OpenShift Dedicated 4

Operator

OpenShift Dedicated の Operator

Last Updated: 2024-04-17

OpenShift Dedicated 4 Operator

OpenShift Dedicated の Operator

法律上の通知

概要

コントロールプレーンでのサービスのパッケージ化、デプロイメント、管理に Operator がどのように役立つかを説明します。

目次

第1章 OPERATOR の概要

Operator は OpenShift Dedicated の最も重要なコンポーネントです。Operator はコントロールプレーンでサービスをパッケージ化し、デプロイし、管理するための優先される方法です。Operator の使用は、ユーザーが実行するアプリケーションにも各種の利点があります。

Operator は **kubectl** や **oc** コマンドなどの Kubernetes API および CLI ツールと統合します。Operator はアプリケーションの監視、ヘルスチェックの実行、OTA (over-the-air) 更新の管理を実行し、アプリケーションが指定した状態にあることを確認するための手段となります。

OpenShift Dedicated の Operator は、目的に応じて 2 つの異なるシステムによって管理されます。どちらも同様の Operator の概念と目標に準拠しています。

- Cluster Version Operator (CVO) によって管理されるクラスター Operator は、クラスター機能を実行するためにデフォルトでインストールされます。
- Operator Lifecycle Manager (OLM) によって管理されるオプションのアドオン Operator は、ユーザーがアプリケーションで実行できるようにアクセスできるようにすることができます。

Operator を使用すると、クラスター内で実行中のサービスを監視するアプリケーションを作成できます。Operator は、アプリケーション専用に設計されています。Operator は、インストールや設定などの一般的な Day 1 の操作と、自動スケーリングやバックアップの作成などの Day 2 の操作を実装および自動化します。これらのアクティビティーはすべて、クラスター内で実行されているソフトウェアの一部です。

1.1. 開発者の場合

開発者は、次の Operator タスクを実行できます。

- Operator SDK CLI をインストール します。
- Go ベースの Operator、Ansible ベースの Operator、および Helm ベースの Operator を作成します。
- Operator SDK を使用して Operator をビルド、テスト、デプロイ します。
- インストールされた Operator から Web コンソールを介してアプリケーションを作成 します。

1.2. 管理者の場合

dedicated-admin ロールを持つ管理者は、次の Operator タスクを実行できます。

- カスタムカタログを管理 します。
- OperatorHub から Operator をインストール します。
- オペレータのステータスを表示 します。
- Operator の状態を管理 します。
- インストールされている Operator をアップグレード します。
- インストールされている Operator を削除 します。
- プロキシーサポートを設定 します。

1.3. 次のステップ

Operator の詳細はOperator とはを参照してください。

第2章 OPERATOR について

2.1. OPERATOR について

概念的に言うと、**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設定されます。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一員のように機能し、Kubernetes 環境 (OpenShift Dedicated など) を監視し、その現在の状態を使用してリアルタイムで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。

技術的に言うと、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デプロイメントする場所。

Kubernetes にデプロイする理由

Kubernetes (ひいては OpenShift Dedicated) には、オンプレミスとクラウドプロバイダーで動作する複雑な分散システム (シークレット処理、負荷分散、サービスディスカバリー、自動スケーリング) を構築するために必要なすべての基本要素が含まれています。

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、Operator もこれらのクラスター外のサービスに使用できますが、これらをサービスブローカーで使用できます。

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

Operator SDK 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

Operator Lifecycle Manager (OLM) は、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 Dedicated 4 ではデフォルトでデプロイされます。

Operator レジストリー

Operator レジストリーは、クラスターで作成するためのクラスターサービスバージョン (Cluster Service Version、CSV) およびカスタムリソース定義 (CRD) を保存し、パッケージおよびチャネルについての Operator メタデータを保存します。これは Kubernetes または OpenShift クラスターで実行され、この Operator カタログデータを OLM に指定します。

OperatorHub

OperatorHub は、クラスター管理者がクラスター上にインストールする Operator を検出し、選択するための Web コンソールです。OpenShift Dedicated ではデフォルトでデプロイされます。

