11e7-af6a-28d2447dc82b
  spec:
    cronSpec: '* * * * /5' 1
    image: my-awesome-cron-image 2
```

1 2 オブジェクトの作成に使用した YAML からのカスタムデータが表示されます。

第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。

本書では、リソースクォータの仕組みや、クラスター管理者がリソースクォータはプロジェクトごとにどのように設定し、管理できるか、および開発者やクラスター管理者がそれらをどのように表示できるかについて説明します。

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について説明します。

注記

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

表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、交換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、交換可能なものとして使用できます。
ephemeral-storage	非終了状態のすべての Pod での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、交換可能なものとして使用できます。このリソースは、一時ストレージのテクノロジープレビュー機能が有効にされている場合にのみ利用できます。この機能はデフォルトでは無効にされています。
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、交換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、交換可能なものとして使用できます。

リソース名	説明
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、交換可能なものとして使用できます。このリソースは、一時ストレージのテクノロジープレビュー機能が有効にされている場合にのみ利用できます。この機能はデフォルトでは無効にされています。
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。
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。このリソースは、一時ストレージのテクノロジープレビュー機能が有効にされている場合にのみ利用できます。この機能はデフォルトでは無効にされています。

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 Persistent Volume Claim (永続ボリューム要求、PVC) でのストレージ要求の合計は、この値を超えることができません。
persistentvolumeclaims	プロジェクトに存在できる Persistent Volume Claim (永続ボリューム要求、PVC) の合計数です。
<storage-class-name>.storageclass.storage.k8s.io/requests.storage	一致するストレージクラスを持つ、任意の状態のすべての Persistent Volume Claim (永続ボリューム要求、PVC) でのストレージ要求の合計はこの値を超えることができません。
<storage-class-name>.storageclass.storage.k8s.io/persistentvolumeclaims	プロジェクトに存在できる、一致するストレージクラスを持つ Persistent Volume Claim (永続ボリューム要求、PVC) の合計数です。

表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。

リソース名	説明
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。
persistentvolumeclaims	プロジェクトに存在できる Persistent Volume Claim (永続ボリューム要求、PVC) の合計数です。
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
Terminating	**spec.activeDeadlineSeconds >= 0** の Pod に一致します。
NotTerminating	**spec.activeDeadlineSeconds** が **nil** の Pod に一致します。
BestEffort	**cpu** または **memory** のいずれかについてのサービスの QoS (Quality of Service) が Best Effort の Pod に一致します。
NotBestEffort	**cpu** および **memory** についてのサービスの QoS (Quality of Service) が Best Effort ではない Pod に一致します。

BestEffort スコープは、以下のリソースに制限するようにクォータを制限します。

- **pods**

Terminating、**NotTerminating**、および **NotBestEffort** スコープは、以下のリソースを追跡するようにクォータを制限します。

- **pods**
- **memory**
- **requests.memory**
- **limits.memory**
- **cpu**
- **requests.cpu**
- **limits.cpu**
- **ephemeral-storage**
- **requests.ephemeral-storage**
- **limits.ephemeral-storage**

注記

一時ストレージ要求と制限は、テクノロジープレビューとして提供されている一時ストレージを有効にした場合にのみ適用されます。この機能はデフォルトでは無効にされています。

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 vs 制限

コンピュートリソースの割り当て時に、各コンテナーは CPU、メモリー、一時ストレージのそれぞれに要求値と制限値を指定できます。クォータはこれらの値のいずれも制限できます。

クォータに **requests.cpu** または **requests.memory** の値が指定されている場合、すべての着信コンテナーがそれらのリソースを明示的に要求することが求められます。クォータに **limits.cpu** または **limits.memory** の値が指定されている場合、すべての着信コンテナーがそれらのリソースの明示的な制限を指定することが求められます。

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 Persistent Volume Claim (永続ボリューム要求、PVC) の合計数です。

(3) プロジェクトに存在できる ReplicationController 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
    requests.ephemeral-storage: 2Gi 4
    limits.cpu: "2" 5
    limits.memory: 2Gi 6
    limits.ephemeral-storage: 4Gi 7
```

1 プロジェクトに存在できる非終了状態の Pod の合計数です。

2 非終了状態のすべての Pod において、CPU 要求の合計は 1 コアを超えることができません。

3 非終了状態のすべての Pod において、メモリー要求の合計は 1 Gi を超えることができません。

4 非終了状態のすべての Pod において、一時ストレージ要求の合計は 2 Gi を超えることができません。

5 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。

6 非終了状態のすべての Pod において、メモリー制限の合計は 2 Gi を超えることができません。

7 非終了状態のすべての Pod において、一時ストレージ制限の合計は 4 Gi を超えることができません。

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
```

```
    limits.ephemeral-storage: "4Gi" 4
  scopes:
  - NotTerminating 5
```

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 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この値を超えることができません。

5 クォータを**spec.activeDeadlineSeconds** が **nil** に設定されている一致する Pod のみに制限します。ビルド Pod は、**RestartNever** ポリシーが適用されない場合に **NotTerminating** になります。

compute-resources-time-bound.yaml

```
apiVersion: v1
kind: ResourceQuota
metadata:
  name: compute-resources-time-bound
spec:
  hard:
    pods: "2" 1
    limits.cpu: "1" 2
    limits.memory: "1Gi" 3
    limits.ephemeral-storage: "1Gi" 4
  scopes:
  - Terminating 5
```

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 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この値を超えることができません。

5 クォータを**spec.activeDeadlineSeconds >=0** に設定されている一致する Pod のみに制限します。たとえば、このクォータはビルド Pod またはデプロイヤー Pod に影響を与えますが、web サーバーまたはデータベースなどの長時間実行されない Pod には影響を与えません。

storage-consumption.yaml

```
apiVersion: v1
kind: ResourceQuota
metadata:
  name: storage-consumption
spec:
  hard:
    persistentvolumeclaims: "10" 1
```

```
requests.storage: "50Gi" 2
gold.storageclass.storage.k8s.io/requests.storage: "10Gi" 3
silver.storageclass.storage.k8s.io/requests.storage: "20Gi" 4
silver.storageclass.storage.k8s.io/persistentvolumeclaims: "5" 5
bronze.storageclass.storage.k8s.io/requests.storage: "0" 6
bronze.storageclass.storage.k8s.io/persistentvolumeclaims: "0" 7
```

1. プロジェクト内の Persistent Volume Claim (永続ボリューム要求、PVC) の合計数です。
2. プロジェクトのすべての Persistent Volume Claim (永続ボリューム要求、PVC) において、要求されるストレージの合計はこの値を超えることができません。
3. プロジェクトのすべての Persistent Volume Claim (永続ボリューム要求、PVC) において、gold ストレージクラスで要求されるストレージの合計はこの値を超えることができません。
4. プロジェクトのすべての Persistent Volume Claim (永続ボリューム要求、PVC) において、silver ストレージクラスで要求されるストレージの合計はこの値を超えることができません。
5. プロジェクトのすべての Persistent Volume Claim (永続ボリューム要求、PVC) において、silver ストレージクラスの要求の合計数はこの値を超えることができません。
6. プロジェクトのすべての Persistent Volume Claim (永続ボリューム要求、PVC) において、bronze ストレージクラスで要求されるストレージの合計はこの値を超えることができません。これが **0** に設定される場合、bronze ストレージクラスはストレージを要求できないことを意味します。
7. プロジェクトのすべての Persistent Volume Claim (永続ボリューム要求、PVC) において、bronze ストレージクラスで要求されるストレージの合計はこの値を超えることができません。これが **0** に設定される場合は、bronze ストレージクラスでは要求を作成できないことを意味します。

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>]
```

例:

```
$ 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** などの、OpenShift Container Platform の標準的な namespace を使用しているリソースタイプのすべてにオブジェクトカウントクォータを作成できます。オブジェクトクォータカウントは、定義されたクォータをすべての標準的な namespace を使用しているリソースタイプに設定します。

リソースクォータの使用時に、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。

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>** 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 グループの一覧に **oc api-resources** コマンドを使用します。

以下は例になります。

```
$ oc create quota test \
    --
hard=count/deployments.extensions=2,count/replicasets.extensions=4,count/pods=3,count/secrets=4
resourcequota "test" created
```

この例では、一覧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
Name:                   test
Namespace:                 quota
Resource                  Used  Hard
--------                  ----  ----
count/deployments.extensions  0     2
count/pods                    0     3
count/replicasets.extensions  0     4
count/secrets                 0     4
```

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、プレフィックス **requests.** のあるクォータ項目のみが拡張リソースに許可されます。以下は、GPU リソース **nvidia.com/gpu** のリソースクォータを設定する方法についてのシナリオ例です。

手順

1. クラスター内のノードで利用可能な GPU の数を判別します。以下は例になります。

```
# oc describe node ip-172-31-27-209.us-west-2.compute.internal | egrep
'Capacity|Allocatable|gpu'
                    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. 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。

```
# cat gpu-quota.yaml
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
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
Name:                   gpu-quota
Namespace:              nvidia
Resource                Used  Hard
--------                ----  ----
requests.nvidia.com/gpu  0     1
```

5. 単一 GPU を要求する Pod を実行します。

```
# oc create -f gpu-pod.yaml
```

```
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 が実行されていることを確認します。

```
# oc get pods
NAME            READY    STATUS     RESTARTS   AGE
gpu-pod-s46h7   1/1      Running    0          1m
```

7. 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
Name:                   gpu-quota
Namespace:              nvidia
Resource                Used  Hard
--------                ----  ----
requests.nvidia.com/gpu  1     1
```

8. **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
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一覧を取得します。たとえば、**demoproject** というプロジェクトの場合、以下を実行します。

```
$ oc get quota -n demoproject
NAME              AGE
besteffort        11m
compute-resources   2m
core-object-counts  29m
```

2. 関連するクォータについて記述します。たとえば、**core-object-counts** クォータの場合、以下を実行します。

```
$ oc describe quota core-object-counts -n demoproject
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. クォータの同期期間の設定

リソースのセットを削除する場合、クォータの使用が回復する前にユーザーがリソースの再利用を試行すると、問題が発生する可能性があります。リソースの同期タイムフレームは、クラスター管理者が設定できる **resource-quota-sync-period** 設定によって決定されます。

再生成時間の調整は、リソースの作成および自動化が使用される場合のリソース使用状況の判別に役立ちます。

注記

resource-quota-sync-period 設定は、システムパフォーマンスのバランスを取るように設計されています。同期期間を短縮すると、マスターに大きな負荷がかかる可能性があります。

手順

クォータの同期期間を設定するには、以下を実行します。

1. Kubernetes コントローラーマネージャーを編集します。

```
$ oc edit kubecontrollermanager cluster
```

2. **unsupportedconfigOverrides** フィールドを、以下の設定で、**resource-quota-sync-period** フィールドに時間 (秒単位) を指定するように変更します。

```
unsupportedConfigOverrides:
  extendedArguments:
    resource-quota-sync-period:
    - 60s
```

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について説明します。

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: 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: 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 ClusterResourceQuota 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** リソースを使っ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
```

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 アプリケーションの正常性のモニタリング

ソフトウェアのシステムでは、コンポーネントは一時的な問題 (一時的に接続が失われるなど)、設定エラー、または外部の依存関係に関する問題などにより正常でなくなることがあります。OpenShift Container Platform アプリケーションには、正常でないコンテナーを検出し、これに対応するための数多くのオプションがあります。

8.1. ヘルスチェックについて

プローブは実行中のコンテナーで定期的に実行する Kubernetes の動作です。現時点では、2 つのタイプのプローブがあり、それぞれが目的別に使用されています。

readiness プローブ

readiness チェックは、スケジュールの対象になるコンテナーでサービス要求に対応する準備が整っているかどうかを判別します。readiness プローブがコンテナーで失敗する場合、エンドポイントコントローラーはコンテナーの IP アドレスがすべてのエンドポイントから削除されるようにします。readiness プローブを使用すると、コンテナーが実行されていても、それがプロキシーからトラフィックを受信しないようエンドポイントコントローラーに信号を送ることができます。

たとえば、readiness チェックでは、どの Pod を使用するかを制御することができます。Pod の準備ができない場合、削除されます。

liveness プローブ

liveness チェックは、スケジュールされているコンテナーがまだ実行中であるかどうかを判断します。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。 その後に、コンテナー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** が設定されているノードでの liveness プローブは、ノード上のコンテナーを強制終了してから、これを再起動します。

liveness チェックの例

```
apiVersion: v1
kind: Pod
metadata:
  labels:
    test: liveness
  name: liveness-http
spec:
  containers:
  - name: liveness-http
    image: k8s.gcr.io/liveness (1)
    args:
    - /server
    livenessProbe: (2)
      httpGet: (3)
        # host: my-host
        # scheme: HTTPS
        path: /healthz
        port: 8080
        httpHeaders:
        - name: X-Custom-Header
          value: Awesome
```

```
    initialDelaySeconds: 15 4
    timeoutSeconds: 1 5
  name: liveness 6
```

1 liveness プローブに使用するイメージを指定します。

2 ヘルスチェックのタイプを指定します。

3 liveness チェックのタイプを指定します。

- HTTP チェック。**httpGet** を指定します。
- コンテナー実行チェック。**exec** を指定します。
- TCP ソケットチェック。**tcpSocket** を指定します。

4 コンテナーが起動してから最初のプローブが実行されるまでの秒数を指定します。

5 プローブ間の秒数を指定します。

正常ではないコンテナーについての liveness チェック出力の例

```
$ oc describe pod pod1

....

FirstSeen LastSeen    Count   From            SubobjectPath          Type       Reason      Message
--------- --------    -----   ----            -------------          --------   ------      -------
37s       37s     1   {default-scheduler }                       Normal     Scheduled   Successfully assigned
liveness-exec to worker0
36s       36s     1   {kubelet worker0}   spec.containers{liveness}   Normal     Pulling     pulling image
"k8s.gcr.io/busybox"
36s       36s     1   {kubelet worker0}   spec.containers{liveness}   Normal     Pulled      Successfully
pulled image "k8s.gcr.io/busybox"
36s       36s     1   {kubelet worker0}   spec.containers{liveness}   Normal     Created     Created
container with docker id 86849c15382e; Security:[seccomp=unconfined]
36s       36s     1   {kubelet worker0}   spec.containers{liveness}   Normal     Started     Started
container with docker id 86849c15382e
2s        2s      1   {kubelet worker0}   spec.containers{liveness}   Warning    Unhealthy   Liveness
probe failed: cat: can't open '/tmp/healthy': No such file or directory
```

8.1.1. ヘルスチェックのタイプについて

liveness チェックと readiness チェックは 3 つの方法で設定できます。

HTTP チェック

kubelet は web hook を使用してコンテナーの正常性を判別します。このチェックは HTTP の応答コードが 200 から 399 までの値の場合に正常とみなされます。

HTTP チェックは、これが完全に初期化されている場合は HTTP ステータスコードを返すアプリケーションに適しています。

コンテナー実行チェック

kubeletは、コンテナー内でコマンドを実行します。ステータス 0 でチェックを終了すると、成功とみなされます。

TCP ソケットチェック

kubelet はコンテナーに対してソケットを開くことを試行します。コンテナーはチェックで接続を確立できる場合にのみ正常であるとみなされます。TCP ソケットチェック は、初期化が完了するまでリスニングを開始しないアプリケーションに適しています。

8.2. ヘルスチェックの設定

ヘルスチェックを設定するには、必要とされるチェックの種類ごとに Pod を作成します。

手順

ヘルスチェックを作成するには、以下の手順を実行します。

1. liveness コンテナー実行チェックを作成します。
   a. 以下のような YAML ファイルを作成します。

```
apiVersion: v1
kind: Pod
metadata:
  labels:
    test: liveness
  name: liveness-exec
spec:
  containers:
  - args:
    image: k8s.gcr.io/liveness
    livenessProbe:
      exec: 1
        command: 2
        - cat
        - /tmp/health
      initialDelaySeconds: 15 3
...
```

1. liveness チェックと liveness チェックのタイプを指定します。
2. コンテナー内で使用するコマンドを指定します。
3. コンテナーが起動してから最初のプローブが実行されるまでの秒数を指定します。

   b. ヘルスチェック Pod の状態を確認します。

```
$ oc describe pod liveness-exec

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
"k8s.gcr.io/liveness"
```

```
  Normal  Pulled     1s    kubelet, ip-10-0-143-40.ec2.internal  Successfully pulled image
"k8s.gcr.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
```

注記

timeoutSeconds パラメーターは、コンテナー実行チェックの readiness および liveness プローブには影響を与えません。OpenShift Container Platform はコンテナーへの実行呼び出しでタイムアウトにならないため、タイムアウトをプローブ自体に実装できます。プローブでタイムアウトを実装する 1 つの方法として、**timeout** パラメーターを使用して liveness プローブおよび readiness プローブを実行できます。

```
spec:
  containers:
    livenessProbe:
      exec:
        command:
          - /bin/bash
          - '-c'
          - timeout 60 /opt/eap/bin/livenessProbe.sh 1
      timeoutSeconds: 1
      periodSeconds: 10
      successThreshold: 1
      failureThreshold: 3
```

1 タイムアウト値およびプローブスクリプトへのパスです。

c. チェックを作成します。

```
$ oc create -f <file-name>.yaml
```

2. liveness TCP ソケットチェックを作成します。

a. 以下のような YAML ファイルを作成します。

```
apiVersion: v1
kind: Pod
metadata:
  labels:
    test: liveness
  name: liveness-tcp
spec:
  containers:
  - name: contaier1 1
    image: k8s.gcr.io/liveness
    ports:
    - containerPort: 8080 2
    livenessProbe: 3
      tcpSocket:
```

```
      port: 8080
    initialDelaySeconds: 15 4
    timeoutSeconds: 1 5
```

1 2 チェックの接続先としてのコンテナーの名前とポートを指定します。

3 liveness ヘルスチェックと liveness チェックのタイプを指定します。

4 コンテナーが起動してから最初のプローブが実行されるまでの秒数を指定します。

5 プローブ間の秒数を指定します。

b. チェックを作成します。

```
$ oc create -f <file-name>.yaml
```

3. readiness HTTP チェックを作成します。

a. 以下のような YAML ファイルを作成します。

```
apiVersion: v1
kind: Pod
metadata:
  labels:
    test: readiness
  name: readiness-http
spec:
  containers:
  - args:
    image: k8s.gcr.io/readiness 1
    readinessProbe: 2
    httpGet:
    # host: my-host 3
    # scheme: HTTPS 4
      path: /healthz
      port: 8080
    initialDelaySeconds: 15 5
    timeoutSeconds: 1 6
```

1 liveness プローブに使用するイメージを指定します。

2 readiness ヘルスチェックと readiness チェックのタイプを指定します。

3 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。

4 **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。

5 コンテナーが起動してから最初のプローブが実行されるまでの秒数を指定します。

6 プローブ間の秒数を指定します。

b. チェックを作成します。

```
$ oc create -f <file-name>.yaml
```

第9章 アプリケーションのアイドリング

クラスター管理者は、アプリケーションをアイドリング状態にしてリソース消費を減らすことができます。これは、コストがリソース消費と関連付けられるパブリッククラウドにデプロイされている場合に役立ちます。

スケーラブルなリソースが使用されていない場合、OpenShift Container Platform はリソースを検出した後にそれらを **0** レプリカに設定してアイドリングします。ネットワークトラフィックがリソースに送信される場合、レプリカをスケールアップしてアイドリング解除を実行し、通常の操作を続行します。

アプリケーションは複数のサービスや DeploymentConfig などの他のスケーラブルなリソースで構成されています。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。

9.1. 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、または **--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。

9.1.1. 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>
```

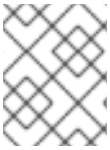
9.1.2. 複数サービスのアイドリング

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. 複数サービスの一覧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。

9.2. 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10章 リソースを回収するためのオブジェクトのプルーニング

時間の経過と共に、OpenShift Container Platform で作成される API オブジェクトは、アプリケーションのビルドおよびデプロイなどの通常のユーザーの操作によってクラスターの etcd データストアに蓄積されます。

クラスター管理者は、不要になった古いバージョンのオブジェクトをクラスターから定期的にプルーニングできま。たとえば、イメージのプルーニングにより、使用されなくなったものの、ディスク領域を使用している古いイメージや層を削除できます。

10.1. 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>**。

10.2. 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10.1 グループのプルーニング用の CLI の設定オプション

オプション	説明
--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。

prune コマンドが削除するグループを表示するには、以下を実行します。

```
$ oc adm prune groups --sync-file=ldap-sync-config.yaml
```

prune 操作を実行するには、以下を実行します。

```
$ oc adm prune groups --sync-file=ldap-sync-config.yaml --confirm
```

10.3. デプロイメントのプルーニング

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 deployments [<options>]
```

表10.2 デプロイメントのプルーニング用の CLI の設定オプション

オプション	説明
--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	DeploymentConfig 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デプロイメントをプルーニングします。
--keep-complete=<N>	DeploymentConfig 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** デプロイメントを維持します (デフォルト:**5**)。
--keep-failed=<N>	DeploymentConfig 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** デプロイメントを保持します (デフォルト:**1**)。
--keep-younger-than=<duration>	現在の時間との対比で **<duration>** より後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: **60m**)。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。

プルーニング操作によって削除されるものを確認するには、以下を実行します。

```
$ oc adm prune deployments --orphans --keep-complete=5 --keep-failed=1 \
    --keep-younger-than=60m
```

プルーニング操作を実際に実行するには、以下を実行します。

```
$ oc adm prune deployments --orphans --keep-complete=5 --keep-failed=1 \
    --keep-younger-than=60m --confirm
```

10.4. 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10.3 ビルドのプルーニング用の CLI の設定オプション

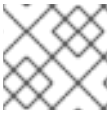
オプション	説明
--confirm	ドライランを実行する代わりにプルーニングが実行されることを示します。
--orphans	ビルド設定が存在せず、ステータスが complete (完了)、failed (失敗)、error (エラー)、または canceled (中止) のすべてのビルドをプルーニングします。
--keep-complete=<N>	ビルド設定基づいて、ステータスが complete (完了) の最後の **N** ビルドを保持します (デフォルト: **5**)。
--keep-failed=<N>	ビルド設定に基づいて、ステータスが failed (失敗)、error (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:**1**)。
--keep-younger-than=<duration>	現在の時間との対比で **<duration>** より後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: **60m**)。

プルーニング操作によって削除されるものを確認するには、以下を実行します。

```
$ oc adm prune builds --orphans --keep-complete=5 --keep-failed=1 \
    --keep-younger-than=60m
```

プルーニング操作を実際に実行するには、以下を実行します。

```
$ 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。

追加リソース

- Performing advanced builds → Pruning builds

10.5. イメージのプルーニング

使用年数やステータスまたは制限の超過によりシステムで不要となったイメージを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 images [<options>]
```

現時点でイメージをプルーニングするには、まずアクセストークンを使ってユーザーとして CLI にログインする必要があります。ユーザーにはクラスターロール **system:image-pruner** 以上のロールがなければなりません (例: **cluster-admin**)。

--prune-registry=false が使用されていない限り、イメージのプルーニングにより、統合レジストリーのデータが削除されます。この操作が適切に機能するには、**storage:delete:enabled** が **true** に設定された状態でレジストリーを設定する必要があります。

--namespace フラグの付いたイメージをプルーニングしてもイメージは削除されず、イメージストリームのみが削除されます。イメージは namespace を使用しないリソースです。そのため、プルーニングを特定の namespace に制限すると、イメージの現在の使用量を算出できなくなります。

デフォルトで、統合レジストリーは Blob メタデータをキャッシュしてストレージに対する要求数を減らし、要求の処理速度を高めます。プルーニングによって統合レジストリーのキャッシュが更新されることはありません。プルーニング後にプッシュされる、プルーニングされた層を含むイメージは破損します。キャッシュにメタデータを持つプルーニングされた層はプッシュされないためです。これは、レジストリーの再デプロイによって実行できます。

```
# oc patch deployment image-registry -n openshift-image-registry --type=merge --patch="{\"spec\":{\"template\":{\"metadata\":{\"annotations\":{\"kubectl.kubernetes.io/restartedAt\": \"$(date '+%Y-%m-%dT%H:%M:%SZ' -u)\"}}}}}"
```

統合レジストリーが Redis キャッシュを使用する場合、データベースを手動でクリーンアップする必要があります。

oc adm prune images 操作ではレジストリーのルートが必要です。レジストリーのルートはデフォルトでは作成されません。レジストリーのルートの作成方法に関する詳細は「Image Registry Operator in OpenShift Container Platform」を参照し、レジストリーサービスを公開する方法の詳細は「Exposing the registry」を参照してください。

表10.4 イメージのプルーニング用の CLI の設定オプション

オプション	説明
--all	レジストリーにプッシュされていないものの、プルスルー (pullthrough) でミラーリングされたイメージを組み込みます。これはデフォルトでオンに設定されます。プルーニングを統合レジストリーにプッシュされたイメージに制限するには、**--all=false** を渡します。
--certificate-authority	OpenShift Container Platform で管理されるレジストリーと通信する際に使用する認証局ファイルへのパスです。デフォルトは現行ユーザーの設定ファイルの認証局データに設定されます。これが指定されている場合、セキュアな通信が実行されます。
--confirm	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。これには、統合コンテナーイメージレジストリーへの有効なルートが必要になります。このコマンドがクラスターネットワーク外で実行される場合、ルートは **--registry-url** を使用して指定される必要があります。
--force-insecure	このオプションは注意して使用してください。HTTP 経由でホストされるか、または無効な HTTPS 証明書を持つコンテナーレジストリーへの非セキュアな接続を許可します。
--keep-tag-revisions=<N>	それぞれのイメージストリームについては、タグごとに最大 **N** のイメージリビジョンを保持します (デフォルト: **3**)。
--keep-younger-than=<duration>	現在の時間との対比で **<duration>** 未満の新しいイメージはプルーニングしません。現在の時間との対比で **<duration>** 未満の他のオブジェクトで参照されるイメージはプルーニングしません (デフォルト: **60m**)。

オプション	説明
--prune-over-size-limit	同じプロジェクトに定義される最小の制限を超える各イメージをプルーニングします。このフラグは **--keep-tag-revisions** または **--keep-younger-than** と共に使用することはできません。
--registry-url	レジストリーと通信する際に使用するアドレスです。このコマンドは、管理されるイメージおよびイメージストリームから判別されるクラスター内の URL の使用を試行します。これに失敗する (レジストリーを解決できないか、これにアクセスできない) 場合、このフラグを使用して他の機能するルートを指定する必要があります。レジストリーのホスト名の前には、特定の接続プロトコルを実施する **https://** または **http://** を付けることができます。
--prune-registry	他のオプションで規定される条件と共に、このオプションは、OpenShift Container Platform イメージ API オブジェクトに対応するレジストリーのデータがプルーニングされるかどうかを制御します。デフォルトで、イメージのプルーニングは、イメージ API オブジェクトとレジストリーの対応するデータの両方を処理します。このオプションは、イメージオブジェクトの数を減らすなどの目的で etcd の内容のみを削除することを検討していて (ただしレジストリーのストレージのクリーンアップは検討していない場合)、レジストリーの適切なメンテナンス期間中などにレジストリーのハードプルーニングによってこれを別途実行しようとする場合に役立ちます。

10.5.1. イメージのプルーニングの各種条件

- **--keep-younger-than** 分前よりも後に作成され、現時点で以下によって参照されていない OpenShift Container Platform で管理されるイメージ (アノテーション **openshift.io/image.managed** を持つイメージ) を削除します。
  - **--keep-younger-than** 分前よりも後に作成された Pod
  - **--keep-younger-than** 分前よりも後に作成されたイメージストリーム
  - 実行中の Pod
  - 保留中の Pod
  - ReplicationController
  - DeploymentConfig
  - ビルド設定
  - ビルド
  - **stream.status.tags[].items** の **--keep-tag-revisions**の最新のアイテム
- 同じプロジェクトで定義される最小の制限を超えており、現時点で以下によって参照されていない OpenShift Container Platform で管理されるイメージ (アノテーション **openshift.io/image.managed** を持つイメージ) を削除します。
  - 実行中の Pod

- 保留中の Pod
- ReplicationController
- DeploymentConfig
- ビルド設定
- ビルド

- 外部レジストリーからのプルーニングはサポートされていません。
- イメージがプルーニングされる際、イメージのすべての参照は **status.tags** にイメージの参照を持つすべてのイメージストリームから削除されます。
- イメージによって参照されなくなったイメージ層は削除されます。

注記

--prune-over-size-limit フラグは **--keep-tag-revisions** または **--keep-younger-than** フラグと共に使用することができません。これを実行すると、この操作が許可されないことを示す情報が返されます。

--prune-registry=false とその後にレジストリーのハードプルーニングを実行することで、OpenShift Container Platform イメージ API オブジェクトの削除とイメージデータのレジストリーからの削除を分離することができます。これにより、タイミングウィンドウが制限され、1 つのコマンドで両方をプルーニングする場合よりも安全に実行できるようになります。ただし、タイミングウィンドウを完全に取り除くことはできません。

たとえばプルーニングの実行時にプルーニング対象のイメージを特定する場合も、そのイメージを参照する Pod を引き続き作成することができます。また、プルーニングの操作時にイメージを参照している可能性のある API オブジェクトを追跡することもできます。これにより、削除されたコンテンツの参照に関連して発生する可能性のある問題を軽減することができます。

また、**--prune-registry** オプションを指定しないか、または **--prune-registry=true** を指定してプルーニングを再実行しても、**--prune-registry=false** を指定して以前にプルーニングされたイメージの、イメージレジストリー内で関連付けられたストレージがプルーニングされる訳ではないことに注意してください。**--prune-registry=false** を指定してプルーニングされたすべてのイメージは、レジストリーのハードプルーニングによってのみ削除できます。

10.5.2. イメージのプルーニング操作の実行

手順

1. プルーニング操作によって削除されるものを確認するには、以下を実行します。
   a. 最高 3 つのタグリビジョンを保持し、6 分前よりも後に作成されたリソース (イメージ、イメージストリームおよび Pod) を保持します。

```
$ oc adm prune images --keep-tag-revisions=3 --keep-younger-than=60m
```

   b. 定義された制限を超えるすべてのイメージをプルーニングします。

```
$ oc adm prune images --prune-over-size-limit
```

2. 前述のステップからオプションを指定してプルーニングの操作を実際に実行するには、以下を実行します。

```
$ oc adm prune images --keep-tag-revisions=3 --keep-younger-than=60m --confirm
```

```
$ oc adm prune images --prune-over-size-limit --confirm
```

10.5.3. セキュアまたは非セキュアな接続の使用

セキュアな通信の使用は優先され、推奨される方法です。これは、必須の証明書検証と共に HTTPS 経由で実行されます。**prune** コマンドは、可能な場合は常にセキュアな通信の使用を試行します。これを使用できない場合には、非セキュアな通信にフォールバックすることがあり、これには危険が伴います。この場合、証明書検証は省略されるか、または単純な HTTP プロトコルが使用されます。

非セキュアな通信へのフォールバックは、**--certificate-authority** が指定されていない場合、以下のケースで可能になります。

1. **prune** コマンドが **--force-insecure** オプションと共に実行される。
2. 指定される **registry-url** の前に **http://** スキームが付けられる。
3. 指定される **registry-url** はローカルリンクアドレスまたは **localhost** である。
4. 現行ユーザーの設定が非セキュアな接続を許可する。これは、ユーザーが **--insecure-skip-tls-verify** を使用してログインするか、またはプロンプトが出される際に非セキュアな接続を選択することによって生じる可能性があります。

重要

レジストリーのセキュリティーが、OpenShift Container Platform で使用されるものとは異なる認証局で保護される場合、これを **--certificate-authority** フラグを使用して指定する必要があります。そうしない場合、**prune** コマンドがエラーを出して失敗します。

10.5.4. イメージのプルーニングに関する問題

イメージがプルーニングされない
イメージが蓄積し続け、**prune** コマンドが予想よりも小規模な削除を実行する場合、プルーニング候補のイメージについて満たすべきイメージプルーティングの条件があることを確認します。

とくに削除する必要のあるイメージが、それぞれのタグ履歴において選択したタグリビジョンのしきい値よりも高い位置にあることを確認します。たとえば、**sha:abz** という名前の古く陳腐化したイメージがあるとします。イメージがタグ付けされている namespace **N** で以下のコマンドを実行すると、イメージが **myapp** という単一イメージストリームで 3 回タグ付けされていることに気づかれるでしょう。

```
$ image_name="sha:abz"
$ oc get is -n N -o go-template='{{range $isi, $is := .items}}{{range $ti, $tag := $is.status.tags}}'\
  '{{range $ii, $item := $tag.items}}{{if eq $item.image """${image_name}"'\
  $'"}}{{$is.metadata.name}}:{{$tag.tag}} at position {{$ii}} out of {{len $tag.items}}\n'\
  '{{end}}{{end}}{{end}}{{end}}'
myapp:v2 at position 4 out of 5
myapp:v2.1 at position 2 out of 2
myapp:v2.1-may-2016 at position 0 out of 1
```

デフォルトオプションが使用される場合、イメージは **myapp:v2.1-may-2016** タグの履歴の **0** の位置にあるためプルーニングされません。イメージがプルーニングの対象とみなされるようにするには、管理者は以下を実行する必要があります。

- **oc adm prune images** コマンドで **--keep-tag-revisions=0**を指定します。

  注意

  このアクションを実行すると、イメージが指定されたしきい値よりも新しいか、またはこれよりも新しいオブジェクトによって参照されていない限り、すべてのタグが基礎となるイメージと共にすべての namespace から削除されます。

- リビジョンのしきい値の下にあるすべての **istags**、つまり **myapp:v2.1** および **myapp:v2.1-may-2016** を削除します。

- 同じ **istag** にプッシュする新規ビルドを実行するか、または他のイメージをタグ付けしてイメージを履歴内でさらに移動させます。ただし、これは古いリリースタグの場合には常に適切な操作となる訳ではありません。

特定のイメージのビルド日時が名前の一部になっているタグは、その使用を避ける必要があります (イメージが未定義の期間保持される必要がある場合を除きます)。このようなタグは履歴内で 1 つのイメージのみに関連付けられる可能性があり、その場合にこれらをプルーニングできなくなります。

非セキュアなレジストリーに対するセキュアな接続の使用
oc adm prune images コマンドの出力で以下のようなメッセージが表示される場合、レジストリーのセキュリティーは保護されておらず、**oadm prune images** クライアントがセキュアな接続の使用を試行することを示しています。

```
error: error communicating with registry: Get https://172.30.30.30:5000/healthz: http: server gave HTTP response to HTTPS client
```

1. 推奨される解決法として、レジストリーのセキュリティーを保護することができます。そうしない場合は、**--force-insecure** をコマンドに追加して、クライアントに対して非セキュアな接続の使用を強制することができますが、これは推奨される方法ではありません。

セキュリティーが保護されたレジストリーに対する非セキュアな接続の使用
oadm prune images コマンドの出力に以下のエラーのいずれかが表示される場合、レジストリーのセキュリティー保護に使用されている認証局で署名された証明書が、接続の検証用に **oadm prune images** クライアントで使用されるものとは異なることを意味します。

```
error: error communicating with registry: Get http://172.30.30.30:5000/healthz: malformed HTTP response "\x15\x03\x01\x00\x02\x02"
error: error communicating with registry: [Get https://172.30.30.30:5000/healthz: x509: certificate signed by unknown authority, Get http://172.30.30.30:5000/healthz: malformed HTTP response "\x15\x03\x01\x00\x02\x02"]
```

デフォルトでは、ユーザーの接続ファイルに保存されている認証局データが使用されます。 これはマスター API との通信の場合も同様です。

--certificate-authority オプションを使用してコンテナーイメージレジストリーサーバーに適切な認証局を指定します。

正しくない認証局の使用
以下のエラーは、セキュリティーが保護されたコンテナーイメージレジストリーの証明書の署名に使用される認証局がクライアントで使用される認証局とは異なることを示しています。

```
error: error communicating with registry: Get https://172.30.30.30:5000/: x509: certificate signed by unknown authority
```

フラグ **--certificate-authority** を使用して適切な認証局を指定します。

回避策として、**--force-insecure** フラグを代わりに追加することもできます。ただし、これは推奨される方法ではありません。

追加リソース

- レジストリーへのアクセス
- レジストリーの公開

10.6. レジストリーのハードプルーニング

OpenShift Container レジストリーは、OpenShift Container Platform クラスターの etcd で参照されない Blob を蓄積します。基本的なイメージプルーニングの手順はこれらに対応しません。これらの Blob は **孤立した Blob** と呼ばれています。

孤立した Blob は以下のシナリオで発生する可能性があります。

- **oc delete image <sha256:image-id>** コマンドを使ってイメージを手動で削除すると、etcd のイメージのみが削除され、レジストリーのストレージからは削除されません。
- **docker** デーモンの障害によって生じるレジストリーへのプッシュにより、一部の Blob はアップロードされるものの、(最後のコンポーネントとしてアップロードされる) イメージマニフェスト はアップロードされません。固有のイメージ Blob すべてが孤立します。
- OpenShift Container Platform がクォータの制限によりイメージを拒否します。
- 標準のイメージプルーナーがイメージマニフェストを削除するが、関連する Blob を削除する前に中断されます。
- 対象の Blob を削除できないというレジストリープルーナーのバグにより、それらを参照するイメージオブジェクトは削除され、Blob は孤立します。

基本的なイメージプルーニングとは異なるレジストリーの **ハードプルーニング** により、クラスター管理者は孤立した Blob を削除することができます。OpenShift Container レジストリーのストレージ領域が不足している場合や、孤立した Blob があると思われる場合にはハードプルーニングを実行する必要があります。

これは何度も行う操作ではなく、多数の孤立した Blob が新たに作成されているという証拠がある場合にのみ実行する必要があります。または、(作成されるイメージの数によって異なりますが) 1 日 1 回などの定期的な間隔で標準のイメージプルーニングを実行することもできます。

手順

孤立した Blob をレジストリーからハードプルーニングするには、以下を実行します。

1. **ログイン**
   アクセストークンを持つユーザーとして、CLI でクラスターにログインします。
2. **基本的なイメージプルーニングの実行**
   基本的なイメージプルーニングにより、不要になった追加のイメージが削除されます。ハードプルーニングによってイメージが削除される訳ではありません。レジストリーストレージに保

存された Blob のみが削除されます。したがって、ハードプルーニングの実行前にこれを実行する必要があります。

3. **レジストリーの読み取り専用モードへの切り替え**
レジストリーが読み取り専用モードで実行されていない場合、プルーニングと同時に実行されているプッシュの結果は以下のいずれかになります。

   - 失敗する。孤立した Blob が新たに発生します。
   - 成功する。ただし、(参照される Blob の一部が削除されたため) イメージをプルできません。

   プッシュは、レジストリーが読み取り書き込みモードに戻されるまで成功しません。したがって、ハードプルーニングは注意してスケジューリングする必要があります。

   レジストリーを読み取り専用モードに切り換えるには、以下を実行します。

   a. 以下の環境変数を設定します。

   ```
   $ oc set env -n default \
       dc/docker-registry \
       'REGISTRY_STORAGE_MAINTENANCE_READONLY={"enabled":true}'
   ```

   b. デフォルトで、レジストリーは直前の手順が完了すると自動的に再デプロイします。再デプロイが完了するのを待機してから次に進んでください。ただし、これらのトリガーを無効にしている場合は、レジストリーを手動で再デプロイし、新規の環境変数が選択されるようにする必要があります。

   ```
   $ oc rollout -n default \
       latest dc/docker-registry
   ```

4. **system:image-pruner ロールの追加**
一部のリソースを一覧表示するには、レジストリーインスタンスの実行に使用するサービスアカウントに追加のパーミッションが必要になります。

   a. サービスアカウント名を取得します。

   ```
   $ service_account=$(oc get -n default \
       -o jsonpath=$'system:serviceaccount:{.metadata.namespace}:
   {.spec.template.spec.serviceAccountName}\n' \
       dc/docker-registry)
   ```

   b. **system:image-pruner** クラスターロールをサービスアカウントに追加します。

   ```
   $ oc adm policy add-cluster-role-to-user \
       system:image-pruner \
       ${service_account}
   ```

5. **(オプション) プルーナーのドライランモードでの実行**
削除される Blob の数を確認するには、ドライランモードでハードプルーナーを実行します。実際の変更は加えられません。

   ```
   $ oc -n default \
       exec -i -t "$(oc -n default get pods -l deploymentconfig=docker-registry \
   ```

```
    -o jsonpath=$'{.items[0].metadata.name}\n')" \
    -- /usr/bin/dockerregistry -prune=check
```

または、プルーニング候補の実際のパスを取得するには、ロギングレベルを上げます。

```
$ oc -n default \
    exec "$(oc -n default get pods -l deploymentconfig=docker-registry \
      -o jsonpath=$'{.items[0].metadata.name}\n')" \
    -- /bin/sh \
    -c 'REGISTRY_LOG_LEVEL=info /usr/bin/dockerregistry -prune=check'
```

出力サンプル (切り捨て済み)

```
$ oc exec docker-registry-3-vhndw \
    -- /bin/sh -c 'REGISTRY_LOG_LEVEL=info /usr/bin/dockerregistry -prune=check'

time="2017-06-22T11:50:25.066156047Z" level=info msg="start prune (dry-run mode)"
distribution_version="v2.4.1+unknown" kubernetes_version=v1.6.1+$Format:%h$
openshift_version=unknown
time="2017-06-22T11:50:25.092257421Z" level=info msg="Would delete blob:
sha256:00043a2a5e384f6b59ab17e2c3d3a3d0a7de01b2cabeb606243e468acc663fa5"
go.version=go1.7.5 instance.id=b097121c-a864-4e0c-ad6c-cc25f8fdf5a6
time="2017-06-22T11:50:25.092395621Z" level=info msg="Would delete blob:
sha256:0022d49612807cb348cabc562c072ef34d756adfe0100a61952cbcb87ee6578a"
go.version=go1.7.5 instance.id=b097121c-a864-4e0c-ad6c-cc25f8fdf5a6
time="2017-06-22T11:50:25.092492183Z" level=info msg="Would delete blob:
sha256:0029dd4228961086707e53b881e25eba0564fa80033fbbb2e27847a28d16a37c"
go.version=go1.7.5 instance.id=b097121c-a864-4e0c-ad6c-cc25f8fdf5a6
time="2017-06-22T11:50:26.673946639Z" level=info msg="Would delete blob:
sha256:ff7664dfc213d6cc60fd5c5f5bb00a7bf4a687e18e1df12d349a1d07b2cf7663"
go.version=go1.7.5 instance.id=b097121c-a864-4e0c-ad6c-cc25f8fdf5a6
time="2017-06-22T11:50:26.674024531Z" level=info msg="Would delete blob:
sha256:ff7a933178ccd931f4b5f40f9f19a65be5eeeec207e4fad2a5bafd28afbef57e"
go.version=go1.7.5 instance.id=b097121c-a864-4e0c-ad6c-cc25f8fdf5a6
time="2017-06-22T11:50:26.674675469Z" level=info msg="Would delete blob:
sha256:ff9b8956794b426cc80bb49a604a0b24a1553aae96b930c6919a6675db3d5e06"
go.version=go1.7.5 instance.id=b097121c-a864-4e0c-ad6c-cc25f8fdf5a6
...
Would delete 13374 blobs
Would free up 2.835 GiB of disk space
Use -prune=delete to actually delete the data
```

6. **ハードプルーニングの実行**
ハードプルーニングを実行するには、**docker-registry** Pod の実行中のインスタンスで以下のコマンドを実行します。

```
$ oc -n default \
    exec -i -t "$(oc -n default get pods -l deploymentconfig=docker-registry -o
jsonpath=$'{.items[0].metadata.name}\n')" \
    -- /usr/bin/dockerregistry -prune=delete
```

出力サンプル

```
$ oc exec docker-registry-3-vhndw \
```

```
    -- /usr/bin/dockerregistry -prune=delete

Deleted 13374 blobs
Freed up 2.835 GiB of disk space
```

7. **レジストリーを読み取り/書き込みモードに戻す**
   プルーニングの終了後は、以下を実行してレジストリーを読み取り/書き込みモードに戻すことができます。