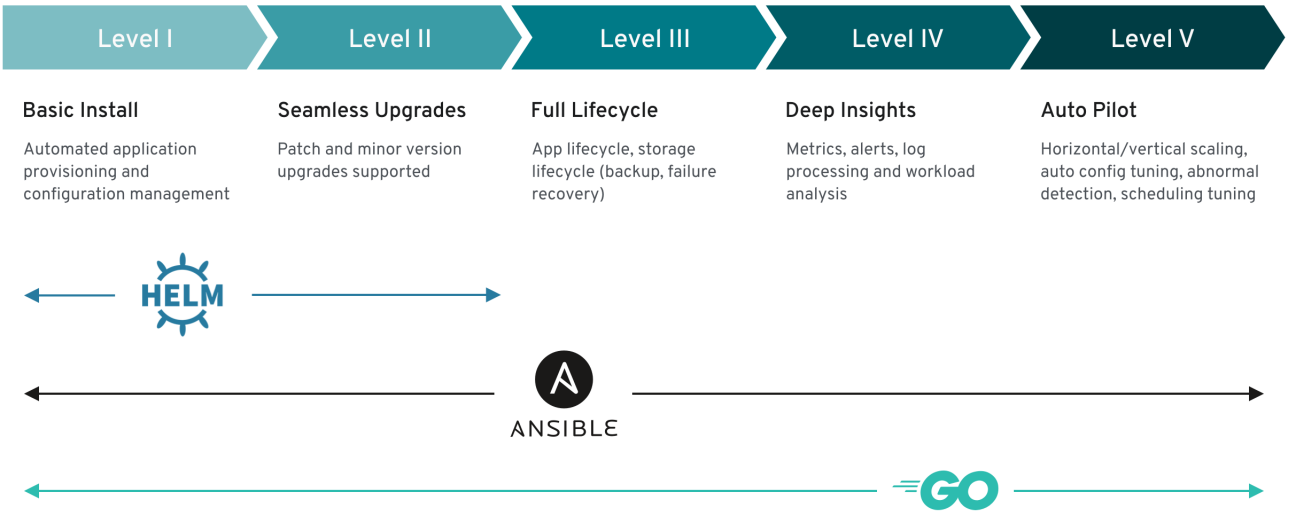
これらのツールは組み立て可能なツールとして設計されているため、役に立つと思われるツールを使用できます。

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の規模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 FRAMEWORK パッケージ形式

ここでは、OpenShift Dedicated の Operator Lifecycle Manager (OLM) によってサポートされる Operator のパッケージ形式の概要を説明します。

2.2.1. Bundle Format

Operator の **Bundle Format** は、Operator Framework によって導入されるパッケージ形式です。スケーラビリティーを向上させ、アップストリームユーザーがより効果的に独自のカタログをホストできるようにするために、Bundle Format 仕様は Operator メタデータのディストリビューションを単純化します。

Operator バンドルは、Operator の単一バージョンを表します。ディスク上の バンドルマニフェストは、Kubernetes マニフェストおよび Operator メタデータを保存する実行不可能なコンテナーイメージである バンドルイメージ としてコンテナー化され、提供されます。次に、バンドルイメージの保存および配布は、**podman**、**docker**、および Quay などのコンテナーレジストリーを使用して管理されます。

Operator メタデータには以下を含めることができます。

- Operator を識別する情報 (名前およびバージョンなど)。
- UI を駆動する追加情報 (アイコンや一部のカスタムリソース (CR) など)。
- 必須および提供される API。
- 関連するイメージ。

マニフェストを Operator レジストリーデータベースに読み込む際に、以下の要件が検証されます。

- バンドルには、アノテーションで定義された 1 つ以上のチャネルが含まれる必要がある。

- すべてのバンドルには、1 つのクラスターサービスバージョン (CSV) がある。
- CSV がクラスターリソース定義 (CRD) を所有する場合、その CRD はバンドルに存在する必要がある。

2.2.1.1. マニフェスト

バンドルマニフェストは、Operator のデプロイメントおよび RBAC モデルを定義する Kubernetes マニフェストのセットを指します。

バンドルにはディレクトリーごとに 1 つの CSV が含まれ、通常は **manifest/** ディレクトリーの CSV の所有される API を定義する CRD が含まれます。

Bundle Format のレイアウトの例

```
etcd
├── manifests
│   ├── etcdcluster.crd.yaml
│   └── etcdoperator.clusterserviceversion.yaml
│   └── secret.yaml
│   └── configmap.yaml
└── metadata
    └── annotations.yaml
    └── dependencies.yaml
```

その他のサポート対象のオブジェクト
以下のオブジェクトタイプは、バンドルの **/manifests** ディレクトリーにオプションとして追加することもできます。

サポート対象のオプションオブジェクトタイプ

- **ClusterRole**
- **clusterRoleBinding**
- **ConfigMap**
- **ConsoleCLIDownload**
- **ConsoleLink**
- **ConsoleQuickStart**
- **ConsoleYamlSample**
- **PodDisruptionBudget**
- **PriorityClass**
- **PrometheusRule**
- **Role**
- **RoleBinding**
- **Secret**

- **Service**
- **ServiceAccount**
- **ServiceMonitor**
- **VerticalPodAutoscaler**

これらのオプションオブジェクトがバンドルに含まれる場合、Operator Lifecycle Manager (OLM) はバンドルからこれらを作成し、CSV と共にそれらのライフサイクルを管理できます。

オプションオブジェクトのライフサイクル

- CSV が削除されると、OLM はオプションオブジェクトを削除します。
- CSV がアップグレードされると、以下を実行します。
  - オプションオブジェクトの名前が同じである場合、OLM はこれを更新します。
  - オプションオブジェクトの名前がバージョン間で変更された場合、OLM はこれを削除し、再作成します。

2.2.1.2. アノテーション

バンドルには、その **metadata/** ディレクトリーに **annotations.yaml** ファイルも含まれます。このファイルは、バンドルをバンドルのインデックスに追加する方法についての形式およびパッケージ情報の記述に役立つ高レベルの集計データを定義します。

annotations.yaml の例

```
annotations:
  operators.operatorframework.io.bundle.mediatype.v1: "registry+v1" (1)
  operators.operatorframework.io.bundle.manifests.v1: "manifests/" (2)
  operators.operatorframework.io.bundle.metadata.v1: "metadata/" (3)
  operators.operatorframework.io.bundle.package.v1: "test-operator" (4)
  operators.operatorframework.io.bundle.channels.v1: "beta,stable" (5)
  operators.operatorframework.io.bundle.channel.default.v1: "stable" (6)
```

(1) Operator バンドルのメディアタイプまたは形式。**registry+v1** 形式の場合、これに CSV および関連付けられた Kubernetes オブジェクトが含まれることを意味します。

(2) Operator マニフェストが含まれるディレクトリーへのイメージのパス。このラベルは今後使用するために予約され、現時点ではデフォの **manifests/** に設定されています。**manifests.v1** の値は、バンドルに Operator マニフェストが含まれることを示します。

(3) バンドルについてのメタデータファイルが含まれるディレクトリーへのイメージのパス。このラベルは今後使用するために予約され、現時点ではデフォの **metadata/** に設定されています。**metadata.v1** の値は、このバンドルに Operator メタデータがあることを意味します。

(4) バンドルのパッケージ名。

(5) Operator レジストリーに追加される際にバンドルがサブスクライブするチャネルのリスト。

(6) レジストリーからインストールされる場合に Operator がサブスクライブされるデフォルトチャネル。

注記

一致しない場合、**annotations.yaml** ファイルは、これらのアノテーションに依存するクラスター上の Operator レジストリーのみがこのファイルにアクセスできるために権威を持つファイルになります。

2.2.1.3. Dependencies

Operator の依存関係は、バンドルの **metadata/** フォルダー内の **dependencies.yaml** ファイルに一覧表示されます。このファイルはオプションであり、現時点では明示的な Operator バージョンの依存関係を指定するためにのみ使用されます。