```
$ oc set env -n default dc/docker-registry
REGISTRY_STORAGE_MAINTENANCE_READONLY-
```

10.7. 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。

追加リソース

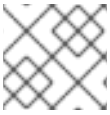
- ジョブを使用した Pod でのタスクの実行
- 複数のプロジェクト間のリソースクォータ
- RBAC の使用によるパーミッションの定義および適用

第11章 OPERATOR SDK

11.1. OPERATOR SDK の使用を開始する

以下では、Operator SDK の基本事項についての概要を説明し、単純な Go ベースの Memcached Operator のビルドおよびインストールからアップグレードまでのそのライフサイクル管理の例を使って、(OpenShift Container Platform などの) クラスター管理者の Kubernetes ベースのクラスターへのアクセスを持つ Operator の作成者を支援します。

これは、Operator SDK (**operator-sdk** CLI ツールおよび **controller-runtime** ライブラリー API) と Operator Lifecycle Manager (OLM) という 2 つの Operator Framework の重要な構成要素を使用して実行されます。



注記

OpenShift Container Platform 4 は Operator SDK v0.7.0 以降をサポートします。

11.1.1. Operator SDK のアーキテクチャー

Operator Framework は **Operator** という Kubernetes ネイティブアプリケーションを効果的かつ自動化された拡張性のある方法で管理するためのオープンソースツールキットです。Operator は、プロビジョニング、スケーリング、バックアップおよび復元などのクラウドサービスの自動化の利点を提供し、同時に Kubernetes が実行されるいずれの場所でも実行できます。

Operator により、Kubernetes の上部に複雑で、ステートフルなアプリケーションを管理することが容易になります。ただし、現時点で Operator の作成は、低レベルの API の使用、スケルトンコードの作成、モジュール化の欠如による重複の発生などの課題があるために困難になる場合があります。

Operator SDK は、以下を提供して Operator をより容易に作成できるように設計されたフレームワークです。

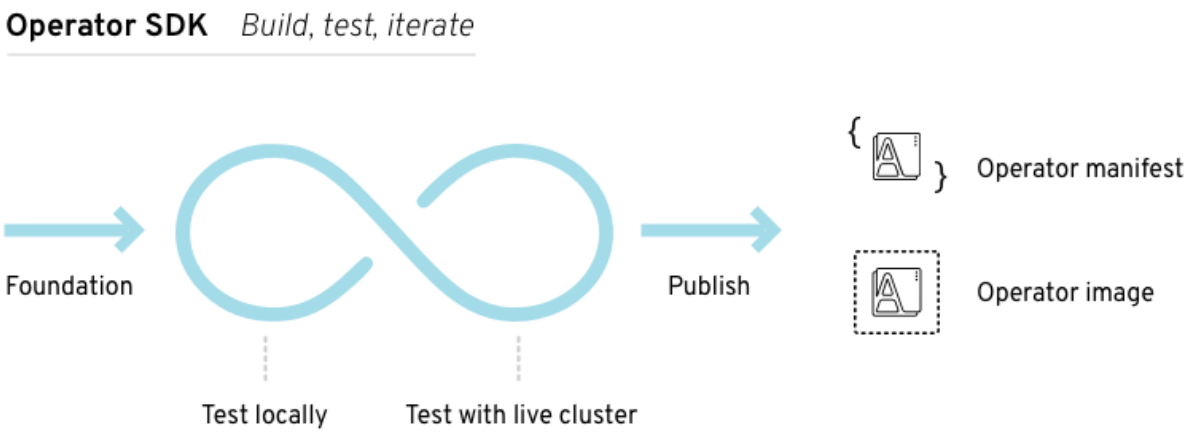
- 運用ロジックをより直感的に作成するための高レベルの API および抽象化
- 新規プロジェクトを迅速にブートストラップするためのスケルトンコードの作成およびコード生成ツール
- 共通する Operator ユースケースに対応する拡張機能

11.1.1.1. ワークフロー

Operator SDK は、新規 Operator を開発するために以下のワークフローを提供します。

1. Operator SDK コマンドラインインターフェース (CLI) を使用した新規 Operator プロジェクトの作成。
2. カスタムリソース定義 (CRD) を追加することによる新規リソース API の定義。
3. Operator SDK API を使用した監視対象リソースの指定。
4. 指定されたハンドラーでの Operator 調整 (reconciliation) ロジックの定義、およびリソースと対話するための Operator SDK API の使用。
5. Operator Deployment マニフェストをビルドし、生成するための Operator SDK CLI の使用。

図11.1 Operator SDK ワークフロー



高次元では、Operator SDK を使用する Operator は Operator の作成者が定義するハンドラーで監視対象のリソースについてのイベントを処理し、アプリケーションの状態を調整するための動作を実行します。

11.1.1.2. マネージャーファイル

Operator の主なプログラムは、**cmd/manager/main.go** のマネージャーファイルです。マネージャーは、**pkg/apis/** で定義されるすべてのカスタムリソース(CR) のスキームを自動的に登録し、**pkg/controller/** 下のすべてのコントローラーを実行します。

マネージャーは、すべてのコントローラーがリソースの監視に使用する namespace を制限できます。

```
mgr, err := manager.New(cfg, manager.Options{Namespace: namespace})
```

デフォルトでは、これは Operator が実行されている namespace です。すべての namespace を確認するには、namespace オプションのオプションを空のままにすることができます。

```
mgr, err := manager.New(cfg, manager.Options{Namespace: ""})
```

11.1.1.3. Prometheus Operator

Prometheus はオープンソースのシステムモニタリングおよびアラートツールキットです。Prometheus Operator は、 OpenShift Container Platform などの Kubernetes ベースのクラスターで実行される Prometheus クラスターを作成し、設定し、管理します。

ヘルパー関数は、デフォルトで Operator SDK に存在し、Prometheus Operator がデプロイされているクラスターで使用できるように生成された Go ベースの Operator にメトリクスを自動的にセットアップします。

11.1.2. Operator SDK CLI のインストール

Operator SDK には、開発者による新規 Operator プロジェクトの作成、ビルドおよびデプロイを支援をする CLI ツールが含まれます。ワークステーションに SDK CLI をインストールして、独自の Operator のオーサリングを開始することができます。

注記

以下では、ローカル Kubernetes クラスターとしての minikube v0.25.0+ とパブリックレジストリーの quay.io を使用します。

11.1.2.1. GitHub リリースからのインストール

GitHub のプロジェクトから SDK CLI の事前ビルドリリースのバイナリーをダウンロードし、インストールできます。

前提条件

- **docker** v17.03+
- OpenShift CLI (**oc**) v4.1+ (インストール済み)
- Kubernetes v1.11.3+ に基づくクラスターへのアクセス
- コンテナーレジストリーへのアクセス

手順

1. リリースバージョン変数を設定します。

```
RELEASE_VERSION=v0.8.0
```

2. リリースバイナリーをダウンロードします。
   - Linux の場合

```
$ curl -OJL https://github.com/operator-framework/operator-sdk/releases/download/${RELEASE_VERSION}/operator-sdk-${RELEASE_VERSION}-x86_64-linux-gnu
```

   - MacOS の場合

```
$ curl -OJL https://github.com/operator-framework/operator-sdk/releases/download/${RELEASE_VERSION}/operator-sdk-${RELEASE_VERSION}-x86_64-apple-darwin
```

3. ダウンロードしたリリースのバイナリーを確認します。
   a. 提供された ASC ファイルをダウンロードします。
      - Linux の場合

```
$ curl -OJL https://github.com/operator-framework/operator-sdk/releases/download/${RELEASE_VERSION}/operator-sdk-${RELEASE_VERSION}-x86_64-linux-gnu.asc
```

      - MacOS の場合

```
$ curl -OJL https://github.com/operator-framework/operator-sdk/releases/download/${RELEASE_VERSION}/operator-sdk-${RELEASE_VERSION}-x86_64-apple-darwin.asc
```

b. バイナリーと対応する ASC ファイルを同じディレクトリーに置き、以下のコマンドを実行してバイナリーを確認します。

- Linux の場合

```
$ gpg --verify operator-sdk-${RELEASE_VERSION}-x86_64-linux-gnu.asc
```

- MacOS の場合

```
$ gpg --verify operator-sdk-${RELEASE_VERSION}-x86_64-apple-darwin.asc
```

保守管理者の公開キーがワークステーションにない場合は、以下のエラーが出されます。

```
$ gpg --verify operator-sdk-${RELEASE_VERSION}-x86_64-apple-darwin.asc
$ gpg: assuming signed data in 'operator-sdk-${RELEASE_VERSION}-x86_64-apple-darwin'
$ gpg: Signature made Fri Apr  5 20:03:22 2019 CEST
$ gpg:                using RSA key <key_id> (1)
$ gpg: Can't check signature: No public key
```

(1) RSA キー文字列。

キーをダウンロードするには、以下のコマンドを実行し、 **<key_id>** を直前のコマンドの出力で提供された RSA キー文字列に置き換えます。

```
$ gpg [--keyserver keys.gnupg.net] --recv-key "<key_id>" (1)
```

(1) キーサーバーが設定されていない場合、これを **--keyserver** オプションで指定します。

4. リリースバイナリーを **PATH** にインストールします。

- Linux の場合

```
$ chmod +x operator-sdk-${RELEASE_VERSION}-x86_64-linux-gnu
$ sudo cp operator-sdk-${RELEASE_VERSION}-x86_64-linux-gnu
/usr/local/bin/operator-sdk
$ rm operator-sdk-${RELEASE_VERSION}-x86_64-linux-gnu
```

- MacOS の場合

```
$ chmod +x operator-sdk-${RELEASE_VERSION}-x86_64-apple-darwin
$ sudo cp operator-sdk-${RELEASE_VERSION}-x86_64-apple-darwin
/usr/local/bin/operator-sdk
$ rm operator-sdk-${RELEASE_VERSION}-x86_64-apple-darwin
```

5. CLI ツールが正しくインストールされていることを確認します。

```
$ operator-sdk version
```

11.1.2.2. Homebrew からのインストール

Homebrew を使用して SDK CLI をインストールできます。

前提条件

- Homebrew
- **docker** v17.03+
- OpenShift CLI (**oc**) v4.1+ (インストール済み)
- Kubernetes v1.11.3+ に基づくクラスターへのアクセス
- コンテナーレジストリーへのアクセス

手順

1. **brew** コマンドを使用して SDK CLI をインストールします。

```
$ brew install operator-sdk
```

2. CLI ツールが正しくインストールされていることを確認します。

```
$ operator-sdk version
```

11.1.2.3. ソースを使用したコンパイルおよびインストール

Operator SDK ソースコードを取得して、SDK CLI をコンパイルし、インストールできます。

前提条件

- dep v0.5.0+
- Git
- Go v1.10+
- **docker** v17.03+
- OpenShift CLI (**oc**) v4.1+ (インストール済み)
- Kubernetes v1.11.3+ に基づくクラスターへのアクセス
- コンテナーレジストリーへのアクセス

手順

1. **operator-sdk** リポジトリーのクローンを作成します。

```
$ mkdir -p $GOPATH/src/github.com/operator-framework
$ cd $GOPATH/src/github.com/operator-framework
$ git clone https://github.com/operator-framework/operator-sdk
$ cd operator-sdk
```

2. 必要なリリースブランチをチェックアウトします。

```
$ git checkout master
```

3. SDK CLI ツールをコンパイルし、インストールします。

```
$ make dep
$ make install
```

これにより、**$GOPATH/bin** に CLI バイナリー **operator-sdk** がインストールされます。

4. CLI ツールが正しくインストールされていることを確認します。

```
$ operator-sdk version
```

11.1.3. Operator SDK を使用した Go ベースの Memcached Operator のビルド

Operator SDK は、詳細なアプリケーション固有の運用上の知識を必要とする可能性のあるプロセスである、Kubernetes ネイティブアプリケーションのビルドを容易にします。SDK はこの障壁を低くするだけでなく、メータリングやモニタリングなどの数多くの一般的な管理機能に必要なスケルトンコードの量を減らします。

この手順では、SDK によって提供されるツールおよびライブラリーを使用して単純な Memcached Operator をビルドする例を示します。

前提条件

- 開発ワークステーションにインストールされる Operator SDK CLI
- OpenShift Container Platform 4.1 などの、Kubernetes ベースのクラスター (v1.8 以上の **apps/v1beta2** API グループをサポートするもの) にインストールされる Operator Lifecycle Manager (OLM)
- **cluster-admin** パーミッションのあるアカウントを使用したクラスターへのアクセス
- OpenShift CLI (**oc**) v4.1+ (インストール済み)

手順

1. **新規プロジェクトを作成します。**
CLI を使用して新規 **memcached-operator** プロジェクトを作成します。

```
$ mkdir -p $GOPATH/src/github.com/example-inc/
$ cd $GOPATH/src/github.com/example-inc/
$ operator-sdk new memcached-operator --dep-manager dep
$ cd memcached-operator
```

2. **新規カスタムリソース定義 (CRD) を追加します。**
   a. **APIVersion** を **cache.example.com/v1apha1** に設定し、**Kind** を **Memcached** に設定した状態で、CLI を使用して **Memcached** という新規 CRD API を追加します。

```
$ operator-sdk add api \
    --api-version=cache.example.com/v1alpha1 \
    --kind=Memcached
```

これにより、**pkg/apis/cache/v1alpha1/** の下で Memcached resource API のスキャフォールディングが実行されます。

b. **pkg/apis/cache/v1alpha1/memcached_types.go** ファイルで、 **Memcached** カスタムリソース (CR) の仕様およびステータスを変更します。

```
type MemcachedSpec struct {
 // Size is the size of the memcached deployment
 Size int32 `json:"size"`
}
type MemcachedStatus struct {
 // Nodes are the names of the memcached pods
 Nodes []string `json:"nodes"`
}
```

c. ***_types.go** ファイルを変更後は、以下のコマンドを常に実行し、該当するリソースタイプ用に生成されたコードを更新します。

```
$ operator-sdk generate k8s
```

3. **新規コントローラーを追加します。**

a. 新規コントローラーをプロジェクトに追加し、 Memcached リソースを確認し、調整します。

```
$ operator-sdk add controller \
    --api-version=cache.example.com/v1alpha1 \
    --kind=Memcached
```

これにより、**pkg/controller/memcached/** の下で新規コントローラー実装のスキャフォールディングが実行されます。

b. この例では、生成されたコントローラーファイル **pkg/controller/memcached/memcached_controller.go** を実装例に置き換えます。
コントローラーのサンプルは、それぞれの **Memcached** CR について以下の調整 (reconciliation) ロジックを実行します。

- Memcached Deployment を作成します (ない場合)。
- Deployment のサイズが、 **Memcached** CR 仕様で指定されるのと同じであることを確認します。
- **Memcached** CR ステータスを Memcached Pod の名前で置き換えます。

次の 2 つのサブステップでは、コントローラーがリソースを監視する方法および調整ループがトリガーされる方法を検査します。これらの手順を省略し、直接 Operator のビルドおよび実行に進むことができます。

c. **pkg/controller/memcached/memcached_controller.go** ファイルでコントローラーの実装を検査し、コントローラーのリソースの監視方法を確認します。
最初の監視は、プライマリーソースとしての Memcached タイプに対して実行します。そ

れぞれの Add、Update、または Delete イベントについて、reconcile ループに Memcached オブジェクトの reconcile **Request** (**<namespace>:<name>** キー) が送られます。

```
err := c.Watch(
  &source.Kind{Type: &cachev1alpha1.Memcached{}},
&handler.EnqueueRequestForObject{})
```

次の監視は、Deployment に対して実行されますが、イベントハンドラーは各イベントを、Deployment のオーナーの reconcile **Request** にマップします。この場合、これは Deployment が作成された Memcached オブジェクトです。これにより、コントローラーは Deployment をセカンダリーリソースとして監視できます。

```
err := c.Watch(&source.Kind{Type: &appsv1.Deployment{}},
&handler.EnqueueRequestForOwner{
  IsController: true,
  OwnerType:    &cachev1alpha1.Memcached{},
 })
```

d. すべてのコントローラーには、reconcile ループを実装する **Reconcile()** メソッドのある Reconciler オブジェクトがあります。この reconcile ループには、キャッシュからプライマリーリソースオブジェクトの Memcached を検索するために使用される **<namespace>:<name>** キーである **Request** 引数が渡されます。

```
func (r *ReconcileMemcached) Reconcile(request reconcile.Request) (reconcile.Result,
error) {
  // Lookup the Memcached instance for this reconcile request
  memcached := &cachev1alpha1.Memcached{}
  err := r.client.Get(context.TODO(), request.NamespacedName, memcached)
  ...
}
```

Reconcile() の戻り値に応じて、reconcile **Request** は再度キューに入れられ、ループが再びトリガーされる可能性があります。

```
// Reconcile successful - don't requeue
return reconcile.Result{}, nil
// Reconcile failed due to error - requeue
return reconcile.Result{}, err
// Requeue for any reason other than error
return reconcile.Result{Requeue: true}, nil
```

4. **Operator をビルドし、実行します** 。

a. Operator の実行前に、CRD を Kubernetes API サーバーに再度登録する必要があります。

```
$ oc create \
    -f deploy/crds/cache_v1alpha1_memcached_crd.yaml
```

b. CRD の登録後に、Operator を実行するための 2 つのオプションを選択できます。

- Kubernetes クラスター内の Deployment を使用
- クラスター内の Go プログラムを使用

以下の方法のいずれかを選択します。

i. **オプション A:** クラスター内の Deployment として実行する。

A. **memcached-operator** イメージをビルドし、これをレジストリーにプッシュします。

```
$ operator-sdk build quay.io/example/memcached-operator:v0.0.1
```

B. Deployment マニフェストは **deploy/operator.yaml** に生成されます。デフォルトはプレースホルダーでしかないため、以下のように Deployment イメージを更新します。

```
$ sed -i 's|REPLACE_IMAGE|quay.io/example/memcached-operator:v0.0.1|g'
deploy/operator.yaml
```

C. 次のステップについての quay.io にアカウントがあることを確認するか、または優先しているコンテナーレジストリーで置き換えます。レジストリーには、memcached-operator という名前の **新規パブリックイメージ**リポジトリーを作成します。

D. イメージをレジストリーにプッシュします。

```
$ docker push quay.io/example/memcached-operator:v0.0.1
```

E. RBAC をセットアップし、**memcached-operator** をデプロイします。

```
$ oc create -f deploy/role.yaml
$ oc create -f deploy/role_binding.yaml
$ oc create -f deploy/service_account.yaml
$ oc create -f deploy/operator.yaml
```

F. **memcached-operator** が設定されており、稼働していることを確認します。

```
$ oc get deployment
NAME                 DESIRED   CURRENT   UP-TO-DATE   AVAILABLE   AGE
memcached-operator   1         1         1            1           1m
```

ii. **オプション B:** クラスター外でローカルに実行する。
この方法は、迅速にデプロイメントおよびテストを実行するための開発サイクルで優先される方法です。

$HOME/.kube/config にあるデフォルトの Kubernetes 設定ファイルを使用して Operator をローカルで実行します。

```
$ operator-sdk up local --namespace=default
```

フラグ **--kubeconfig=<path/to/kubeconfig>** を使用して特定の **kubeconfig** を使用できます。

5. Memcached CR を作成して、**Operator が Memcached アプリケーションをデプロイできることを確認します。**

a. **deploy/crds/cache_v1alpha1_memcached_cr.yaml** で生成された **Memcached** CR のサンプルを作成します。

```
$ cat deploy/crds/cache_v1alpha1_memcached_cr.yaml
apiVersion: "cache.example.com/v1alpha1"
kind: "Memcached"
metadata:
  name: "example-memcached"
spec:
  size: 3

$ oc apply -f deploy/crds/cache_v1alpha1_memcached_cr.yaml
```

b. **memcached-operator** が CR の Deployment を作成できることを確認します。

```
$ oc get deployment
NAME                 DESIRED   CURRENT   UP-TO-DATE   AVAILABLE   AGE
memcached-operator       1         1         1            1          2m
example-memcached        3         3         3            3          1m
```

c. ステータスが **memcached** Pod 名で更新されていることを確認するために、Pod および CR ステータスをチェックします。

```
$ oc get pods
NAME                                  READY     STATUS    RESTARTS   AGE
example-memcached-6fd7c98d8-7dqdr     1/1       Running   0          1m
example-memcached-6fd7c98d8-g5k7v     1/1       Running   0          1m
example-memcached-6fd7c98d8-m7vn7     1/1       Running   0          1m
memcached-operator-7cc7cfdf86-vvjqk   1/1       Running   0          2m

$ oc get memcached/example-memcached -o yaml
apiVersion: cache.example.com/v1alpha1
kind: Memcached
metadata:
  clusterName: ""
  creationTimestamp: 2018-03-31T22:51:08Z
  generation: 0
  name: example-memcached
  namespace: default
  resourceVersion: "245453"
  selfLink:
/apis/cache.example.com/v1alpha1/namespaces/default/memcacheds/example-
memcached
  uid: 0026cc97-3536-11e8-bd83-0800274106a1
spec:
  size: 3
status:
  nodes:
  - example-memcached-6fd7c98d8-7dqdr
  - example-memcached-6fd7c98d8-g5k7v
  - example-memcached-6fd7c98d8-m7vn7
```

6. デプロイメントのサイズを更新し、**Operator がデプロイ済みの Memcached アプリケーションを管理できることを確認します** 。

a. **memcached** CR の **spec.size** フィールドを **3** から **4** に変更します。

```
$ cat deploy/crds/cache_v1alpha1_memcached_cr.yaml
apiVersion: "cache.example.com/v1alpha1"
kind: "Memcached"
metadata:
  name: "example-memcached"
spec:
  size: 4
```

b. 変更を適用します。

```
$ oc apply -f deploy/crds/cache_v1alpha1_memcached_cr.yaml
```

c. Operator が Deployment サイズを変更することを確認します。

```
$ oc get deployment
NAME                DESIRED   CURRENT   UP-TO-DATE   AVAILABLE   AGE
example-memcached   4         4         4            4           5m
```

7. **リソースをクリーンアップします。**

```
$ oc delete -f deploy/crds/cache_v1alpha1_memcached_cr.yaml
$ oc delete -f deploy/crds/cache_v1alpha1_memcached_crd.yaml
$ oc delete -f deploy/operator.yaml
$ oc delete -f deploy/role.yaml
$ oc delete -f deploy/role_binding.yaml
$ oc delete -f deploy/service_account.yaml
```

11.1.4. Operator Lifecycle Manager を使用した Memcached Operator の管理

直前のセクションでは、Operator を手動で実行することについて説明しました。次のセクションでは、実稼働環境で実行される Operator のより堅牢なデプロイメントモデルを可能にする Operator Lifecycle Manager (OLM) の使用方法について説明します。

OLM は、Kubernetes クラスターで Operator (およびそれらの関連サービス) をインストールし、更新し、通常はそれらすべての Operator のライフサイクルを管理するのに役立ちます。これは、Kubernetes 拡張として実行され、追加のツールなしにすべてのライフサイクル管理機能について **oc** を使用できます。

前提条件

- OLM が (**apps/v1beta2** API グループをサポートする v1.8 以上のバージョンの) Kubernetes ベースのクラスターにインストールされていること。 たとえば、 OpenShift Container Platform 4.1 Preview OLM が有効にされていること。
- Memcached Operator がビルドされていること。

手順

1. **Operator マニフェストを生成します。**
   Operator マニフェストは、アプリケーションを表示し、作成し、管理する方法について説明します (この場合は Memcached)。これは **ClusterServiceVersion** (CSV) オブジェクトで定義され、OLM が機能するために必要です。

以下のコマンドを使用して CSV マニフェストを生成することができます。