依存関係の一覧には、依存関係の内容を指定するために各項目の **type** フィールドが含まれます。次のタイプの Operator 依存関係がサポートされています。

olm.package

このタイプは、特定の Operator バージョンの依存関係であることを意味します。依存関係情報には、パッケージ名とパッケージのバージョンを semver 形式で含める必要があります。たとえば、**0.5.2** などの特定バージョンや **>0.5.1** などのバージョンの範囲を指定することができます。

olm.gvk

このタイプの場合、作成者は CSV の既存の CRD および API ベースの使用方法と同様に group/version/kind (GVK) 情報で依存関係を指定できます。これは、Operator の作成者がすべての依存関係、API または明示的なバージョンを同じ場所に配置できるようにするパスです。

olm.constraint

このタイプは、任意の Operator プロパティーに対するジェネリック制約を宣言します。

以下の例では、依存関係は Prometheus Operator および etcd CRD について指定されます。

dependencies.yaml ファイルの例

```
dependencies:
  - type: olm.package
    value:
      packageName: prometheus
      version: ">0.27.0"
  - type: olm.gvk
    value:
      group: etcd.database.coreos.com
      kind: EtcdCluster
      version: v1beta2
```

関連情報

- Operator Lifecycle Manager の依存関係の解決

2.2.1.4. opm CLI について

opm CLI ツールは、Operator Bundle Format で使用するために Operator Framework によって提供されます。このツールを使用して、ソフトウェアリポジトリーに相当する Operator バンドルのリストから Operator のカタログを作成し、維持することができます。結果として、コンテナーイメージをコンテナーレジストリーに保存し、その後にクラスターにインストールできます。

カタログには、コンテナーイメージの実行時に提供される組み込まれた API を使用してクエリーでき

る、Operator マニフェストコンテンツへのポインターのデータベースが含まれます。OpenShift Dedicated では、Operator Lifecycle Manager (OLM) は、**CatalogSource** オブジェクトが定義したカタログソース内のイメージ参照できます。これにより、クラスター上にインストールされた Operator への頻度の高い更新を可能にするためにイメージを一定の間隔でポーリングできます。

- **opm** CLI のインストール手順は、CLI ツール を参照してください。

2.2.2. ファイルベースのカタログ

ファイルベースのカタログは、Operator Lifecycle Manager(OLM) のカタログ形式の最新の反復になります。この形式は、プレーンテキストベース (JSON または YAML) であり、以前の SQLite データベース形式の宣言的な設定の進化であり、完全な下位互換性があります。この形式の目標は、Operator のカタログ編集、設定可能性、および拡張性を有効にすることです。

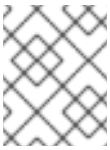
編集

ファイルベースのカタログを使用すると、カタログの内容を操作するユーザーは、形式を直接変更し、変更が有効であることを確認できます。この形式はプレーンテキストの JSON または YAML であるため、カタログメンテナーは、一般的に知られている、サポート対象の JSON または YAML ツール (例: **jq** CLI) を使用して、手動でカタログメタデータを簡単に操作できます。
この編集機能により、以下の機能とユーザー定義の拡張が有効になります。

- 既存のバンドルの新規チャネルへのプロモート
- パッケージのデフォルトチャネルの変更
- アップグレードエッジを追加、更新、および削除するためのカスタムアルゴリズム

コンポーザービリティー

ファイルベースのカタログは、任意のディレクトリー階層に保管され、カタログの作成が可能になります。たとえば、2 つのファイルベースのカタログディレクトリー (**catalogA** および **catalogB**) について見てみましょう。カタログメンテナーは、新規のディレクトリー **catalogC** を作成して **catalogA** と **catalogB** をそのディレクトリーにコピーし、新しく結合カタログを作成できます。
このコンポーザービリティーにより、カタログの分散化が可能になります。この形式により、Operator の作成者は Operator 固有のカタログを維持でき、メンテナーは個別の Operator カタログで設定されるカタログを簡単にビルドできます。ファイルベースのカタログは、他の複数のカタログを組み合わせたり、1 つのカタログのサブセットを抽出したり、またはこれらの両方を組み合わせたりすることで作成できます。