```
$ operator-sdk olm-catalog gen-csv --csv-version 0.0.1
```

注記

このコマンドは、Memcached Operator のビルド時に作成された **memcached-operator/** ディレクトリーから実行されます。

本書の目的を考慮し、次のいくつかのステップにおいても引き続きこの事前に定義されたマニフェストファイルを使用します。このマニフェスト内のイメージフィールドを変更して、直前のステップでビルドしたイメージを反映させることができますが、これは不要です。

注記

マニフェストファイルの手動による定義についての詳細は、「Building a CSV for the Operator Framework」を参照してください。

2. **Operator をデプロイします。**

   a. Operator がターゲットとする namespace を指定する OperatorGroup を作成します。以下の OperatorGroup を、CSV を作成する namespace に作成します。この例では、 **default** namespace が使用されます。

```
apiVersion: operators.coreos.com/v1
kind: OperatorGroup
metadata:
  name: memcached-operator-group
  namespace: default
spec:
  targetNamespaces:
  - default
```

   b. Operator の CSV マニフェストをクラスターの指定された namespace に適用します。

```
$ curl -Lo memcachedoperator.0.0.1.csv.yaml
https://raw.githubusercontent.com/operator-framework/getting-
started/master/memcachedoperator.0.0.1.csv.yaml
$ oc apply -f memcachedoperator.0.0.1.csv.yaml
$ oc get csv memcachedoperator.v0.0.1 -n default -o json | jq '.status'
```

   このマニフェストを適用する際に、クラスターはマニフェストで指定された要件を満たしていないためにすぐに更新を実行しません。

   c. リソースパーミッションを Operator に付与するためにロール、ロールバインディング、およびサービスアカウントを作成し、Operator が管理する Memcached タイプを作成するためにカスタムリソース定義 (CRD、Custom Resource Definition) を作成します。

```
$ oc create -f deploy/crds/cache_v1alpha1_memcached_crd.yaml
$ oc create -f deploy/service_account.yaml
$ oc create -f deploy/role.yaml
$ oc create -f deploy/role_binding.yaml
```

注記

これらのファイルは、Memcached Operator のビルド時に Operator SDK よって **deploy**/ ディレクトリーに生成されています。

マニフェストの適用時に OLM は Operator を特定の namespace に作成するため、管理者は、Operator をインストールできるユーザーを制限するためのネイティブの Kubernetes RBAC パーミッションモデルを利用できます。

3. **アプリケーションインスタンスを作成します**。
Memcached Operator が **default** namespace で実行されるようになります。ユーザーは **CustomResources** のインスタンス経由で Operator と対話します。この場合、リソースには **Memcached** の種類が設定されます。ネイティブの Kubernetes RBAC は **CustomResources** に適用され、管理者には各 Operater と対話できるユーザーへの制御が提供されます。

この namespace で Memcached のインスタンスを作成することにより、Operator で管理される memcached サーバーを実行する Pod をインスタンス化するために Memcached Operator がトリガーされます。**CustomResources** をより多く作成すると、Memcached のより多くの固有なインスタンスがこの namespace で実行されている Memcached Operator によって管理されます。

```
$ cat <<EOF | oc apply -f -
apiVersion: "cache.example.com/v1alpha1"
kind: "Memcached"
metadata:
  name: "memcached-for-wordpress"
spec:
  size: 1
EOF

$ cat <<EOF | oc apply -f -
apiVersion: "cache.example.com/v1alpha1"
kind: "Memcached"
metadata:
  name: "memcached-for-drupal"
spec:
  size: 1
EOF

$ oc get Memcached
NAME                    AGE
memcached-for-drupal      22s
memcached-for-wordpress   27s

$ oc get pods
NAME                                  READY     STATUS    RESTARTS   AGE
memcached-app-operator-66b5777b79-pnsfj    1/1       Running   0          14m
memcached-for-drupal-5476487c46-qbd66      1/1       Running   0          3s
memcached-for-wordpress-65b75fd8c9-7b9x7   1/1       Running   0          8s
```

4. **アプリケーションを更新します**。
新規 Operator マニフェストを、古い Operator マニフェストを参照する **replaces** フィールドを使って作成し、更新を Operator に手動で適用します。OLM は、古い Operator で管理されているすべてのリソースの所有権が、いずれのプログラムも停止することなく新規 Operator に

移行できるようにします。新規バージョンの Operator 下で実行するリソースのアップグレードに必要なデータ移行を実行するかどうかは Operator によって異なります。

以下のコマンドは、新規バージョンの Operator を使用して新規 Operator マニフェストファイルを適用する方法を示し、Pod が実行状態であることを示します。

```
$ curl -Lo memcachedoperator.0.0.2.csv.yaml https://raw.githubusercontent.com/operator-framework/getting-started/master/memcachedoperator.0.0.2.csv.yaml
$ oc apply -f memcachedoperator.0.0.2.csv.yaml
$ oc get pods
NAME                                     READY     STATUS    RESTARTS   AGE
memcached-app-operator-66b5777b79-pnsfj    1/1       Running   0          3s
memcached-for-drupal-5476487c46-qbd66      1/1       Running   0          14m
memcached-for-wordpress-65b75fd8c9-7b9x7   1/1       Running   0          14m
```

11.1.5. 追加リソース

- Operator SDK によって作成されるプロジェクトディレクトリー構造についての詳細は、「Appendices」を参照してください。
- Operator Development Guide for Red Hat Partners

11.2. ANSIBLE ベース OPERATOR の作成

以下では、Operator SDK における Ansible サポートについての概要を説明し、Operator の作成者に、Ansible Playbook およびモジュールを使用する **operator-sdk** CLI ツールを使って Ansible ベースの Operator をビルドし、実行するサンプルを示します。

11.2.1. Operator SDK における Ansible サポート

Operator Framework は **Operator** という Kubernetes ネイティブアプリケーションを効果的かつ自動化された拡張性のある方法で管理するためのオープンソースツールキットです。このフレームワークには Operator SDK が含まれ、これは Kubernetes API の複雑性を把握していなくても、それぞれの専門知識に基づいて Operator のブートストラップおよびビルドを実行できるように開発者を支援します。

Operator プロジェクトを生成するための Operator SDK のオプションの 1 つに、Go コードを作成することなしに Kubernetes リソースを統一されたアプリケーションとしてデプロイするために既存の Ansible Playbook およびモジュールを使用できるオプションがあります。

11.2.1.1. カスタムリソースファイル

Operator は Kubernetes の拡張メカニズムであるカスタムリソース定義 (CRD) を使用するため、カスタムリソース (CR) は、組み込み済みのネイティブ Kubernetes オブジェクトのように表示され、機能します。

CR ファイル形式は Kubernetes リソースファイルです。オブジェクトには、必須およびオプションフィールドが含まれます。

表11.1 カスタムリソースフィールド

フィールド	説明
apiVersion	作成される CR のバージョン。

フィールド	説明
kind	作成される CR の種類。
metadata	作成される Kubernetes 固有のメタデータ。
spec (オプション)	Ansible に渡される変数のキーと値の一覧。このフィールドは、デフォルトでは空です。
status	オブジェクトの現在の状態の概要を示します。Ansible ベースの Operator の場合、**status** サブリソース はデフォルトで CRD について有効にされ、**k8s_status** Ansible モジュールによって管理されます。 これには、CR の **status** に対する **condition** 情報が含まれます。
annotations	CR に付加する Kubernetes 固有のアノテーション。

CR アノテーションの以下の一覧は Operator の動作を変更します。

表11.2 Ansible ベースの Operator アノテーション

アノテーション	説明
ansible.operator-sdk/reconcile-period	CR の調整間隔を指定します。この値は標準的な Golang パッケージ **time** を使用して解析されます。とくに、**ParseDuration** は、**s** のデフォルトサフィックスを適用し、秒単位で値を指定します。

Ansible ベースの Operator アノテーションの例

```
apiVersion: "foo.example.com/v1alpha1"
kind: "Foo"
metadata:
  name: "example"
annotations:
  ansible.operator-sdk/reconcile-period: "30s"
```

11.2.1.2. 監視ファイル

監視ファイルには、**Group**、**Version**、および **Kind** などによって特定される、カスタムリソース (CR) から Ansible ロールまたは Playbook へのマッピングの一覧が含まれます。Operator はこのマッピングファイルが事前に定義された場所の **/opt/ansible/watches.yaml** にあることを予想します。

表11.3 監視ファイルのマッピング

フィールド	説明
group	監視する CR のグループ。
version	監視する CR のバージョン。

フィールド	説明
kind	監視する CR の種類。
role (デフォルト)	コンテナーに追加される Ansible ロールへのパスです。たとえば、 **roles** ディレクトリーが **/opt/ansible/roles/** にあり、ロールの名前が **busybox** の場合、この値は **/opt/ansible/roles/busybox** になります。このフィールドは **playbook** フィールドと相互に排他的です。
playbook	コンテナーに追加される Ansible Playbook へのパスです。この Playbook は単純にロールを呼び出す方法になります。このフィールドは **role** フィールドと相互に排他的です。
reconcilePeriod (オプション)	ロールまたは Playbook が特定の CR について実行される調整 期間および頻度。
manageStatus (オプション)	**true** (デフォルト) に設定されると、Operator は CR のステータスを汎用的に管理します。**false** に設定されると、指定されたロール、または別のコントローラーの Playbook により、CR のステータスは他の場所で管理されます。

監視ファイルの例

```
- version: v1alpha1 (1)
  group: foo.example.com
  kind: Foo
  role: /opt/ansible/roles/Foo

- version: v1alpha1 (2)
  group: bar.example.com
  kind: Bar
  playbook: /opt/ansible/playbook.yml

- version: v1alpha1 (3)
  group: baz.example.com
  kind: Baz
  playbook: /opt/ansible/baz.yml
  reconcilePeriod: 0
  manageStatus: false
```

1. **Foo** の **Foo** ロールへの単純なマッピング例。
2. **Bar** の Playbook への単純なマッピング例。
3. **Baz** の種類についてのより複雑な例。Playbook での CR ステータスを再度キューに入れるタスクまたはその管理を無効にします。

11.2.1.2.1. 高度なオプション

高度な機能は、それらを GVK (グループ、バージョン、および種類) ごとに監視ファイルに追加して有

高度な機能は、それらを GVK (グループ、バージョン、および種類) ごとに監視ファイルに追加して有効にできます。それらは **group**、**version**、**kind** および **playbook** または **role** フィールドの下に移行できます。

一部の機能は、カスタムリソース (CR) のアノテーションを使用してリソースごとに上書きできます。オーバーライドできるオプションには、以下に指定されるアノテーションが含まれます。

表11.4 高度な監視対象ファイルオプション

機能	YAML キー	説明	上書きのアノテーション	デフォルト値
調整期間	**reconcilePeriod**	特定の CR についての調整実行の間隔。	**ansbile.operator-sdk/reconcile-period**	**1m**
ステータスの管理	**manageStatus**	Operator は各 CR の **status** セクションの **conditions** セクションを管理できます。		**true**
依存するリソースの監視	**watchDependentResources**	Operator は Ansible によって作成されるリソースを動的に監視できます。		**true**
クラスタースコープのリソースの監視	**watchClusterScopedResources**	Operator は Ansible によって作成されるクラスタースコープのリソースを監視できます。		**false**
最大 Runner アーティファクト	**maxRunnerArtifacts**	Ansible Runner が各リソースについて Operator コンテナーに保持するアーティファクトディレクトリーの数を管理します。	**ansible.operator-sdk/max-runner-artifacts**	**20**

高度なオプションを含む監視ファイルの例

```
- version: v1alpha1
  group: app.example.com
  kind: AppService
  playbook: /opt/ansible/playbook.yml
  maxRunnerArtifacts: 30
  reconcilePeriod: 5s
  manageStatus: False
  watchDependentResources: False
```

11.2.1.3. Ansible に送信される追加変数

追加の変数を Ansible に送信し、Operator で管理できます。カスタマーリソース (CR) の **spec** セクションでは追加変数としてキーと値のペアを渡します。これは、**ansible-playbook** コマンドに渡される追加変数と同等です。

また Operator は、CR の名前および CR の namespace についての **meta** フィールドの下に追加の変数を渡します。

以下は CR の例になります。

```
apiVersion: "app.example.com/v1alpha1"
kind: "Database"
metadata:
  name: "example"
spec:
  message:"Hello world 2"
  newParameter: "newParam"
```

追加変数として Ansible に渡される構造は以下のとおりです。

```
{ "meta": {
        "name": "<cr_name>",
        "namespace": "<cr_namespace>",
  },
  "message": "Hello world 2",
  "new_parameter": "newParam",
  "_app_example_com_database": {
     <full_crd>
   },
}
```

message および **newParameter** フィールドは追加変数として上部に設定され、**meta** は Operator に定義されるように CR の関連メタデータを提供します。**meta** フィールドは、Ansible のドット表記などを使用してアクセスできます。

```
- debug:
    msg: "name: {{ meta.name }}, {{ meta.namespace }}"
```

11.2.1.4. Ansible Runner ディレクトリー

Ansible Runner はコンテナーに Ansible 実行についての情報を維持します。これは **/tmp/ansible-operator/runner/<group>/<version>/<kind>/<namespace>/<name>** に置かれます。

追加リソース

- **runner** ディレクトリーについての詳細は、Ansible Runner ドキュメントを参照してください。

11.2.2. Operator SDK CLI のインストール

Operator SDK には、開発者による新規 Operator プロジェクトの作成、ビルドおよびデプロイを支援をする CLI ツールが含まれます。ワークステーションに SDK CLI をインストールして、独自の Operator のオーサリングを開始することができます。

注記

以下では、ローカル Kubernetes クラスターとしての minikube v0.25.0+ とパブリックレジストリーの quay.io を使用します。

11.2.2.1. GitHub リリースからのインストール

GitHub のプロジェクトから SDK CLI の事前ビルドリリースのバイナリーをダウンロードし、インストールできます。

前提条件

- **docker** v17.03+
- OpenShift CLI (**oc**) v4.1+ (インストール済み)
- Kubernetes v1.11.3+ に基づくクラスターへのアクセス
- コンテナーレジストリーへのアクセス

手順

1. リリースバージョン変数を設定します。

```
RELEASE_VERSION=v0.8.0
```

2. リリースバイナリーをダウンロードします。
   - Linux の場合

```
$ curl -OJL https://github.com/operator-framework/operator-sdk/releases/download/${RELEASE_VERSION}/operator-sdk-${RELEASE_VERSION}-x86_64-linux-gnu
```

   - MacOS の場合

```
$ curl -OJL https://github.com/operator-framework/operator-sdk/releases/download/${RELEASE_VERSION}/operator-sdk-${RELEASE_VERSION}-x86_64-apple-darwin
```

3. ダウンロードしたリリースのバイナリーを確認します。
   a. 提供された ASC ファイルをダウンロードします。
      - Linux の場合

```
$ curl -OJL https://github.com/operator-framework/operator-sdk/releases/download/${RELEASE_VERSION}/operator-sdk-${RELEASE_VERSION}-x86_64-linux-gnu.asc
```

      - MacOS の場合

```
$ curl -OJL https://github.com/operator-framework/operator-sdk/releases/download/${RELEASE_VERSION}/operator-sdk-${RELEASE_VERSION}-x86_64-apple-darwin.asc
```

   b. バイナリーと対応する ASC ファイルを同じディレクトリーに置き、以下のコマンドを実行してバイナリーを確認します。
      - Linux の場合

```
$ gpg --verify operator-sdk-${RELEASE_VERSION}-x86_64-linux-gnu.asc
```

- MacOS の場合

```
$ gpg --verify operator-sdk-${RELEASE_VERSION}-x86_64-apple-darwin.asc
```

保守管理者の公開キーがワークステーションにない場合は、以下のエラーが出されます。

```
$ gpg --verify operator-sdk-${RELEASE_VERSION}-x86_64-apple-darwin.asc
$ gpg: assuming signed data in 'operator-sdk-${RELEASE_VERSION}-x86_64-apple-
darwin'
$ gpg: Signature made Fri Apr  5 20:03:22 2019 CEST
$ gpg:                using RSA key <key_id> 1
$ gpg: Can't check signature: No public key
```

1 RSA キー文字列。

キーをダウンロードするには、以下のコマンドを実行し、 **<key_id>** を直前のコマンドの出力で提供された RSA キー文字列に置き換えます。

```
$ gpg [--keyserver keys.gnupg.net] --recv-key "<key_id>" 1
```

1 キーサーバーが設定されていない場合、これを **--keyserver** オプションで指定します。

4. リリースバイナリーを **PATH** にインストールします。

- Linux の場合

```
$ chmod +x operator-sdk-${RELEASE_VERSION}-x86_64-linux-gnu
$ sudo cp operator-sdk-${RELEASE_VERSION}-x86_64-linux-gnu
/usr/local/bin/operator-sdk
$ rm operator-sdk-${RELEASE_VERSION}-x86_64-linux-gnu
```

- MacOS の場合

```
$ chmod +x operator-sdk-${RELEASE_VERSION}-x86_64-apple-darwin
$ sudo cp operator-sdk-${RELEASE_VERSION}-x86_64-apple-darwin
/usr/local/bin/operator-sdk
$ rm operator-sdk-${RELEASE_VERSION}-x86_64-apple-darwin
```

5. CLI ツールが正しくインストールされていることを確認します。

```
$ operator-sdk version
```

11.2.2.2. Homebrew からのインストール

Homebrew を使用して SDK CLI をインストールできます。

前提条件

- Homebrew
- **docker** v17.03+
- OpenShift CLI (**oc**) v4.1+ (インストール済み)
- Kubernetes v1.11.3+ に基づくクラスターへのアクセス
- コンテナーレジストリーへのアクセス

手順

1. **brew** コマンドを使用して SDK CLI をインストールします。

```
$ brew install operator-sdk
```

2. CLI ツールが正しくインストールされていることを確認します。

```
$ operator-sdk version
```

11.2.2.3. ソースを使用したコンパイルおよびインストール

Operator SDK ソースコードを取得して、SDK CLI をコンパイルし、インストールできます。

前提条件

- dep v0.5.0+
- Git
- Go v1.10+
- **docker** v17.03+
- OpenShift CLI (**oc**) v4.1+ (インストール済み)
- Kubernetes v1.11.3+ に基づくクラスターへのアクセス
- コンテナーレジストリーへのアクセス

手順

1. **operator-sdk** リポジトリーのクローンを作成します。

```
$ mkdir -p $GOPATH/src/github.com/operator-framework
$ cd $GOPATH/src/github.com/operator-framework
$ git clone https://github.com/operator-framework/operator-sdk
$ cd operator-sdk
```

2. 必要なリリースブランチをチェックアウトします。

```
$ git checkout master
```

3. SDK CLI ツールをコンパイルし、インストールします。

```
$ make dep
$ make install
```

これにより、**$GOPATH/bin** に CLI バイナリー **operator-sdk** がインストールされます。

4. CLI ツールが正しくインストールされていることを確認します。

```
$ operator-sdk version
```

11.2.3. Operator SDK を使用した Ansible ベースの Operator のビルド

以下の手順では、Operator SDK が提供するツールおよびライブラリーを使用した Ansible Playbook がサポートする単純な Memcached Operator のビルドの例について説明します。

前提条件

- 開発ワークステーションにインストールされる Operator SDK CLI
- **cluster-admin** パーミッションを持つアカウントを使用した Kubernetes ベースのクラスターr v1.11.3+ (OpenShift Container Platform 4.1 など) へのアクセス
- OpenShift CLI (**oc**) v4.1+ (インストール済み)
- **ansible** v2.6.0+
- **ansible-runner** v1.1.0+
- **ansible-runner-http** v1.0.0+

手順

1. **operator-sdk new** コマンドを使用して、namespace スコープまたはクラスタースコープのいずれかで **新規 Operator プロジェクトを作成します**。以下のいずれかを選択します。

   a. **namespace スコープ Operator** (デフォルト) は単一 namespace でリソースを監視し、管理します。namespace スコープの Operator は柔軟性があるために優先して使用されます。これらの Operator は切り離されたアップグレード、障害対応およびモニタリングのための namespace の分離、および API 定義の差異化を可能にします。
   新規の Ansible ベース、namespace スコープの **memcached-operator** プロジェクトを作成し、そのディレクトリーに切り換えるには、以下のコマンドを使用します。

   ```
   $ operator-sdk new memcached-operator \
     --api-version=cache.example.com/v1alpha1 \
     --kind=Memcached \
     --type=ansible
   $ cd memcached-operator
   ```

   これにより、APIVersion **example.com/v1apha1** および Kind **Memcached** の Memcached リソースを監視するための **memcached-operator** プロジェクトが作成されます。

   b. **クラスタースコープの Operator** はクラスター全体でリソースを監視し、管理します。これは、 クラスター全体で発行される証明書を管理できるようにクラスタースコープのパーミッションでデプロイされ、監視される **cert-manager** Operator などの場合に役立ちます。

memcached-operator プロジェクトをクラススコープになるように作成し、そのディレクトリーに切り換えるには、以下のコマンドを実行します。

```
$ operator-sdk new memcached-operator \
    --cluster-scoped \
    --api-version=cache.example.com/v1alpha1 \
    --kind=Memcached \
    --type=ansible
$ cd memcached-operator
```

--cluster-scoped フラグを使用することにより、以下の変更を含む新規の Operator のスキャフォールディングが実行されます。

- **deploy/operator.yaml**: Pod の namespace に設定するのではなく、**WATCH_NAMESPACE=""** を設定します。
- **deploy/role.yaml**: **Role** ではなく **ClusterRole** を使用します。
- **deploy/role_binding.yaml**:
  - **RoleBinding** ではなく **ClusterRoleBinding** を使用します。
  - subject の namespace を **REPLACE_NAMESPACE** に設定します。これは、Operator がデプロイされる namespace に変更される必要があります。

2. **Operator ロジックをカスタマイズします。**
この例では、**memcached-operator** はそれぞれの **Memcached** カスタムリソース (CR) について以下の調整 (reconciliation) ロジックを実行します。

- **memcached** Deployment を作成します (ない場合)。
- Deployment のサイズが **Memcached** CR で指定されるのと同じであることを確認します。

デフォルトで、**memcached-operator** は **watches.yaml** ファイルに示されるように **Memcached** リソースイベントを監視し、Ansible ロール **Memcached** を実行します。

```
- version: v1alpha1
  group: cache.example.com
  kind: Memcached
```

オプションで、以下のロジックを **watches.yaml** ファイルでカスタマイズできます。

a. **role** オプションを指定して、**ansible-runner** を Ansible ロールを使って起動する際に Operator がこの特定のパスを使用するように設定します。デフォルトでは、新規コマンドでロールが置かれる場所への絶対パスが入力されます。

```
- version: v1alpha1
  group: cache.example.com
  kind: Memcached
  role: /opt/ansible/roles/memcached
```

b. **playbook** オプションを **watches.yaml** ファイルに指定して、**ansible-runner** を Ansible Playbook で起動する際に Operator がこの指定されたパスを使用するように設定します。

```
- version: v1alpha1
  group: cache.example.com
```

```
kind: Memcached
playbook: /opt/ansible/playbook.yaml
```

3. **Memcached Ansible ロールをビルドします。**
生成された Ansible ロールを **roles/memcached/** ディレクトリーの下で変更します。この Ansible ロールは、リソースの変更時に実行されるロジックを制御します。

a. **Memcached 仕様を定義します。**
Ansible ベースの Operator の定義は Ansible 内ですべて実行できます。Ansible Operator は CR 仕様フィールドのすべてのキー/値ペアを 変数として Ansible に渡します。仕様フィールドのすべての変数の名前は、Ansible の実行前に Operator によってスネークケース (小文字 + アンダースコア) に変換されます。たとえば、仕様の **serviceAccount** は Ansible では **service_account** になります。

ヒント

Ansible で変数についてのタイプの検証を実行し、アプリケーションが予想される入力を受信できることを確認する必要があります。

ユーザーが **spec** フィールドを設定しない場合、 **roles/memcached/defaults/main.yml** ファイルを変更してデフォルトを設定します。

```
size: 1
```

b. **Memcached デプロイメントを定義します。**
Memcached 仕様が定義された状態で、リソースの変更に対する Ansible の実行内容を定義できます。これは Ansible ロールであるため、デフォルトの動作は **roles/memcached/tasks/main.yml** ファイルでタスクを実行します。

ここでの目的は、Ansible で **memcached:1.4.36-alpine** イメージを実行する Deployement を作成することにあります (Deployment がない場合)。Ansible 2.7+ は k8s Ansible モジュールをサポートします。 この例では、このモジュールを活用し、Deployment 定義を制御します。

roles/memcached/tasks/main.yml を以下に一致するように変更します。

```
- name: start memcached
  k8s:
    definition:
      kind: Deployment
      apiVersion: apps/v1
      metadata:
        name: '{{ meta.name }}-memcached'
        namespace: '{{ meta.namespace }}'
      spec:
        replicas: "{{size}}"
        selector:
          matchLabels:
            app: memcached
        template:
          metadata:
            labels:
              app: memcached
          spec:
```

```
      containers:
      - name: memcached
        command:
        - memcached
        - -m=64
        - -o
        - modern
        - -v
        image: "docker.io/memcached:1.4.36-alpine"
        ports:
          - containerPort: 11211
```

注記

この例では、**size** 変数を使用し、 **Memcached** Deployment のレプリカ数を制御しています。この例では、デフォルトを **1** に設定しますが、任意のユーザーがこのデフォルトを上書きする CR を作成することができます。

4. **CRD をデプロイします。**
Operator の実行前に、Kubernetes は Operator が監視する新規カスタムリソース定義 (CRD) について把握している必要があります。**Memcached** CRD をデプロイします。

```
$ oc create -f deploy/crds/cache_v1alpha1_memcached_crd.yaml
```

5. **Operator をビルドし、実行します。**
Operator をビルドし、実行する方法として 2 つの方法を使用できます。

- Kubernetes クラスター内の Pod を使用
- **operator-sdk up** コマンドを使用してクラスター外で Go プログラムを使用

以下の方法のいずれかを選択します。

a. Kubernetes クラスター内で **Pod として実行** します。これは実稼働環境での優先される方法です。

i. **memcached-operator** イメージをビルドし、これをレジストリーにプッシュします。

```
$ operator-sdk build quay.io/example/memcached-operator:v0.0.1
$ podman push quay.io/example/memcached-operator:v0.0.1
```

ii. Deployment マニフェストは **deploy/operator.yaml** ファイルに生成されます。このファイルの Deployment イメージは、プレースホルダー **REPLACE_IMAGE** から直前にビルドされたイメージに変更される必要があります。これを実行するには、以下を実行します。

```
$ sed -i 's|REPLACE_IMAGE|quay.io/example/memcached-operator:v0.0.1|g'
deploy/operator.yaml
```

iii. **--cluster-scoped=true** フラグを使用して Operator を作成した場合、生成された **ClusterRoleBinding** でサービスアカウント namespace を Operator をデプロイする場所に一致するように更新します。

```
$ export OPERATOR_NAMESPACE=$(oc config view --minify -o
```

```
jsonpath='{.contexts[0].context.namespace}')
$ sed -i "s|REPLACE_NAMESPACE|$OPERATOR_NAMESPACE|g"
deploy/role_binding.yaml
```

OSX でこれらの手順を実行している場合には、代わりに以下のコマンドを使用します。

```
$ sed -i "" 's|REPLACE_IMAGE|quay.io/example/memcached-operator:v0.0.1|g'
deploy/operator.yaml
$ sed -i "" "s|REPLACE_NAMESPACE|$OPERATOR_NAMESPACE|g"
deploy/role_binding.yaml
```

iv. **memcached-operator** をデプロイします。

```
$ oc create -f deploy/service_account.yaml
$ oc create -f deploy/role.yaml
$ oc create -f deploy/role_binding.yaml
$ oc create -f deploy/operator.yaml
```

v. **memcached-operator** が稼働していることを確認します。

```
$ oc get deployment
NAME                 DESIRED   CURRENT   UP-TO-DATE   AVAILABLE   AGE
memcached-operator       1         1         1            1          1m
```

b. **クラスター外で実行します**。この方法は、デプロイメントおよびテストの速度を上げるために開発サイクル時に優先される方法です。
Ansible Runner および Ansible Runner HTTP プラグインがインストールされていることを確認します。 インストールされていない場合、CR の作成時に Ansible Runner から予想しないエラーが発生します。

さらに、**watches.yaml** ファイルで参照されるロールパスがマシン上にある必要があります。通常、コンテナーはディスク上のロールが置かれる場所で使用されるため、ロールは設定済みの Ansible ロールパス (例: **/etc/ansible/roles**) に手動でコピーされる必要があります。

i. **$HOME/.kube/config** にあるデフォルトの Kubernetes 設定ファイルを使って Operator をローカルに実行するには、以下を実行します。

```
$ operator-sdk up local
```

提供された Kubernetes 設定ファイルを使って Operator をローカルに実行するには、以下を実行します。

```
$ operator-sdk up local --kubeconfig=config
```

6. **Memcached CR を作成します**。

a. 以下に示されるように **deploy/crds/cache_v1alpha1_memcached_cr.yaml** ファイルを変更し、 **Memcached** CR を作成します。

```
$ cat deploy/crds/cache_v1alpha1_memcached_cr.yaml
apiVersion: "cache.example.com/v1alpha1"
kind: "Memcached"
```

```
metadata:
  name: "example-memcached"
spec:
  size: 3

$ oc apply -f deploy/crds/cache_v1alpha1_memcached_cr.yaml
```

b. **memcached-operator** が CR の Deployment を作成できることを確認します。

```
$ oc get deployment
NAME                 DESIRED   CURRENT   UP-TO-DATE   AVAILABLE   AGE
memcached-operator       1         1         1            1         2m
example-memcached        3         3         3            3         1m
```

c. Pod で 3 つのレプリカが作成されていることを確認します。

```
$ oc get pods
NAME                                  READY     STATUS    RESTARTS   AGE
example-memcached-6fd7c98d8-7dqdr     1/1       Running   0          1m
example-memcached-6fd7c98d8-g5k7v     1/1       Running   0          1m
example-memcached-6fd7c98d8-m7vn7     1/1       Running   0          1m
memcached-operator-7cc7cfdf86-vvjqk   1/1       Running   0          2m
```

7. **サイズを更新します。**

a. **memcached** CR の **spec.size** フィールドを **3** から **4** に変更し、変更を適用します。

```
$ cat deploy/crds/cache_v1alpha1_memcached_cr.yaml
apiVersion: "cache.example.com/v1alpha1"
kind: "Memcached"
metadata:
  name: "example-memcached"
spec:
  size: 4

$ oc apply -f deploy/crds/cache_v1alpha1_memcached_cr.yaml
```

b. Operator が Deployment サイズを変更することを確認します。

```
$ oc get deployment
NAME                 DESIRED   CURRENT   UP-TO-DATE   AVAILABLE   AGE
example-memcached    4         4         4            4           5m
```

8. **リソースをクリーンアップします。**

```
$ oc delete -f deploy/crds/cache_v1alpha1_memcached_cr.yaml
$ oc delete -f deploy/operator.yaml
$ oc delete -f deploy/role_binding.yaml
$ oc delete -f deploy/role.yaml
$ oc delete -f deploy/service_account.yaml
$ oc delete -f deploy/crds/cache_v1alpha1_memcached_crd.yaml
```

11.2.4. K8S Ansible モジュールの使用によるアプリケーションライフサイクルの管理

Ansible を使用して Kubernetes でアプリケーションのライフサイクルを管理するには、**k8s** Ansible モジュールを使用できます。この Ansible モジュールにより、開発者は既存の Kubernetes リソースファイル (YAML で作成されている) を利用するか、またはネイティブの Ansible でライフサイクル管理を表現することができます。

Ansible を既存の Kubernetes リソースファイルと併用する最大の利点の 1 つに、Ansible のいくつかを変数のみを使う単純な方法でのリソースのカスタマイズを可能にする Jinja テンプレートを使用できる点があります。

このセクションでは、**k8s** Ansible モジュールの使用法を詳細に説明します。使用を開始するには、Playbook を使用してローカルワークステーションにモジュールをインストールし、これをテストしてから、Operator 内での使用を開始します。

11.2.4.1. k8s Ansible モジュールのインストール

k8s Ansible モジュールをローカルワークステーションにインストールするには、以下を実行します。

手順

1. Ansible 2.6+ をインストールします。

```
$ sudo yum install ansible
```

2. pip を使用して **OpenShift python クライアント**パッケージをインストールします。

```
$ pip install openshift
```

11.2.4.2. k8s Ansible モジュールのローカルでのテスト

開発者が毎回 Operator を実行し、再ビルドするのではなく、Ansible コードをローカルマシンから実行する方が利点がある場合があります。

手順

1. 新規 Ansible ベースの Operator プロジェクトを初期化します。

```
$ operator-sdk new --type ansible --kind Foo --api-version foo.example.com/v1alpha1 foo-operator
Create foo-operator/tmp/init/galaxy-init.sh
Create foo-operator/tmp/build/Dockerfile
Create foo-operator/tmp/build/test-framework/Dockerfile
Create foo-operator/tmp/build/go-test.sh
Rendering Ansible Galaxy role [foo-operator/roles/Foo]...
Cleaning up foo-operator/tmp/init
Create foo-operator/watches.yaml
Create foo-operator/deploy/rbac.yaml
Create foo-operator/deploy/crd.yaml
Create foo-operator/deploy/cr.yaml
Create foo-operator/deploy/operator.yaml
Run git init ...
Initialized empty Git repository in /home/dymurray/go/src/github.com/dymurray/opsdk/foo-operator/.git/
Run git init done
```

```
$ cd foo-operator
```

2. 必要な Ansible ロジックを使用して **roles/Foo/tasks/main.yml** ファイルを変更します。この例では、変数の切り替えと共に namespace を作成し、削除します。

```
- name: set test namespace to {{ state }}
  k8s:
    api_version: v1
    kind: Namespace
    state: "{{ state }}"
  ignore_errors: true (1)
```

(1) **ignore_errors: true** を設定することにより、存在しないプロジェクトを削除しても失敗しません。

3. **roles/Foo/defaults/main.yml** ファイルを、デフォルトで **state** を **present** に設定するように変更します。

```
state: present
```

4. 上部ディレクトリーに、**Foo** ロールを含む Ansible Playbook **playbook.yml** を作成します。

```
- hosts: localhost
  roles:
    - Foo
```

5. Playbook を実行します。

```
$ ansible-playbook playbook.yml
 [WARNING]: provided hosts list is empty, only localhost is available. Note that the implicit
localhost does not match 'all'

PLAY [localhost] ***************************************************************************

TASK [Gathering Facts] *********************************************************************
ok: [localhost]

Task [Foo : set test namespace to present]
changed: [localhost]

PLAY RECAP *********************************************************************************
localhost                  : ok=2    changed=1    unreachable=0    failed=0
```

6. namespace が作成されていることを確認します。

```
$ oc get namespace
NAME          STATUS    AGE
default       Active    28d
kube-public   Active    28d
kube-system   Active    28d
test          Active    3s
```

7. **state** を **absent** に設定して Playbook を再実行します。

```
$ ansible-playbook playbook.yml --extra-vars state=absent
 [WARNING]: provided hosts list is empty, only localhost is available. Note that the implicit
localhost does not match 'all'

PLAY [localhost] ***************************************************************************

TASK [Gathering Facts] *********************************************************************
ok: [localhost]

Task [Foo : set test namespace to absent]
changed: [localhost]

PLAY RECAP *********************************************************************************
localhost                  : ok=2    changed=1    unreachable=0    failed=0
```

8. namespace が削除されていることを確認します。

```
$ oc get namespace
NAME          STATUS    AGE
default       Active    28d
kube-public   Active    28d
kube-system   Active    28d
```

11.2.4.3. Operator 内での k8s Ansible モジュールのテスト

k8s Ansible モジュールをローカルで使用することに慣れたら、カスタムリソース (CR) の変更時に Operator 内で同じ Ansible ロジックをトリガーできます。この例では、Ansible ロールを、Operator が監視する特定の Kubernetes リソースにマップします。このマッピングは監視ファイルで実行されます。

11.2.4.3.1. Ansible ベース Operator のローカルでのテスト

Ansible ワークフローのテストをローカルで実行することに慣れたら、ローカルに実行される Ansible ベースの Operator 内でロジックをテストできます。

これを実行するには、Operator プロジェクトの上部ディレクトリーから **operator-sdk up local** コマンドを使用します。このコマンドは **./watches.yaml** ファイルから読み取り、 **~/.kube/config** ファイルを使用して **k8s** Ansible モジュールが実行するように Kubernetes クラスターと通信します。

手順

1. **up local** コマンドは **./watches.yaml** ファイルから読み取るため、Operator の作成者はいくつかのオプションを選択できます。**role** が単独で残される場合 (デフォルトでは **/opt/ansible/roles/<name>**)、ロールを Operator から **/opt/ansible/roles/** ディレクトリーに直接コピーする必要があります。
これは、現行ディレクトリーからの変更が反映されないために複雑になります。この代わりに、**role** フィールドを現行ディレクトリーを参照するように変更し、既存の行をコメントアウトします。

```
- version: v1alpha1
  group: foo.example.com
  kind: Foo
  #  role: /opt/ansible/roles/Foo
  role: /home/user/foo-operator/Foo
```

2. カスタムリソース定義 (CRD) およびカスタムリソース (CR) **Foo** の適切なロールベースアクセス制御 (RBAC) 定義を作成します。**operator-sdk** コマンドは、**deploy/** ディレクトリー内にこれらのファイルを自動生成します。

```
$ oc create -f deploy/crds/foo_v1alpha1_foo_crd.yaml
$ oc create -f deploy/service_account.yaml
$ oc create -f deploy/role.yaml
$ oc create -f deploy/role_binding.yaml
```

3. **up local** コマンドを実行します。

```
$ operator-sdk up local
[...]
INFO[0000] Starting to serve on 127.0.0.1:8888
INFO[0000] Watching foo.example.com/v1alpha1, Foo, default
```

4. Operator はリソース **Foo** でイベントを監視しているため、CR の作成により、Ansible ロールの実行がトリガーされます。**deploy/cr.yaml** ファイルを表示します。

```
apiVersion: "foo.example.com/v1alpha1"
kind: "Foo"
metadata:
  name: "example"
```

spec フィールドは設定されていないため、Ansible は追加の変数なしで起動します。次のセクションでは、追加の変数が CR から Ansible に渡される方法について説明します。このため、Operator に同じでデフォルト値を設定することが重要になります。

5. デフォルト変数 **state** を **present** に設定し、**Foo** の CR インスタンスを作成します。

```
$ oc create -f deploy/cr.yaml
```

6. namespace **test** が作成されていることを確認します。

```
$ oc get namespace
NAME          STATUS    AGE
default       Active    28d
kube-public   Active    28d
kube-system   Active    28d
test          Active    3s
```

7. **deploy/cr.yaml** ファイルを、**state** フィールドを **absent** に設定するように変更します。

```
apiVersion: "foo.example.com/v1alpha1"
kind: "Foo"
metadata:
  name: "example"
spec:
  state: "absent"
```

8. 変更を適用し、namespace が定義されていることを確認します。

```
$ oc apply -f deploy/cr.yaml
```

```
$ oc get namespace
NAME          STATUS    AGE
default       Active    28d
kube-public   Active    28d
kube-system   Active    28d
```

11.2.4.3.2. Ansible ベース Operator のクラスター上でのテスト

Ansible ロジックを Ansible ベース Operator 内でローカルに実行することに慣れたら、OpenShift Container Platform などの Kubernetes クラスターの Pod 内で Operator をテストすることができます。Pod のクラスターでの実行は、実稼働環境で優先される方法です。

手順

1. **foo-operator** イメージをビルドし、これをレジストリーにプッシュします。

```
$ operator-sdk build quay.io/example/foo-operator:v0.0.1
$ podman push quay.io/example/foo-operator:v0.0.1
```

2. Deployment マニフェストは **deploy/operator.yaml** ファイルに生成されます。このファイルの Deployment イメージはプレースホルダーの **REPLACE_IMAGE** から以前にビルドされたイメージに変更される必要があります。これを実行するには、以下のコマンドを実行します。

```
$ sed -i 's|REPLACE_IMAGE|quay.io/example/foo-operator:v0.0.1|g' deploy/operator.yaml
```

   OSX でこれらの手順を実行している場合には、代わりに以下のコマンドを実行します。

```
$ sed -i "" 's|REPLACE_IMAGE|quay.io/example/foo-operator:v0.0.1|g' deploy/operator.yaml
```

3. **foo-operator** をデプロイします。

```
$ oc create -f deploy/crds/foo_v1alpha1_foo_crd.yaml # if CRD doesn't exist already
$ oc create -f deploy/service_account.yaml
$ oc create -f deploy/role.yaml
$ oc create -f deploy/role_binding.yaml
$ oc create -f deploy/operator.yaml
```

4. **foo-operator** が稼働していることを確認します。

```
$ oc get deployment
NAME                 DESIRED   CURRENT   UP-TO-DATE   AVAILABLE   AGE
foo-operator      1       1       1          1          1m
```

11.2.5. k8s_status Ansible モジュールの使用によるカスタムリソースステータスの管理

Ansible ベースの Operator は、カスタムリソース (CR) **status** サブリソースを以前の Ansible 実行についての一般的な情報で自動的に更新します。これには、以下のように成功したタスクおよび失敗したタスクの数と関連するエラーメッセージが含まれます。

```
status:
  conditions:
    - ansibleResult:
      changed: 3
```

```
    completion: 2018-12-03T13:45:57.13329
    failures: 1
    ok: 6
    skipped: 0
  lastTransitionTime: 2018-12-03T13:45:57Z
  message: 'Status code was -1 and not [200]: Request failed: <urlopen error [Errno
    113] No route to host>'
  reason: Failed
  status: "True"
  type: Failure
- lastTransitionTime: 2018-12-03T13:46:13Z
  message: Running reconciliation
  reason: Running
  status: "True"
  type: Running
```

Ansible ベースの Operator は、Operator の作成者が **k8s_status** Ansible モジュールでカスタムステータスの値を指定することも可能にします。これにより、作成者は必要に応じ、任意のキー/値のペアを使って Ansible から **status** を更新できます。

デフォルトでは、Ansible ベースの Operator には、上記のように常に汎用的な Ansible 実行出力が含まれます。アプリケーションのステータスが Ansible 出力で更新 **されない** ようにする必要がある場合に、アプリケーションからステータスを手動で追跡することができます。

手順

1. CR ステータスをアプリケーションから手動で追跡するには、 **manageStatus** フィールドを **false** に設定して監視ファイルを更新します。

```
- version: v1
  group: api.example.com
  kind: Foo
  role: /opt/ansible/roles/Foo
  manageStatus: false
```

2. 次に、**k8s_status** Ansible モジュールを使用してサブリソースを更新します。たとえば、キー **foo** および値 **bar** を使用して更新するには、**k8s_status** を以下のように使用することができます。

```
- k8s_status:
  api_version: app.example.com/v1
  kind: Foo
  name: "{{ meta.name }}"
  namespace: "{{ meta.namespace }}"
  status:
    foo: bar
```

追加リソース

- Ansible ベース Operator からのユーザー主導のステータス管理を行う方法についての詳細は、「Ansible Operator Status Proposal」を参照してください。

11.2.5.1. ローカルでのテスト時の k8s_status Ansible モジュールの使用

Operator が **k8s_status** Ansible モジュールを使用し、Operator を **operator-sdk up local** コマンドでローカルにテストする必要がある場合、モジュールを Ansible が予想する場所にインストールする必要があります。これは、Ansible の **library** 設定オプションを使用して実行されます。

この例では、ユーザーがサードパーティーの Ansible モジュールを **/usr/share/ansible/library/** ディレクトリーに配置することを前提としています。

手順

1. **k8s_status** モジュールをインストールするには、**ansible.cfg** ファイルを、インストール済みの Ansible モジュールを **/usr/share/ansible/library/** ディレクトリーで検索するように設定します。