注記

パッケージ内でパッケージおよびバンドルを重複できません。**opm validate** コマンドは、重複が見つかった場合はエラーを返します。

Operator の作成者は Operator、その依存関係およびそのアップグレードの互換性について最も理解しているので、Operator 固有のカタログを独自のカタログに維持し、そのコンテンツを直接制御できます。ファイルベースのカタログの場合に、Operator の作成者はカタログでパッケージをビルドして維持するタスクを所有します。ただし、複合カタログメンテナーは、カタログ内のパッケージのキュレートおよびユーザーにカタログを公開するタスクのみを所有します。

拡張性

ファイルベースのカタログ仕様は、カタログの低レベル表現です。これは低レベルの形式で直接保守できますが、カタログメンテナーは、このレベルの上に任意の拡張をビルドして、独自のカスタムツールを使用して任意数の変更を加えることができます。

たとえば、ツールは **(mode=semver)** などの高レベルの API を、アップグレードエッジ用に低レベルのファイルベースのカタログ形式に変換できます。または、カタログメンテナーは、特定の条件を満たすバンドルに新規プロパティーを追加して、すべてのバンドルメタデータをカスタマイズする必要がある場合があります。

このような拡張性を使用すると、今後の OpenShift Dedicated リリース向けに、追加の正式なツールを低レベル API 上で開発できます。主な利点としては、カタログメンテナーもこの機能を利用できる点が挙げられます。

重要

OpenShift Dedicated 4.11 以降、デフォルトの Red Hat 提供の Operator カタログはファイルベースのカタログ形式でリリースされます。OpenShift Dedicated 4.6 ~ 4.10 用のデフォルトの Red Hat 提供 Operator カタログは、非推奨の SQLite データベース形式でリリースされました。

opm サブコマンド、フラグ、および SQLite データベース形式に関連する機能も非推奨となり、今後のリリースで削除されます。機能は引き続きサポートされており、非推奨の SQLite データベース形式を使用するカタログに使用する必要があります。

opm index prune などの SQLite データベース形式を使用する **opm** サブコマンドおよびフラグの多くは、ファイルベースのカタログ形式では機能しません。ファイルベースのカタログを使用する方法の詳細は、カスタムカタログの管理 を参照してください。

2.2.2.1. ディレクトリー構造

ファイルベースのカタログは、ディレクトリーベースのファイルシステムから保存してロードできます。**opm** CLI は、root ディレクトリーを元に、サブディレクトリーに再帰してカタログを読み込みます。CLI は、検出されるすべてのファイルの読み込みを試行し、エラーが発生した場合には失敗します。

.gitignore ファイルとパターンと優先順位が同じ **.indexignore** ファイルを使用して、カタログ以外のファイルを無視できます。

例: .indexignore ファイル

```
# Ignore everything except non-object .json and .yaml files
**/*
!*.json
!*.yaml
**/objects/*.json
**/objects/*.yaml
```

カタログメンテナーは、必要なレイアウトを柔軟に選択できますが、各パッケージのファイルベースのカタログ Blob は別々のサブディレクトリーに保管することを推奨します。個々のファイルは JSON または YAML のいずれかをしようしてください。カタログ内のすべてのファイルが同じ形式を使用する必要はありません。

推奨される基本構造

```
catalog
├── packageA
│   └── index.yaml
├── packageB
```

```
│   ├── .indexignore
│   ├── index.yaml
│   └── objects
│       └── packageB.v0.1.0.clusterserviceversion.yaml
└── packageC
    └── index.json
    └── deprecations.yaml
```

この推奨の構造には、ディレクトリー階層内の各サブディレクトリーは自己完結型のカタログであるという特性があるので、カタログの作成、検出、およびナビゲーションなどのファイルシステムの操作が簡素化されます。このカタログは、親カタログのルートディレクトリーにコピーして親カタログに追加することもできます。