```
$ echo "library=/usr/share/ansible/library/" >> /etc/ansible/ansible.cfg
```

2. **k8s_status.py** ファイルを **/usr/share/ansible/library/** ディレクトリーに追加します。

```
$ wget https://raw.githubusercontent.com/openshift/ocp-release-operator-
sdk/master/library/k8s_status.py -O /usr/share/ansible/library/k8s_status.py
```

11.2.6. 追加リソース

- Operator SDK によって作成されるプロジェクトディレクトリー構造についての詳細は、「Appendices」を参照してください。
- Reaching for the Stars with Ansible Operator - Red Hat OpenShift Blog
- Operator Development Guide for Red Hat Partners

11.3. HELM ベース OPERATOR の作成

以下では、Operator SDK での Helm チャートのサポートについての概要を説明し、Operator 作成者を対象に、既存の Helm チャートを使用する **operator-sdk** CLI ツールで Nginx Operator をビルドし、実行する例を示します。

11.3.1. Operator SDK での Helm チャートのサポート

Operator Framework は **Operator** という Kubernetes ネイティブアプリケーションを効果的かつ自動化された拡張性のある方法で管理するためのオープンソースツールキットです。このフレームワークには Operator SDK が含まれ、これは Kubernetes API の複雑性を把握していなくても、それぞれの専門知識に基づいて Operator のブートストラップおよびビルドを実行できるように開発者を支援します。

Operator プロジェクトを生成するための Operator SDK のオプションの 1 つとして、Go コードを作成せずに既存の Helm チャートを使用して Kubernetes リソースを統一されたアプリケーションとしてデプロイするオプションがあります。このような Helm ベースの Operator では、変更はチャートの一部として生成される Kubernetes オブジェクトに適用されるため、ロールアウト時にロジックをほとんど必要としないステートレスなアプリケーションを使用する際に適しています。いくらか制限があるような印象を与えるかもしれませんが、Kubernetes コミュニティーがビルドする Helm チャートが急速に増加していることからも分かるように、この Operator は数多くのユーザーケースに対応することができます。

Operator の主な機能として、アプリケーションインスタンスを表すカスタムオブジェクトから読み取り、必要な状態を実行されている内容に一致させることができます。Helm ベース Operator の場合、オブジェクトの仕様フィールドは、通常 Helm の **values.yaml** ファイルに記述される設定オプションの一

覧です。Helm CLI を使用してフラグ付きの値を設定する代わりに (例: **helm install -f values.yaml**)、これらをカスタムリソース (CR) 内で表現することができます。 これにより、ネイティブ Kubernetes オブジェクトとして、適用される RBAC および監査証跡の利点を活用できます。

Tomcat という単純な CR の例:

```
apiVersion: apache.org/v1alpha1
kind: Tomcat
metadata:
  name: example-app
spec:
  replicaCount: 2
```

この場合の **replicaCount** 値、**2** は以下が使用されるチャートのテンプレートに伝播されます。

```
{{ .Values.replicaCount }}
```

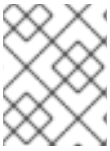
Operator のビルドおよびデプロイ後に、CR の新規インスタンスを作成してアプリケーションの新規インスタンスをデプロイしたり、 **oc** コマンドを使用してすべての環境で実行される異なるインスタンスを一覧表示したりすることができます。

```
$ oc get Tomcats --all-namespaces
```

Helm CLI を使用したり、Tiller をインストールしたりする必要はありません。Helm ベースの Operator はコードを Helm プロジェクトからインポートします。Operator のインスタンスを実行状態にし、カスタムリソース定義 (CRD) に CR を登録することのみが必要になります。 さらにこれは RBAC に準拠するため、実稼働環境の変更を簡単に防止することができます。

11.3.2. Operator SDK CLI のインストール

Operator SDK には、開発者による新規 Operator プロジェクトの作成、ビルドおよびデプロイを支援をする CLI ツールが含まれます。ワークステーションに SDK CLI をインストールして、独自の Operator のオーサリングを開始することができます。



注記

以下では、ローカル Kubernetes クラスターとしての minikube v0.25.0+ とパブリックレジストリーの quay.io を使用します。

11.3.2.1. GitHub リリースからのインストール

GitHub のプロジェクトから SDK CLI の事前ビルドリリースのバイナリーをダウンロードし、インストールできます。

前提条件

- **docker** v17.03+
- OpenShift CLI (**oc**) v4.1+ (インストール済み)
- Kubernetes v1.11.3+ に基づくクラスターへのアクセス
- コンテナーレジストリーへのアクセス

手順

1. リリースバージョン変数を設定します。

```
RELEASE_VERSION=v0.8.0
```

2. リリースバイナリーをダウンロードします。

   - Linux の場合

```
$ curl -OJL https://github.com/operator-framework/operator-sdk/releases/download/${RELEASE_VERSION}/operator-sdk-${RELEASE_VERSION}-x86_64-linux-gnu
```

   - MacOS の場合

```
$ curl -OJL https://github.com/operator-framework/operator-sdk/releases/download/${RELEASE_VERSION}/operator-sdk-${RELEASE_VERSION}-x86_64-apple-darwin
```

3. ダウンロードしたリリースのバイナリーを確認します。

   a. 提供された ASC ファイルをダウンロードします。

      - Linux の場合

```
$ curl -OJL https://github.com/operator-framework/operator-sdk/releases/download/${RELEASE_VERSION}/operator-sdk-${RELEASE_VERSION}-x86_64-linux-gnu.asc
```

      - MacOS の場合

```
$ curl -OJL https://github.com/operator-framework/operator-sdk/releases/download/${RELEASE_VERSION}/operator-sdk-${RELEASE_VERSION}-x86_64-apple-darwin.asc
```

   b. バイナリーと対応する ASC ファイルを同じディレクトリーに置き、以下のコマンドを実行してバイナリーを確認します。

      - Linux の場合

```
$ gpg --verify operator-sdk-${RELEASE_VERSION}-x86_64-linux-gnu.asc
```

      - MacOS の場合

```
$ gpg --verify operator-sdk-${RELEASE_VERSION}-x86_64-apple-darwin.asc
```

      保守管理者の公開キーがワークステーションにない場合は、以下のエラーが出されます。

```
$ gpg --verify operator-sdk-${RELEASE_VERSION}-x86_64-apple-darwin.asc
$ gpg: assuming signed data in 'operator-sdk-${RELEASE_VERSION}-x86_64-apple-darwin'
```

```
$ gpg: Signature made Fri Apr  5 20:03:22 2019 CEST
$ gpg:                using RSA key <key_id> 1
$ gpg: Can't check signature: No public key
```

1 RSA キー文字列。

キーをダウンロードするには、以下のコマンドを実行し、 **<key_id>** を直前のコマンドの出力で提供された RSA キー文字列に置き換えます。

```
$ gpg [--keyserver keys.gnupg.net] --recv-key "<key_id>" 1
```

1 キーサーバーが設定されていない場合、これを **--keyserver** オプションで指定します。

4. リリースバイナリーを **PATH** にインストールします。
   - Linux の場合

```
$ chmod +x operator-sdk-${RELEASE_VERSION}-x86_64-linux-gnu
$ sudo cp operator-sdk-${RELEASE_VERSION}-x86_64-linux-gnu
/usr/local/bin/operator-sdk
$ rm operator-sdk-${RELEASE_VERSION}-x86_64-linux-gnu
```

   - MacOS の場合

```
$ chmod +x operator-sdk-${RELEASE_VERSION}-x86_64-apple-darwin
$ sudo cp operator-sdk-${RELEASE_VERSION}-x86_64-apple-darwin
/usr/local/bin/operator-sdk
$ rm operator-sdk-${RELEASE_VERSION}-x86_64-apple-darwin
```

5. CLI ツールが正しくインストールされていることを確認します。

```
$ operator-sdk version
```

11.3.2.2. Homebrew からのインストール

Homebrew を使用して SDK CLI をインストールできます。

前提条件

- Homebrew
- **docker** v17.03+
- OpenShift CLI (**oc**) v4.1+ (インストール済み)
- Kubernetes v1.11.3+ に基づくクラスターへのアクセス
- コンテナーレジストリーへのアクセス

手順

1. **brew** コマンドを使用して SDK CLI をインストールします。

```
$ brew install operator-sdk
```

2. CLI ツールが正しくインストールされていることを確認します。

```
$ operator-sdk version
```

11.3.2.3. ソースを使用したコンパイルおよびインストール

Operator SDK ソースコードを取得して、SDK CLI をコンパイルし、インストールできます。

前提条件

- dep v0.5.0+
- Git
- Go v1.10+
- **docker** v17.03+
- OpenShift CLI (**oc**) v4.1+ (インストール済み)
- Kubernetes v1.11.3+ に基づくクラスターへのアクセス
- コンテナーレジストリーへのアクセス

手順

1. **operator-sdk** リポジトリーのクローンを作成します。

```
$ mkdir -p $GOPATH/src/github.com/operator-framework
$ cd $GOPATH/src/github.com/operator-framework
$ git clone https://github.com/operator-framework/operator-sdk
$ cd operator-sdk
```

2. 必要なリリースブランチをチェックアウトします。

```
$ git checkout master
```

3. SDK CLI ツールをコンパイルし、インストールします。

```
$ make dep
$ make install
```

これにより、**$GOPATH/bin** に CLI バイナリー **operator-sdk** がインストールされます。

4. CLI ツールが正しくインストールされていることを確認します。

```
$ operator-sdk version
```

11.3.3. Operator SDK を使用した Helm ベースの Operator のビルド

以下の手順では、Operator SDK が提供するツールおよびライブラリーを使用して Helm チャートがサポートする単純な Nginx Operator のビルドの例について説明します。

ヒント

各チャートについて新規 Operator をビルドすることは最も効果的な方法と言えます。これにより、Hem ベースの Operator から移行して Go で完全装備の Operator を作成する場合などに、さらに多くのネイティブ動作をする Kubernetes API (例: **oc get Nginx**) の使用および柔軟性が可能になります。

前提条件

- 開発ワークステーションにインストールされる Operator SDK CLI
- **cluster-admin** パーミッションを持つアカウントを使用した Kubernetes ベースのクラスターr v1.11.3+ (OpenShift Container Platform 4.1 など) へのアクセス
- OpenShift CLI (**oc**) v4.1+ (インストール済み)

手順

1. **operator-sdk new** コマンドを使用して、namespace スコープまたはクラススコープのいずれかで **新規 Operator プロジェクトを作成します**。以下のいずれかを選択します。

   a. **namespace スコープ Operator** (デフォルト) は単一 namespace でリソースを監視し、管理します。namespace スコープの Operator は柔軟性があるために優先して使用されます。これらの Operator は切り離されたアップグレード、障害対応およびモニタリングのための namespace の分離、および API 定義の差異化を可能にします。
   新規の Helm ベース、namespace スコープの **nginx-operator** プロジェクトを作成するには、以下のコマンドを使用します。

   ```
   $ operator-sdk new nginx-operator \
     --api-version=example.com/v1alpha1 \
     --kind=Nginx \
     --type=helm
   $ cd nginx-operator
   ```

   これにより、とりわけ APIVersion **example.com/v1apha1** および Kind **Nginx** の Nginx リソースを監視する目的で **nginx-operator** プロジェクトが作成されます。

   b. **クラススコープの Operator** はクラスター全体でリソースを監視し、管理します。これは、 クラスター全体で発行される証明書を管理できるようにクラススコープのパーミッションでデプロイされ、監視される **cert-manager** Operator などの場合に役立ちます。
   nginx-operator プロジェクトをクラススコープになるように作成するには、以下のコマンドを実行します。

   ```
   $ operator-sdk new nginx-operator \
       --cluster-scoped \
       --api-version=example.com/v1alpha1 \
       --kind=Nginx \
       --type=helm
   ```

   --cluster-scoped フラグを使用することにより、以下の変更を含む新規の Operator のスキャフォールディングが実行されます。

- **deploy/operator.yaml**: Pod の namespace に設定するのではなく、**WATCH_NAMESPACE=""** を設定します。
- **deploy/role.yaml**: **Role** ではなく **ClusterRole** を使用します。
- **deploy/role_binding.yaml**:
  - **RoleBinding** ではなく **ClusterRoleBinding** を使用します。
  - subject の namespace を **REPLACE_NAMESPACE** に設定します。これは、Operator がデプロイされる namespace に変更される必要があります。

2. **Operator ロジックをカスタマイズします。**
この例では、**nginx-operator** はそれぞれの **Nginx** カスタムリソース (CR) について以下の調整 (reconciliation) ロジックを実行します。

- Nginx デプロイメントを作成します (ない場合)。
- Nginx サービスを作成します (ない場合)。
- Nginx Ingress を作成します (有効にされているが存在しない場合)。
- Deployment、Service、およびオプションの Ingress が Nginx CR で指定される必要な設定 (レプリカ数、イメージ、サービスタイプなど) に一致することを確認します。

デフォルトで、**nginx-operator** は **watches.yaml** ファイルに示されるように **Nginx** リソースイベントを監視し、指定されたチャートを使用して Helm リリースを実行します。

```
- version: v1alpha1
  group: example.com
  kind: Nginx
  chart: /opt/helm/helm-charts/nginx
```

a. **Nginx Helm チャートを確認します。**
Helm Operator プロジェクトの作成時に、Operator SDK は、単純な Nginx リリース用のテンプレートセットが含まれる Helm チャートのサンプルを作成します。

この例では、Helm チャート開発者がリリースについての役立つ情報を伝えるために使用する **NOTES.txt** テンプレートと共に、Deployment、Service、および Ingress リソース用にテンプレートを利用できます。

Helm チャートの使用に慣れていない場合は、Helm Chart 開発者用のドキュメント を参照してください。

b. **Nginx CR 仕様を確認します。**
Helm は値 (value) という概念を使用して、Helm チャートの **values.yaml** ファイルに定義される Helm チャートのデフォルトをカスタマイズします。

CR 仕様に必要な値を設定し、これらのデフォルトを上書きします。例としてレプリカ数を使用することができます。

i. まず、**helm-charts/nginx/values.yaml** ファイルで、チャートに **replicaCount** という値が含まれ、これがデフォルトで **1** に設定されていることを検査します。デプロイメントに 2 つの Nginx インスタンスを設定するには、CR 仕様に **replicaCount: 2** が含まれる必要があります。
deploy/crds/example_v1alpha1_nginx_cr.yaml ファイルを以下のように更新します。

```
apiVersion: example.com/v1alpha1
kind: Nginx
metadata:
  name: example-nginx
spec:
  replicaCount: 2
```

ii. 同様に、デフォルトのサービスポートは **80** に設定されます。**8080** を代わりに使用するには、サービスポートの上書きを追加して **deploy/crds/example_v1alpha1_nginx_cr.yaml** ファイルを再度更新します。

```
apiVersion: example.com/v1alpha1
kind: Nginx
metadata:
  name: example-nginx
spec:
  replicaCount: 2
  service:
    port: 8080
```

Helm Operator は、**helm install -f ./overrides.yaml** コマンドが機能するように、仕様全体を values ファイルの内容のように適用します。

3. **CRD をデプロイします。**
Operator の実行前に、Kubernetes は Operator が監視する新規カスタムリソース定義 (CRD) について把握している必要があります。以下の CRD をデプロイします。

```
$ oc create -f deploy/crds/example_v1alpha1_nginx_crd.yaml
```

4. **Operator をビルドし、実行します。**
Operator をビルドし、実行する方法として 2 つの方法を使用できます。

- Kubernetes クラスター内の Pod を使用
- **operator-sdk up** コマンドを使用してクラスター外で Go プログラムを使用

以下の方法のいずれかを選択します。

a. Kubernetes クラスター内で **Pod として実行** します。これは実稼働環境での優先される方法です。

i. **nginx-operator** イメージをビルドし、これをレジストリーにプッシュします。

```
$ operator-sdk build quay.io/example/nginx-operator:v0.0.1
$ docker push quay.io/example/nginx-operator:v0.0.1
```

ii. Deployment マニフェストは **deploy/operator.yaml** ファイルに生成されます。このファイルの Deployment イメージは、プレースホルダー **REPLACE_IMAGE** から直前にビルドされたイメージに変更される必要があります。これを実行するには、以下を実行します。

```
$ sed -i 's|REPLACE_IMAGE|quay.io/example/nginx-operator:v0.0.1|g'
deploy/operator.yaml
```

iii. **--cluster-scoped=true** フラグを使用して Operator を作成した場合、生成された **ClusterRoleBinding** でサービスアカウント namespace を Operator をデプロイする場所に一致するように更新します。

```
$ export OPERATOR_NAMESPACE=$(oc config view --minify -o
jsonpath='{.contexts[0].context.namespace}')
$ sed -i "s|REPLACE_NAMESPACE|$OPERATOR_NAMESPACE|g"
deploy/role_binding.yaml
```

OSX でこれらの手順を実行している場合には、代わりに以下のコマンドを使用します。

```
$ sed -i "" 's|REPLACE_IMAGE|quay.io/example/nginx-operator:v0.0.1|g'
deploy/operator.yaml
$ sed -i "" "s|REPLACE_NAMESPACE|$OPERATOR_NAMESPACE|g"
deploy/role_binding.yaml
```

iv. **nginx-operator** をデプロイします。

```
$ oc create -f deploy/service_account.yaml
$ oc create -f deploy/role.yaml
$ oc create -f deploy/role_binding.yaml
$ oc create -f deploy/operator.yaml
```

v. **nginx-operator** が稼働していることを確認します。

```
$ oc get deployment
NAME             DESIRED   CURRENT   UP-TO-DATE   AVAILABLE   AGE
nginx-operator     1         1         1            1          1m
```

b. **クラスター外で実行します**。この方法は、デプロイメントおよびテストの速度を上げるために開発サイクル時に優先される方法です。
watches.yaml ファイルで参照されるチャートパスがマシン上に存在している必要があります。デフォルトで、 **watches.yaml** ファイルは **operator-sdk build** コマンドでビルドされるOperator イメージを使用できるようにスキャフォールディングされます。Operator を **operator-sdk up local** コマンドで開発し、テストする場合、SDK はローカルファイルシステムでこのパスを検索します。

i. この場所に、Helm チャートのパスを参照するシンボリックリンクを作成します。

```
$ sudo mkdir -p /opt/helm/helm-charts
$ sudo ln -s $PWD/helm-charts/nginx /opt/helm/helm-charts/nginx
```

ii. **$HOME/.kube/config** にあるデフォルトの Kubernetes 設定ファイルを使って Operator をローカルに実行するには、以下を実行します。

```
$ operator-sdk up local
```

提供された Kubernetes 設定ファイルを使って Operator をローカルに実行するには、以下を実行します。

```
$ operator-sdk up local --kubeconfig=<path_to_config>
```

5. **Nginx CR をデプロイします。**
   これまでに変更した **Nginx** CR を適用します。

```
$ oc apply -f deploy/crds/example_v1alpha1_nginx_cr.yaml
```

   nginx-operator が CR の Deployment を作成することを確認します。

```
$ oc get deployment
NAME                                     DESIRED   CURRENT   UP-TO-DATE   AVAILABLE   AGE
example-nginx-b9phnoz9spckcrua7ihrbkrt1   2         2         2            2           1m
```

   Pod で 2 つのレプリカが作成されていることを確認します。

```
$ oc get pods
NAME                                                READY     STATUS    RESTARTS   AGE
example-nginx-b9phnoz9spckcrua7ihrbkrt1-f8f9c875d-fjcr9   1/1       Running   0          1m
example-nginx-b9phnoz9spckcrua7ihrbkrt1-f8f9c875d-ljbzl   1/1       Running   0          1m
```

   サービスポートが **8080** に設定されていることを確認します。

```
$ oc get service
NAME                                     TYPE        CLUSTER-IP   EXTERNAL-IP   PORT(S)    AGE
example-nginx-b9phnoz9spckcrua7ihrbkrt1   ClusterIP   10.96.26.3   <none>        8080/TCP
1m
```

6. **replicaCount を更新し、ポートを削除します。**
   spec.replicaCount フィールドを **2** から **3** に変更し、**spec.service** フィールドを削除して、変更を適用します。

```
$ cat deploy/crds/example_v1alpha1_nginx_cr.yaml
apiVersion: "example.com/v1alpha1"
kind: "Nginx"
metadata:
  name: "example-nginx"
spec:
  replicaCount: 3

$ oc apply -f deploy/crds/example_v1alpha1_nginx_cr.yaml
```

   Operator が Deployment サイズを変更することを確認します。

```
$ oc get deployment
NAME                                     DESIRED   CURRENT   UP-TO-DATE   AVAILABLE   AGE
example-nginx-b9phnoz9spckcrua7ihrbkrt1   3         3         3            3           1m
```

   サービスポートがデフォルトの **80** に設定されていることを確認します。

```
$ oc get service
NAME                                     TYPE        CLUSTER-IP   EXTERNAL-IP   PORT(S)  AGE
example-nginx-b9phnoz9spckcrua7ihrbkrt1   ClusterIP   10.96.26.3   <none>        80/TCP
1m
```

7. **リソースをクリーンアップします。**

```
$ oc delete -f deploy/crds/example_v1alpha1_nginx_cr.yaml
$ oc delete -f deploy/operator.yaml
$ oc delete -f deploy/role_binding.yaml
$ oc delete -f deploy/role.yaml
$ oc delete -f deploy/service_account.yaml
$ oc delete -f deploy/crds/example_v1alpha1_nginx_crd.yaml
```

11.3.4. 追加リソース

- Operator SDK によって作成されるプロジェクトディレクトリー構造についての詳細は、「Appendices」を参照してください。
- Operator Development Guide for Red Hat Partners

11.4. CLUSTERSERVICEVERSION (CSV) の生成

ClusterServiceVersion (CSV) は、Operator Lifecycle Manager (OLM) のクラスターでの Operator の実行を支援する Operator メタデータから作成される YAML マニフェストです。これは、ユーザーインターフェースにロゴ、説明、およびバージョンなどの情報を設定するために使用される Operator コンテナーイメージを伴うメタデータです。CSV は、Operator が必要とする RBAC ルールやそれが管理したり、依存したりするカスタムリソース (CR) などの Operator の実行に必要な技術情報の情報源でもあります。

Operator SDK には、手動で定義された YAML マニフェストおよび Operator ソースファイルに含まれる情報を使用してカスタマイズされた現行 Operator プロジェクトの **ClusterServiceVersion** (CSV) を生成するための **olm-catalog gen-csv** サブコマンドが含まれます。

CSV で生成されるコマンドにより、Operator の作成者が Operator Lifecycle Manager (OLM) について詳しく知らなくても、Operator が OLM と対話させたり、メタデータをカタログレジストリーに公開したりできます。また、Kubernetes および OLM の新機能が実装される過程で CSV 仕様は変更されるため、Operator SDK はその後の新規 CSV 機能を処理できるように更新システムを容易に拡張できるようになっています。

CSV バージョンは Operator のバージョンと同じであり、新規 CSV は Operator バージョンのアップグレード時に生成されます。Operator 作成者は **--csv-version** フラグを使用して、それらの Operator の状態を指定されたセマンティクスバージョンと共に CSV にカプセル化できます。

```
$ operator-sdk olm-catalog gen-csv --csv-version <version>
```

このアクションはべき等であり、新規バージョンが指定されるか、または YAML マニフェストまたはソースファイルが変更される場合にのみ CSV ファイルを更新します。Operator の作成者は CSV マニフェストのほとんどのフィールドを直接変更する必要はありません。変更が必要なフィールドについて、本書で定義されています。たとえば、CSV バージョンについては **metadata.name** に組み込む必要があります。

11.4.1. CSV 生成の仕組み

Operator プロジェクトの **deploy**/ ディレクトリーは、Operator をデプロイするために必要なすべてのマニフェストの標準的な場所です。Operator SDK は **deploy**/ のマニフェストのデータを使用し、CSV を作成できます。以下がコマンドになります。

```
$ operator-sdk olm-catalog gen-csv --csv-version <version>
```

デフォルトで、CSV YAML ファイルを **deploy**/**olm-catalog**/ ディレクトリーに書き込みます。

3 つのタイプのマニフェストが CSV の生成に必要になります。

- **operator.yaml**
- ***_{crd,cr}.yaml**
- RBAC ロールファイル (例: **role.yaml**)

Operator の作者にはこれらのファイルについてそれぞれ異なるバージョン管理の要件がある場合があり、**deploy/olm-catalog/csv-config.yaml** ファイルに組み込む特定のファイルを設定できます。

ワークフロー
検出される既存の CSV に応じて、またすべての設定のデフォルト値が使用されることを仮定すると、**olm-catalog gen-csv** サブコマンドは以下のいずれかを実行します。

- 既存の場所および命名規則と同じ設定で、YAML マニフェストおよびソースファイルの利用可能なデータを使用して新規 CSV を作成します。
    a. 更新メカニズムは、**deploy/** で既存の CSV の有無をチェックします。これが見つからない場合、ここでは **キャッシュ** と呼ばれる ClusterServiceVersion オブジェクトを作成し、Kubernetes API **ObjectMeta** などの Operator メタデータから派生するフィールドを簡単に設定できます。
    b. 更新メカニズムは、**deploy/** で Deployment リソースなどの CSV が使用するデータが含まれるマニフェストを検索し、このデータを使ってキャッシュ内の該当する CSV フィールドを設定します。
    c. 検索が完了したら、設定されたすべてのキャッシュフィールドが CSV YAML ファイルに書き込まれます。

または、以下を実行します。

- YAML マニフェストおよびソースファイルで利用可能なデータを使用して、現時点で事前に定義されている場所で既存の CSV を更新します。
    a. 更新メカニズムは、**deploy/** で既存の CSV の有無をチェックします。これが見つかる場合、CSV YAML ファイルのコンテンツは ClusterServiceVersion キャッシュにマーシャルされます。
    b. 更新メカニズムは、**deploy/** で Deployment リソースなどの CSV が使用するデータが含まれるマニフェストを検索し、このデータを使ってキャッシュ内の該当する CSV フィールドを設定します。
    c. 検索が完了したら、設定されたすべてのキャッシュフィールドが CSV YAML ファイルに書き込まれます。

注記

ファイル全体ではなく、個別の YAML フィールドが上書きされます。 CSV の説明および他の生成されない部分が保持される必要があるためです。

11.4.2. CSV 構成の設定

Operator の作者者は、**deploy/olm-catalog/csv-config.yaml** ファイルでいくつかのフィールドを設定し、CSV の構成を設定できます。

フィールド	説明
operator-path (文字列)	Operator リソースマニフェストファイルのパス。デフォルトで **deploy/operator.yaml** に設定されます。
crd-cr-path-list (string(, string)*)	CRD および CR マニフェストファイルのパス。デフォルトで **[deploy/crds/*_{crd,cr}.yaml]** に設定されます。
rbac-path-list (string(, string)*)	RBAC ロールマニフェストファイルのパス。デフォルトで **[deploy/role.yaml]** に設定されます。

11.4.3. 手動で定義される CSV フィールド

数多くの CSV フィールドは、生成される SDK 固有のマニフェスト以外のファイルを使用して設定することができません。これらのフィールドは、ほとんどの場合、人間が作成する、Operator および各種のカスタムリソース定義 (CRD) についての英語のメタデータです。

Operator 作成者はそれらの CSV YAML ファイルを直接変更する必要があり、パーソナライズ設定されたデータを以下の必須フィールドに追加します。Operator SDK は、必須フィールドのいずれかにデータが欠落していることが検出されると、CSV 生成に関する警告を送信します。

表11.5 必須

フィールド	説明
metadata.name	CSV の固有名。Operator バージョンは、**app-operator.v0.1.1** などのように一意性を確保するために名前に含める必要があります。
spec.displayName	Operator を識別するためのパブリック名。
spec.description	Operator の機能についての簡単な説明。
spec.keywords	Operator について記述するキーワード。
spec.maintainers	**name** および **email** を持つ、Operator を維持する人または組織上のエンティティー
spec.provider	**name** を持つ、Operator のプロバイダー (通常は組織)
spec.labels	Operator 内部で使用されるキー/値のペア。
spec.version	Operator のセマンティクスバージョン。 例: **0.1.1**。

フィールド	説明
spec.customresourcedefinitions	Operator が使用する任意の CRD。このフィールドは、CRD YAML ファイルが **deploy/** にある場合に Operator SDK によって自動的に設定されます。ただし、CRD マニフェスト仕様にない複数のフィールドでは、ユーザーの入力が必要です。 • **description**: CRD の説明。 • **resources**: CRD によって利用される任意の Kubernetes リソース (例: Pod および StatefulSet)。 • **specDescriptors**: Operator の入力および出力についての UI ヒント。

表11.6 オプション

フィールド	説明
spec.replaces	この CSV によって置き換えられる CSV の名前。
spec.links	それぞれが **name** および **url** を持つ、Operator および管理されているアプリケーションに関する URL (例: Web サイトおよびドキュメント)。
spec.selector	Operator がクラスターでのリソースのペアの作成に使用するセレクター。
spec.icon	**mediatype** で **base64data** フィールドに設定される、Operator に固有の base64 でエンコーディングされるアイコン。
spec.maturity	Operator の成熟度モデルに応じた Operator の機能レベル (例: **Seamless Upgrades**)。

上記の各フィールドが保持するデータについての詳細は、「CSV spec」を参照してください。

注記

現時点でユーザーの介入を必要とするいくつかの YAML フィールドは、Operator コードから解析される可能性があります。 このような Operator SDK 機能は、今後の設計ドキュメントで扱われます。

追加リソース

- Operator 成熟度モデル

11.4.4. CSV の生成

前提条件

- Operator プロジェクトが Operator SDK を使用して生成されている

手順

1. Operator プロジェクトで、必要な場合に **deploy/olm-catalog/csv-config.yaml** ファイルを変更して CSV 構成を設定します。
2. CSV を生成します。

```
$ operator-sdk olm-catalog gen-csv --csv-version <version>
```

3. **deploy/olm-catalog/** ディレクトリーに生成される新規 CSV で、すべての必須で、手動で定義されたフィールドが適切に設定されていることを確認します。

11.4.5. カスタムリソース定義 (CRD)

Operator が使用できる以下の 2 つのタイプのカスタムリソース定義 (CRD) があります。1 つ目は Operator が所有する **所有** タイプと、もう 1 つは Operator が依存する **必須** タイプです。

11.4.5.1. 所有 CRD (Owned CRD)

Operator が所有する CRD は CSV の最も重要な部分です。これは Operator と必要な RBAC ルール間のリンク、依存関係の管理、および他の Kubernetes の概念を設定します。

Operator は通常、複数の CRD を使用して複数の概念を結び付けます (あるオブジェクトの最上位のデータベース設定と別のオブジェクトの ReplicaSet の表現など)。それぞれは CSV ファイルに一覧表示される必要があります。

表11.7 所有 CRD フィールド

フィールド	説明	必須/オプション
名前	CRD のフルネーム。	必須
Version	オブジェクト API のバージョン。	必須
Kind	CRD の機械可読名。	必須
DisplayName	CRD 名の人間が判読できるバージョン (例: **MongoDB Standalone**)。	必須
説明	Operator がこの CRD を使用する方法についての短い説明、または CRD が提供する機能の説明。	必須
Group	この CRD が所属する API グループ (例: **database.example.com**)。	オプション

フィールド	説明	必須/オプション
Resources	CRD が 1 つ以上の Kubernetes オブジェクトのタイプを所有する。これらは、トラブルシューティングが必要になる可能性のあるオブジェクトや、データベースを公開するサービスまたは Ingress ルールなどのアプリケーションに接続する方法についてユーザーに知らせるためにリソースセクションに一覧表示されます。 この場合、オーケストレーションするすべての一覧ではなく、重要なオブジェクトのみを一覧表示することが推奨されます。たとえば、ユーザーが変更できない内部状態を保存する ConfigMap はここには表示しません。	オプション
SpecDescriptors、**StatusDescriptors**、および **ActionDescriptors**	これらの記述子は、エンドユーザーにとって最も重要な Operator の入力および出力で UI にヒントを提供する手段になります。CRD にユーザーが指定する必要のあるシークレットまたは ConfigMap の名前が含まれる場合は、それをここに指定できます。これらのアイテムはリンクされ、互換性のある UI で強調表示されます。 記述子には、3 つの種類があります。 ● **SpecDescriptors**: オブジェクトの **spec** ブロックのフィールドへの参照。 ● **StatusDescriptors**: オブジェクトの **status** ブロックのフィールドへの参照。 ● **ActionDescriptors**: オブジェクトで実行できるアクションへの参照。 すべての記述子は以下のフィールドを受け入れます。 ● **DisplayName**: 仕様、ステータス、またはアクションの人間が判読できる名前。 ● **Description**: 仕様、ステータス、またはアクション、およびそれが Operator によって使用される方法についての短い説明。 ● **Path**: この記述子が記述するオブジェクトのフィールドのドットで区切られたパス。 ● **X-Descriptors**: この記述子が持つ「機能」および使用する UI コンポーネントを判別するために使用されます。OpenShift Container Platform の正規の React UI X-Descriptor の一覧については、**openshift/console** プロジェクトを参照してください。 記述子一般についての詳細は、**openshift/console** プロジェクトも参照してください。	オプション

以下の例は、シークレットおよび ConfigMap でユーザー入力を必要とし、サービス、StatefulSet、Pod および ConfigMap のオーケストレーションを行う **MongoDB Standalone** CRD を示しています。

所有 CRD の例

```
    - displayName: MongoDB Standalone
      group: mongodb.com
      kind: MongoDbStandalone
      name: mongodbstandalones.mongodb.com
      resources:
        - kind: Service
          name: ''
          version: v1
        - kind: StatefulSet
          name: ''
          version: v1beta2
        - kind: Pod
          name: ''
          version: v1
        - kind: ConfigMap
          name: ''
          version: v1
      specDescriptors:
        - description: Credentials for Ops Manager or Cloud Manager.
          displayName: Credentials
          path: credentials
          x-descriptors:
            - 'urn:alm:descriptor:com.tectonic.ui:selector:core:v1:Secret'
        - description: Project this deployment belongs to.
          displayName: Project
          path: project
          x-descriptors:
            - 'urn:alm:descriptor:com.tectonic.ui:selector:core:v1:ConfigMap'
        - description: MongoDB version to be installed.
          displayName: Version
          path: version
          x-descriptors:
            - 'urn:alm:descriptor:com.tectonic.ui:label'
      statusDescriptors:
        - description: The status of each of the Pods for the MongoDB cluster.
          displayName: Pod Status
          path: pods
          x-descriptors:
            - 'urn:alm:descriptor:com.tectonic.ui:podStatuses'
      version: v1
      description: >-
        MongoDB Deployment consisting of only one host. No replication of
        data.
```

11.4.5.2. 必須 CRD (Required CRD)

他の必須 CRD の使用は完全にオプションであり、これらは個別 Operator のスコープを縮小し、エンドツーエンドのユースケースに対応するために複数の Operator を一度に作成するために使用できます。

一例として、Operator がアプリケーションをセットアップし、分散ロックに使用する (etcd Operator からの) etcd クラスター、およびデータストレージ用に (Postgres Operator からの) Postgres データベースをインストールする場合があります。

Operator Lifecycle Manager (OLM) は、これらの要件を満たすためにクラスター内の利用可能な CRD および Operator に対してチェックを行います。適切なバージョンが見つかると、Operator は必要な

namespace 内で起動し、サービスアカウントが各 Operator が必要な Kubernetes リソースを作成し、監視し、変更できるようにするために作成されます。

表11.8 必須 CRD フィールド

フィールド	説明	必須/オプション
名前	必要な CRD のフルネーム。	必須
Version	オブジェクト API のバージョン。	必須
Kind	Kubernetes オブジェクトの種類。	必須
DisplayName	CRD の人間による可読可能なバージョン。	必須
説明	大規模なアーキテクチャーにおけるコンポーネントの位置付けについてのサマリー。	必須

必須 CRD の例

```
required:
- name: etcdclusters.etcd.database.coreos.com
  version: v1beta2
  kind: EtcdCluster
  displayName: etcd Cluster
  description: Represents a cluster of etcd nodes.
```

11.4.5.3. CRD テンプレート

Operator のユーザーは、どのオプションが必須またはオプションであるかを認識している必要があります。**alm-examples** という名前のアノテーションとして設定の最小セットを使用して、各 CRD のテンプレートを提供できます。互換性のある UI は、ユーザーがさらにカスタマイズできるようにこのテンプレートの事前入力を行います。

アノテーションは、**kind**の一覧で構成されます (例: CRD 名および Kubernetes オブジェクトの対応する **metadata** および **spec**)。

以下の詳細の例では、**EtcdCluster**、**EtcdBackup** および **EtcdRestore** のテンプレートを示しています。

```
metadata:
  annotations:
    alm-examples: >-
      [{"apiVersion":"etcd.database.coreos.com/v1beta2","kind":"EtcdCluster","metadata":
{"name":"example","namespace":"default"},"spec":{"size":3,"version":"3.2.13"}},
{"apiVersion":"etcd.database.coreos.com/v1beta2","kind":"EtcdRestore","metadata":
{"name":"example-etcd-cluster"},"spec":{"etcdCluster":{"name":"example-etcd-
cluster"},"backupStorageType":"S3","s3":{"path":"<full-s3-path>","awsSecret":"<aws-secret>"}}},
{"apiVersion":"etcd.database.coreos.com/v1beta2","kind":"EtcdBackup","metadata":
{"name":"example-etcd-cluster-backup"},"spec":{"etcdEndpoints":["<etcd-cluster-
endpoints>"],"storageType":"S3","s3":{"path":"<full-s3-path>","awsSecret":"<aws-secret>"}}}]
```

11.4.6. API サービスについて

CRD の場合のように、Operator が使用できる APIService の 2 つのタイプ（**所有 (owned)** および **必須 (required)**）があります。

11.4.6.1. 所有 APIService (Owned APIService)

CSV が APIService を所有する場合、CSV は APIService をサポートする拡張 **api-server** およびこれが提供する **group-version-kinds** のデプロイメントを記述します。

APIService はこれが提供する **group-version** によって一意に識別され、提供することが予想される複数の種類を示すために複数回一覧表示できます。

表11.9 所有 APIService フィールド

フィールド	説明	必須/オプション
Group	APIService が提供するグループ (**database.example.com**など)。	必須
Version	APIService のバージョン (**v1alpha1** など)。	必須
Kind	APIService が提供することが予想される種類。	必須
名前	指定された APIService の複数形の名前	必須
DeploymentName	APIService に対応する CSV で定義されるデプロイメントの名前 (所有 APIService に必要)。CSV が保留中のフェーズにある場合、OLM Operator は CSV の InstallStrategy で一致する名前を持つデプロイメント仕様を検索し、これが見つからない場合には、CSV をインストールの準備完了フェーズに移行しません。	必須
DisplayName	APIService 名の人間が判読できるバージョン (例: **MongoDB Standalone**)。	必須
説明	Operator がこの APIService を使用する方法についての短い説明、または APIService が提供する機能の説明。	必須
Resources	APIService は 1 つ以上の Kubernetes オブジェクトのタイプを所有します。これらは、トラブルシューティングが必要になる可能性のあるオブジェクトや、データベースを公開するサービスまたは Ingress ルールなどのアプリケーションに接続する方法についてユーザーに知らせるためにリソースセクションに一覧表示されます。 この場合、オーケストレーションするすべての一覧ではなく、重要なオブジェクトのみを一覧表示することが推奨されます。たとえば、ユーザーが変更できない内部状態を保存する ConfigMap はここには表示しません。	オプション

フィールド	説明	必須/オプション
SpecDescriptors、**StatusDescriptors**、および **ActionDescriptors**	所有 CRDと基本的に同じです。	オプション

11.4.6.1.1. APIService リソースの作成

Operator Lifecycle Manager (OLM) はそれぞれ固有の所有 APIService のサービスおよび APIService リソースを作成するか、またはこれらを置き換えます。

- サービス Pod セレクターは APIServiceDescription の **DeDeploymentName** に一致する CSV デプロイメントからコピーされます。
- 新規の CA キー/証明書ペアが各インストールについて生成され、base64 でエンコードされた CA バンドルがそれぞれの APIService リソースに組み込まれます。

11.4.6.1.2. APIService 提供証明書

OLM は、所有 APIService がインストールされるたびに、提供するキー/証明書のペアの生成を処理します。提供証明書には、生成されるサービスリソースのホスト名が含まれる CN が含まれ、これは対応する APIService リソースに組み込まれた CA バンドルのプライベートキーによって署名されます。

証明書は、デプロイメント namespace の **kubernetes.io/tls** タイプのシークレットとして保存され、**apiservice-cert** という名前のボリュームは、 APIServiceDescription の **DeploymentName** フィールドに一致する CSV のデプロイメントのボリュームセクションに自動的に追加されます。

存在していない場合、一致する名前を持つ VolumeMount もそのデプロイメントのすべてのコンテナーに追加されます。これにより、ユーザーは、カスタムパスの要件に対応するために、予想される名前のボリュームマウントを定義できます。生成される volumeMount のパスは **/apiserver.local.config/certificates** にデフォルト設定され、既存の volumeMounts が同じパスと置き換えられます。

11.4.6.2. 必須 APIService

OLM は、必要なすべての CSV に利用可能な APIService があり、すべての予想される **group-version-kinds** がインストールの試行前に検出可能であることを確認します。これにより、CSV は所有しない APIServices によって提供される特定の種類に依存できます。

表11.10 必須 APIService フィールド

フィールド	説明	必須/オプション
Group	APIService が提供するグループ (**database.example.com**など)。	必須
Version	APIService のバージョン (**v1alpha1** など)。	必須

フィールド	説明	必須/オプション
Kind	APIService が提供することが予想される種類。	必須
DisplayName	APIService 名の人間が判読できるバージョン (例: **MongoDB Standalone**)。	必須
説明	Operator がこの APIService を使用する方法についての短い説明、または APIService が提供する機能の説明。	必須

11.5. PROMETHEUS による組み込みモニタリングの設定

以下では、Prometheus Operator を使用して Operator SDK いよって提供されるビルトインされたモニタリングサポートについて説明し、Operator 作成者がどのように使用できるかについて詳しく説明します。

11.5.1. Prometheus Operator

Prometheus はオープンソースのシステムモニタリングおよびアラートツールキットです。Prometheus Operator は、 OpenShift Container Platform などの Kubernetes ベースのクラスターで実行される Prometheus クラスターを作成し、設定し、管理します。

ヘルパー関数は、デフォルトで Operator SDK に存在し、Prometheus Operator がデプロイされているクラスターで使用できるように生成された Go ベースの Operator にメトリクスを自動的にセットアップします。

11.5.2. メトリクスヘルパー

Operator SDK を使用して生成される Go ベース Operator では、以下の関数が実行中のプログラムについての一般的なメトリクスを公開します。

```
func ExposeMetricsPort(ctx context.Context, port int32) (*v1.Service, error)
```

これらのメトリクスは **controller-runtime** ライブラリー API から継承されます。メトリクスはデフォルトで **0.0.0.0:8383/metrics** で提供されます。

サービスオブジェクトは、メトリクスポートが公開された状態で作成されます。これはその後 Prometheus によってアクセスされます。 サービスオブジェクトは、リーダー Pod のルート所有者が削除されるとガベージコレクションの対象になります。

以下のサンプルは、Operator SDK を使用して生成されるすべての Operator の **cmd/manager/main.go** ファイルにあります。

```
import(
   "github.com/operator-framework/operator-sdk/pkg/metrics"
   "machine.openshift.io/controller-runtime/pkg/manager"
)

var (
   // Change the below variables to serve metrics on a different host or port.
   metricsHost       = "0.0.0.0" 1
   metricsPort int32 = 8383 2
```

```
)
...
func main() {
    ...
    // Pass metrics address to controller-runtime manager
    mgr, err := manager.New(cfg, manager.Options{
        Namespace:          namespace,
        MetricsBindAddress: fmt.Sprintf("%s:%d", metricsHost, metricsPort),
    })