2.2.2.2. スキーマ

ファイルベースのカタログは、任意のスキーマで拡張できる CUE 言語仕様 に基づく形式を使用します。以下の **_Meta** CUE スキーマは、すべてのファイルベースのカタログ Blob が順守する必要のある形式を定義します。

_Meta スキーマ

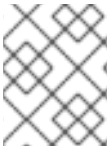
```
_Meta: {
  // schema is required and must be a non-empty string
  schema: string & !=""

  // package is optional, but if it's defined, it must be a non-empty string
  package?: string & !=""

  // properties is optional, but if it's defined, it must be a list of 0 or more properties
  properties?: [... #Property]
}

#Property: {
  // type is required
  type: string & !=""

  // value is required, and it must not be null
  value: !=null
}
```

注記

この仕様にリストされている CUE スキーマは網羅されていると見なされます。**opm validate** コマンドには、CUE で簡潔に記述するのが困難または不可能な追加の検証が含まれます。

Operator Lifecycle Manager(OLM) カタログは、現時点で OLM の既存のパッケージおよびバンドルの概念に対応する 3 つのスキーマ (**olm.package**、**olm.channel** および **olm.bundle**) を使用します。

カタログの各 Operator パッケージには、**olm.package** Blob が 1 つ (少なくとも **olm.channel** Blob 1 つ、および 1 つ以上の **olm.bundle** Blob) が必要です。



注記

olm.* スキーマは OLM 定義スキーマ用に予約されています。カスタムスキーマには、所有しているドメインなど、一意の接頭辞を使用する必要があります。

2.2.2.2.1. olm.package スキーマ

olm.package スキーマは Operator のパッケージレベルのメタデータを定義します。これには、名前、説明、デフォルトのチャネル、およびアイコンが含まれます。

例2.1 **olm.package** スキーマ

```
#Package: {
  schema: "olm.package"

  // Package name
  name: string & !=""

  // A description of the package
  description?: string

  // The package's default channel
  defaultChannel: string & !=""

  // An optional icon
  icon?: {
    base64data: string
    mediatype:  string
  }
}
```

2.2.2.2.2. olm.channel スキーマ

olm.channel スキーマは、パッケージ内のチャネル、チャネルのメンバーであるバンドルエントリー、およびそれらのバンドルのアップグレードエッジを定義します。

バンドルエントリーが複数の **olm.channel** Blob 内のエッジを表す場合、バンドルエントリーはチャネルごとに 1 つだけ指定できます。

エントリーの **replaces** 値が、このカタログにも別のカタログにも存在しない別のバンドル名を参照していても、有効とされます。ただし、他のすべてのチャネルの普遍条件に該当する必要があります (チャネルに複数のヘッドがない場合など)。

例2.2 **olm.channel** スキーマ

```
#Channel: {
  schema: "olm.channel"
  package: string & !=""
  name: string & !=""
  entries: [...#ChannelEntry]
}

#ChannelEntry: {
```

```
  // name is required. It is the name of an `olm.bundle` that
  // is present in the channel.
  name: string & !=""

  // replaces is optional. It is the name of bundle that is replaced
  // by this entry. It does not have to be present in the entry list.
  replaces?: string & !=""

  // skips is optional. It is a list of bundle names that are skipped by
  // this entry. The skipped bundles do not have to be present in the
  // entry list.
  skips?: [...string & !=""]

  // skipRange is optional. It is the semver range of bundle versions
  // that are skipped by this entry.
  skipRange?: string & !=""
}
```

警告

skipRange フィールドを使用すると、スキップされた Operator バージョンが更新グラフからプルーニングされ、ユーザーが **Subscription** オブジェクトの **spec.startingCSV** プロパティーを使用してそのバージョンをインストールできなくなります。

SkipRange フィールドと **replaces** フィールドの両方を使用すると、以前にインストールしたバージョンをユーザーが将来インストールできるように維持しながら、Operator を段階的に更新できます。**replaces** フィールドが当該 Operator バージョンの直前のバージョンを参照していることを確認してください。