    ...
    // Create Service object to expose the metrics port.
    _, err = metrics.ExposeMetricsPort(ctx, metricsPort)
    if err != nil {
        // handle error
        log.Info(err.Error())
    }
    ...
}
```

1. メトリクスの公開に使用されるホスト。
2. メトリクスの公開に使用されるポート。

11.5.2.1. メトリクスポートの変更

Operator の作成者は、メトリクスが公開されるポートを変更できます。

前提条件

- Operator SDK を使用して生成される Go ベースの Operator
- Prometheus Operator がデプロイされた Kubernetes ベースのクラスター

手順

- 生成された Operator の **cmd/manager/main.go** ファイルで、 **var metricsPort int32 = 8383** 行の **metricsPort** の値を変更します。

11.5.3. ServiceMonitor リソース

ServiceMonitor は、Prometheus Operator によって提供されるカスタマーリソース定義 (CRD) であり、サービスオブジェクトで **Endpoints** を検出し、Prometheus がこれらの Pod を監視するように設定します。

Operator SDK を使用して生成される Go ベースの Operator では、 **GenerateServiceMonitor()** ヘルパー関数がサービスオブジェクトを取り、これに基づいて ServiceMonitor カスタムリソース (CR) を生成することができます。

追加リソース

- ServiceMonitor CRD についての詳細は、Prometheus Operator のドキュメント を参照してください。

11.5.3.1. ServiceMonitor リソースの作成

Operator の作成者は、新規に作成されるサービスを受け入れる **metrics.CreateServiceMonitor()** ヘルパー関数を使用して、作成されたモニタリングサービスのサービスターゲット検出を追加できます。

前提条件

- Operator SDK を使用して生成される Go ベースの Operator
- Prometheus Operator がデプロイされた Kubernetes ベースのクラスター

手順

- **metrics.CreateServiceMonitor()** ヘルパー関数を Operator コードに追加します。

```
import(
   "k8s.io/api/core/v1"
   "github.com/operator-framework/operator-sdk/pkg/metrics"
   "machine.openshift.io/controller-runtime/pkg/client/config"
)
func main() {

   ...
   // Populate below with the Service(s) for which you want to create ServiceMonitors.
   services := []*v1.Service{}
   // Create one ServiceMonitor per application per namespace.
   // Change the below value to name of the Namespace you want the ServiceMonitor to be
created in.
   ns := "default"
   // restConfig is used for talking to the Kubernetes apiserver
   restConfig := config.GetConfig()

   // Pass the Service(s) to the helper function, which in turn returns the array of
ServiceMonitor objects.
   serviceMonitors, err := metrics.CreateServiceMonitors(restConfig, ns, services)
   if err != nil {
      // Handle errors here.
   }
   ...
}
```

11.6. リーダー選択の設定

Operator のライフサイクル中は、いずれかの時点で複数のインスタンスが実行される可能性があります。たとえば、Operator のアップグレードをロールアウトしている場合などがこれに含まれます。この場合、リーダー選択を使用して複数の Operator 間の競合を避ける必要があります。 これにより、1 つのリーダーインスタンスのみが調整を行い、他のインスタンスは非アクティブな状態であるものの、リーダーがその役割を実行しなくなる場合に引き継げる状態にできます。

2 種類のリーダー選択の実装を選択できますが、それぞれに考慮すべきトレードオフがあります。

- **Leader-for-life**: リーダー Pod は削除される場合のみリーダーシップを放棄します (ガベージコレクションを使用)。この実装は 2 つのインスタンスが誤ってリーダーとして実行されるのを防ぎます (スプリットブレイン)。しかし、この方法では、新規リーダーの選択に遅延が生じる可能性があります。たとえば、リーダー Pod が応答しないノードまたはパーティション化された

ノードにある場合、**pod-eviction-timeout** はリーダー Pod がノードから削除され、リーダーシップを中止するまでの時間を判別します（デフォルトは **5m**）。詳細は、Leader-for-life Go ドキュメントを参照してください。

- **Leader-with-lease**: リーダー Pod は定期的にリーダーリースを更新し、リースを更新できない場合にリーダーシップを放棄します。この実装により、既存リーダーが分離される場合に新規リーダーへの迅速な移行が可能になりますが、スピリットブレインが特定の状況で生じる場合があります。詳細は、Leader-with-lease Go ドキュメントを参照してください。

デフォルトで、Operator SDK は Leader-for-life 実装を有効にします。実際のユースケースに適した選択ができるように両方のアプローチのトレードオフについて、関連する Go ドキュメントを参照してください。

以下の例は、これらの 2 つのオプションを使用する方法について説明しています。

11.6.1. Leader-for-life 選択の使用

Leader-for-life 選択の実装の場合、**leader.Become()** の呼び出しは、**memcached-operator-lock** という名前の ConfigMap を作成して、リーダー選択までの再試行中に Operator をブロックします。

```
import (
  ...
  "github.com/operator-framework/operator-sdk/pkg/leader"
)

func main() {
  ...
  err = leader.Become(context.TODO(), "memcached-operator-lock")
  if err != nil {
    log.Error(err, "Failed to retry for leader lock")
    os.Exit(1)
  }
  ...
}
```

Operator がクラスター内で実行されていない場合、 **leader.Become()** はエラーなしに返し、Operator の namespace を検出できないことからリーダー選択をスキップします。

11.6.2. Leader-with-lease 選択の使用

Leader-with-lease 実装は、リーダー選択について Manager オプションを使用して有効にできます。

```
import (
  ...
  "sigs.k8s.io/controller-runtime/pkg/manager"
)

func main() {
  ...
  opts := manager.Options{
    ...
    LeaderElection: true,
    LeaderElectionID: "memcached-operator-lock"
  }
```

```
  mgr, err := manager.New(cfg, opts)
  ...
}
```

Operator がクラスターで実行されていない場合、Manager はリーダー選択用の ConfigMap を作成するための Operator の namespace を検出できないことから開始時にエラーを返します。Manager の **LeaderElectionNamespace** オプションを設定してこの namespace を上書きできます。

11.7. OPERATOR SDK CLI リファレンス

以下では、Operator SDK CLI コマンドおよびそれらの構文について説明します。

```
$ operator-sdk <command> [<subcommand>] [<argument>] [<flags>]
```

11.7.1. build

operator-sdk build コマンドはコードをコンパイルし、実行可能プロジェクトをビルドします。**build** が完了すると、イメージは **docker** でローカルにビルドされます。これは次にリモートレジストリーにプッシュされる必要があります。

表11.11 build 引数

引数	説明
<image>	ビルトされるコンテナーイメージ (例: **quay.io/example/operator:v0.0.1**)。

表11.12 build フラグ

フラグ	説明
--enable-tests (ブール)	テストバイナリーをイメージに追加することにより、クラスター内でのテストを有効にします。
--namespaced-manifest (文字列)	テスト用の namespace を使用したリソースマニフェストのパス。デフォルト: **deploy/operator.yaml**
--test-location (文字列)	テストの場所。デフォルト: **./test/e2e**
-h, --help	使用方法についてのヘルプの出力。

--enable-tests が設定される場合、**build** コマンドはテストバイナリーもビルドし、これをコンテナーイメージに追加して、ユーザーがテストをクラスター上で Pod として実行できるように **deploy/test-pod.yaml** ファイルを生成します。

出力例

```
$ operator-sdk build quay.io/example/operator:v0.0.1

building example-operator...
```

```
building container quay.io/example/operator:v0.0.1...
Sending build context to Docker daemon  163.9MB
Step 1/4 : FROM alpine:3.6
 ---> 77144d8c6bdc
Step 2/4 : ADD tmp/_output/bin/example-operator /usr/local/bin/example-operator
 ---> 2ada0d6ca93c
Step 3/4 : RUN adduser -D example-operator
 ---> Running in 34b4bb507c14
Removing intermediate container 34b4bb507c14
 ---> c671ec1cff03
Step 4/4 : USER example-operator
 ---> Running in bd336926317c
Removing intermediate container bd336926317c
 ---> d6b58a0fcb8c
Successfully built d6b58a0fcb8c
Successfully tagged quay.io/example/operator:v0.0.1
```

11.7.2. completion

operator-sdk completion コマンドは、CLI コマンドをより迅速に、より容易に実行できるようにシェル補完を生成します。

表11.13 completion サブコマンド

サブコマンド	説明
bash	bash 補完を生成します。
zsh	zsh 補完を生成します。

表11.14 completion フラグ

フラグ	説明
-h, --help	使用方法についてのヘルプの出力。

出力例

```
$ operator-sdk completion bash

# bash completion for operator-sdk                  -*- shell-script -*-
...
# ex: ts=4 sw=4 et filetype=sh
```

11.7.3. print-deps

operator-sdk print-deps コマンドは、Operator が必要とする最新の Golang パッケージおよびバージョンを出力します。これはデフォルトで単票形式 (columnar format) の出力を行います。

表11.15 print-deps フラグ

フラグ	説明
--as-file	**Gopkg.toml** 形式でパッケージおよびバージョンを出力します。

出力例

```
$ operator-sdk print-deps --as-file
required = [
  "k8s.io/code-generator/cmd/defaulter-gen",
  "k8s.io/code-generator/cmd/deepcopy-gen",
  "k8s.io/code-generator/cmd/conversion-gen",
  "k8s.io/code-generator/cmd/client-gen",
  "k8s.io/code-generator/cmd/lister-gen",
  "k8s.io/code-generator/cmd/informer-gen",
  "k8s.io/code-generator/cmd/openapi-gen",
  "k8s.io/gengo/args",
]

[[override]]
  name = "k8s.io/code-generator"
  revision = "6702109cc68eb6fe6350b83e14407c8d7309fd1a"
...
```

11.7.4. generate

operator-sdk generate コマンドは特定のジェネレーターを起動して、必要に応じてコードを生成します。

表11.16 generate サブコマンド

サブコマンド	説明
k8s	すべての CRD API の Kubernetes code-generators を **pkg/apis/** の下に実行します。現時点で、**k8s** は **deepcopy-gen** のみを実行し、すべてのカスタムリソース (CR) タイプに必要な **DeepCopy()** 関数を生成します。

注記

このコマンドは、カスタムリソースの API (**spec** および **status**) が更新されるたびに実行される必要があります。

出力例

```
$ tree pkg/apis/app/v1alpha1/
pkg/apis/app/v1alpha1/
├── appservice_types.go
├── doc.go
├── register.go

$ operator-sdk generate k8s
Running code-generation for Custom Resource (CR) group versions: [app:v1alpha1]
```

```
Generating deepcopy funcs

$ tree pkg/apis/app/v1alpha1/
pkg/apis/app/v1alpha1/
├── appservice_types.go
├── doc.go
├── register.go
└── zz_generated.deepcopy.go
```

11.7.5. olm-catalog

operator-sdk olm-catalog は、すべての Operator Lifecycle Manager (OLM) Catalog 関連コマンドの親コマンドです。

11.7.5.1. gen-csv

gen-csv サブコマンドは、Cluster Service Version (CSV) マニフェスト、およびオプションでカスタムリソースリソース定義 (CRD) ファイルを **deploy/olm-catalog/<operator_name>/<csv_version>** に書き込みます。

表11.17 olm-catalog gen-csv フラグ

フラグ	説明
--csv-version (文字列)	CSV マニフェストのセマンティックバージョン。必須。
--from-version (文字列)	新規バージョンのベースとして使用する CSV マニフェストのセマンティックバージョン。
--csv-config (文字列)	CSV 設定ファイルへのパス。デフォルト: **deploy/olm-catalog/csv-config.yaml**。
--update-crds	最新の CRD マニフェストを使用して **deploy/<operator_name>/<csv_version>** で CRD マニフェストを更新します。

出力例

```
$ operator-sdk olm-catalog gen-csv --csv-version 0.1.0 --update-crds
INFO[0000] Generating CSV manifest version 0.1.0
INFO[0000] Fill in the following required fields in file deploy/olm-catalog/operator-name/0.1.0/operator-name.v0.1.0.clusterserviceversion.yaml:
 spec.keywords
 spec.maintainers
 spec.provider
 spec.labels
INFO[0000] Created deploy/olm-catalog/operator-name/0.1.0/operator-name.v0.1.0.clusterserviceversion.yaml
```

11.7.6. new

operator-sdk new コマンドは新規の Operator アプリケーションを作成し、入力された **<project_name>** に基づいてデフォルトのプロジェクトディレクトリーのレイアウトの生成 (または **スキャフォールディング**) を実行します。

表11.18 new 引数

引数	説明
<project_name>	新規プロジェクトの名前。

表11.19 new フラグ

フラグ	説明
--api-version	**$GROUP_NAME/$VERSION** 形式の CRD **APIVersion** (例: **app.example.com/v1alpha1**)。**ansible** または **helm** タイプで使用されます。
--dep-manager [dep|modules]	新規プロジェクトが使用する依存関係マネージャー。 **go** タイプで使用されます。(デフォルト: **modules**)
--generate-playbook	Ansible Playbook のスケルトンを生成します。**ansible** タイプで使用されます。
--header-file <string>	生成される Go ファイルのヘッダーを含むファイルへのパスです。**hack/boilerplate.go.txt** にコピーされます。
--helm-chart <string>	既存の Helm チャートで Helm Operator を初期化します。**<url>**、**<repo>/<name>** 、またはローカルパス。
--helm-chart-repo <string>	要求される Helm チャートのチャートリポジトリー URL。
--helm-chart-version <string>	Helm チャートの特定バージョン。(デフォルト: latest version)
--help, -h	使用方法およびヘルプの出力。
--kind <string>	CRD **Kind** (例: **AppService**)。**ansible** または **helm** タイプで使用されます。
--skip-git-init	ディレクトリーを Git リポジトリーとして実行しません。
--type	初期化する Operator のタイプ: **go**、 **ansible** または **helm**。(デフォルト: **go**)

Go プロジェクトの使用例

```
$ mkdir $GOPATH/src/github.com/example.com/
$ cd $GOPATH/src/github.com/example.com/
$ operator-sdk new app-operator
```

Ansible プロジェクトの使用例

```
$ operator-sdk new app-operator \
    --type=ansible \
    --api-version=app.example.com/v1alpha1 \
    --kind=AppService
```

11.7.7. add

operator-sdk add コマンドは、コントローラーまたはリソースをプロジェクトに追加します。コマンドは、Operator プロジェクトのルートディレクトリーから実行される必要があります。

表11.20 add サブコマンド

サブコマンド	説明
api	新規カスタムリソース (CR) の新規 API 定義を **pkg/apis** の下に追加し、カスタムリソース定義 (CRD) およびカスタムリソース (CR) ファイルを **deploy/crds/** の下に生成します。API が **pkg/apis/<group>/<version>** にすでにある場合には、コマンドは上書きせず、エラーを返します。
controller	新規コントローラーを **pkg/controller/<kind>/** の下に追加します。コントローラーは **operator-sdk add api --kind=<kind> --api-version=<group/version>** コマンドで **pkg/apis/<group>/<version>** の下にすでに定義されている必要のある CR タイプを使用することを予想します。その **Kind** のコントローラーパッケージが **pkg/controller/<kind>** にすでに存在する場合、コマンドは上書きせず、エラーが返されます。
crd	CRD および CR ファイルを追加します。**<project-name>/deploy** パスがすでに存在している必要があります。**--api-version** および **--kind** フラグが、新規 Operator アプリケーションを生成するために必要です。 ● 生成される CRD ファイル名: **<project-name>/deploy/crds/<group>_<version>_<kind>_crd.yaml** ● 生成される CR ファイル名: **<project-name>/deploy/crds/<group>_<version>_<kind>_cr.yaml**

表11.21 add api フラグ

フラグ	説明
--api-version (文字列)	**$GROUP_NAME/$VERSION** 形式の CRD **APIVersion** (例: **app.example.com/v1alpha1**)。
--image (文字列)	CRD **Kind** (例: **AppService**)。

add api 出力サンプル

```
$ operator-sdk add api --api-version app.example.com/v1alpha1 --kind AppService
Create pkg/apis/app/v1alpha1/appservice_types.go
```

```
Create pkg/apis/addtoscheme_app_v1alpha1.go
Create pkg/apis/app/v1alpha1/register.go
Create pkg/apis/app/v1alpha1/doc.go
Create deploy/crds/app_v1alpha1_appservice_cr.yaml
Create deploy/crds/app_v1alpha1_appservice_crd.yaml
Running code-generation for Custom Resource (CR) group versions: [app:v1alpha1]
Generating deepcopy funcs

$ tree pkg/apis
pkg/apis/
├── addtoscheme_app_appservice.go
├── apis.go
└── app
    └── v1alpha1
        ├── doc.go
        ├── register.go
        ├── types.go
```

add controller 出力サンプル

```
$ operator-sdk add controller --api-version app.example.com/v1alpha1 --kind AppService
Create pkg/controller/appservice/appservice_controller.go
Create pkg/controller/add_appservice.go

$ tree pkg/controller
pkg/controller/
├── add_appservice.go
├── appservice
│   └── appservice_controller.go
└── controller.go
```

add crd 出力サンプル

```
$ operator-sdk add crd --api-version app.example.com/v1alpha1 --kind AppService
Generating Custom Resource Definition (CRD) files
Create deploy/crds/app_v1alpha1_appservice_crd.yaml
Create deploy/crds/app_v1alpha1_appservice_cr.yaml
```

11.7.8. test

operator-sdk test コマンドは Operator をローカルでテストできます。

11.7.8.1. local

local サブコマンドは、Operator SDK のテストフレームワークを使用してビルドされた Go テストをローカルで実行します。

表11.22 test local 引数

引数	説明
<test_location> (文字列)	e2e テストファイルの場所 (例: **./test/e2e/**)。

表11.23 test local フラグ

フラグ	説明
--kubeconfig (文字列)	クラスターの **kubeconfig** の場所。デフォルト: **~/.kube/config**。
--global-manifest (文字列)	グローバルリソースのマニフェストへのパス。デフォルト: **deploy/crd.yaml**。
--namespaced-manifest (文字列)	テスト別の namespace を使用したリソースのマニフェストへのパス。デフォルト: **deploy/service_account.yaml**、**deploy/rbac.yaml**、および **deploy/operator.yaml** の組み合わせ。
--namespace (文字列)	空ではない場合、テストを実行する単一の namespace (例: **operator-test**)。デフォルト: **""**
--go-test-flags (string)	**go test** に渡す追加の引数 (例:**-f "-v -parallel=2"**)。
--up-local	クラスターのイメージとしてではなく、**go run** を使用した Operator のローカルの実行を有効にします。
--no-setup	テストリソースの作成を無効にします。
--image (文字列)	namespace を使用したマニフェストで指定されたイメージとは異なる Operator イメージを使用します。
-h, --help	使用方法についてのヘルプの出力。

出力例

```
$ operator-sdk test local ./test/e2e/

# Output:
ok   github.com/operator-framework/operator-sdk-samples/memcached-operator/test/e2e 20.410s
```

11.7.9. up

operator-sdk up コマンドには、様々な方法で Operator を実行できるサブコマンドが含まれます。

11.7.9.1. local

local サブコマンドは、**kubeconfig** ファイルを使用して Kubernetes クラスターにアクセスできる機能を使って Operator バイナリーをビルドし、Operator をローカルマシンで起動します。

表11.24 up local 引数

引数	説明
--kubeconfig (文字列)	Kubernetes 設定ファイルへのファイルパス。デフォルト: **$HOME/.kube/config**

引数	説明
--namespace (文字列)	Operator が変更の有無を監視する namespace。デフォルト: **default**
--operator-flags	ローカル Operator が必要とする可能性のあるフラグ。例: **--flag1 value1 --flag2=value2**
-h, --help	使用方法についてのヘルプの出力。

出力例

```
$ operator-sdk up local \
  --kubeconfig "mycluster.kubecfg" \
  --namespace "default" \
  --operator-flags "--flag1 value1 --flag2=value2"
```

以下の例では、デフォルトの **kubeconfig**、デフォルトの namespace 環境変数を使用し、Operator のフラグで渡します。Operator フラグを使用するには、Operator がこのオプションの処理方法を認識している必要があります。たとえば、**resync-interval** フラグを認識する Operator の場合は、以下を実行します。

```
$ operator-sdk up local --operator-flags "--resync-interval 10"
```

デフォルト以外の namespace を使用することを予定している場合は、 **--namespace** フラグを使用して、Operator が作成されるカスタムリソース (CR) を監視する場所を変更します。

```
$ operator-sdk up local --namespace "testing"
```

これが機能させるには、Operator が **WATCH_NAMESPACE** 環境変数を処理する必要があります。これは、Operator に ユーティリティー機能 の**k8sutil.GetWatchNamespace** を使用して実行できます。

11.8. 付録

11.8.1. Operator プロジェクトのスキャフォールディングレイアウト

operator-sdk CLI は、それぞれの Operator プロジェクトに多数のパッケージを生成します。以下のセクションには、生成される各ファイルおよびディレクトリーの基本的な要約が含まれます。

11.8.1.1. Go ベースプロジェクト

operator-sdk new コマンドを使用して生成される Go ベースの Operator プロジェクト (デフォルトタイプ) には、以下のディレクトリーおよびファイルが含まれます。

ファイル/フォルダー	目的

ファイル/フォルダー	目的
cmd/	Operator のメインプログラムである **manager/main.go** ファイルが含まれます。これは Operator の主なプログラムです。 これは、すべてのカスタムリソース定義を **pkg/apis/** の下に定義し、すべてのコントローラーを **pkg/controllers/** の下で起動する新規マネージャーをインスタンス化します。
pkg/apis/	カスタムリソース定義 (CRD) の API を定義するディレクトリーツリーが含まれます。ユーザーは **pkg/apis/<group>/<version>/<kind>_types.go** ファイルを編集し、各リソースタイプの API を定義し、それらのパッケージをコントローラーにインポートしてリソースタイプの有無について監視することが想定されます。
pkg/controller	この **pkg** には、コントローラーの実装が含まれます。ユーザーは **pkg/controller/<kind>/<kind>_controller.go** ファイルを編集し、指定された **kind** のリソースタイプを処理するためのコントローラーの調整 (reconciliation) ロジックを定義することが想定されます。
build/	Operator をビルドするために使用される **Dockerfile** およびビルドスクリプトが含まれます。
deploy/	CRD を登録し、RBAC をセットアップし、Deployment として Operator をデプロイするための各種 YAML マニフェストが含まれます。
Gopkg.toml **Gopkg.lock**	この Operator の外部の依存関係を記述する Go Dep マニフェスト。
vendor/	このプロジェクトのインポートの条件を満たす外部の依存関係のローカルコピーが含まれる golang vendor フォルダー。Go Dep はベンダーを直接管理します。

11.8.1.2. Helm ベースのプロジェクト

operator-sdk new --type helm コマンドを使用して生成される Helm ベース Operator プロジェクトには、以下のディレクトリーおよびファイルが含まれます。

ファイル/フォルダー	目的
deploy/	CRD を登録し、RBAC をセットアップし、Deployment として Operator をデプロイするための各種 YAML マニフェストが含まれます。
helm-charts/<kind>	**helm create** と同等のコマンドを使用して初期化された Helm チャートが含まれます。
build/	Operator をビルドするために使用される **Dockerfile** およびビルドスクリプトが含まれます。

ファイル/フォルダー	目的
watches.yaml	**Group**、**Version**、 **Kind**、および Helm チャートの場所が含まれます。