2.2.2.2.3. olm.bundle スキーマ

例2.3 **olm.bundle** スキーマ

```
#Bundle: {
  schema: "olm.bundle"
  package: string & !=""
  name: string & !=""
  image: string & !=""
  properties: [...#Property]
  relatedImages?: [...#RelatedImage]
}

#Property: {
  // type is required
  type: string & !=""

  // value is required, and it must not be null
  value: !=null
```

```
}

#RelatedImage: {
  // image is the image reference
  image: string & !=""

  // name is an optional descriptive name for an image that
  // helps identify its purpose in the context of the bundle
  name?: string & !=""
}
```

2.2.2.2.4. olm.deprecations スキーマ

オプションの **olm.deprecations** スキーマは、カタログ内のパッケージ、バンドル、チャネルの非推奨情報を定義します。Operator の作成者は、このスキーマを使用して、サポートステータスや推奨アップグレードパスなど、Operator に関する関連メッセージを、カタログから Operator を実行しているユーザーに提供できます。

olm.deprecations スキーマエントリーには、非推奨の範囲を示す次の **reference** タイプが 1 つ以上含まれています。Operator がインストールされると、指定されたメッセージが、関連する **Subscription** オブジェクトのステータス状況として表示されます。

表2.1 非推奨の **reference** タイプ

タイプ	範囲	ステータス状況
olm.package	パッケージ全体を表します。	**PackageDeprecated**
olm.channel	1 つのチャンネルを表します。	**ChannelDeprecated**
olm.bundle	1 つのバンドルバージョンを表します。	**BundleDeprecated**

次の例で詳しく説明するように、各 **reference** タイプには独自の要件があります。

例2.4 各 **reference** タイプを使用した **olm.deprecations** スキーマの例

```
schema: olm.deprecations
package: my-operator 1
entries:
  - reference:
      schema: olm.package 2
    message: | 3
    The 'my-operator' package is end of life. Please use the
    'my-operator-new' package for support.
  - reference:
      schema: olm.channel
      name: alpha 4
    message: |
    The 'alpha' channel is no longer supported. Please switch to the
    'stable' channel.
  - reference:
```

```
  schema: olm.bundle
  name: my-operator.v1.68.0 5
message: |
my-operator.v1.68.0 is deprecated. Uninstall my-operator.v1.68.0 and
install my-operator.v1.72.0 for support.
```

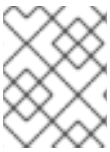
1 各非推奨スキーマには **package** 値が必要であり、そのパッケージ参照はカタログ全体で一意である必要があります。関連する **name** フィールドを含めることはできません。

2 **olm.package** スキーマに **name** フィールドを含めることはできません。このフィールドは、スキーマ内で前に定義した **package** フィールドによって決定されるためです。

3 すべての **message** フィールドは、**reference** タイプを問わず、長さが 0 以外である必要があり、不透明なテキスト Blob として表す必要があります。

4 **olm.channel** スキーマの **name** フィールドは必須です。

5 **olm.bundle** スキーマの **name** フィールドは必須です。



注記

非推奨機能では、パッケージ、チャネル、バンドルなど、重複する非推奨は考慮されません。

Operator の作成者は 、**olm.deprecations** スキーマエントリーを **deprecations.yaml** ファイルとしてパッケージの **index.yaml** ファイルと同じディレクトリーに保存できます。

非推奨を含むカタログのディレクトリー構造の例

```
my-catalog
└── my-operator
    ├── index.yaml
    └── deprecations.yaml
```

関連情報

- ファイルベースのカタログイメージの更新またはフィルタリング

2.2.2.3. プロパティー

プロパティーは、ファイルベースのカタログスキーマに追加できる任意のメタデータです。**type** フィールドは、**value** フィールドのセマンティックおよび構文上の意味を効果的に指定する文字列です。値には任意の JSON または YAML を使用できます。

OLM は、予約済みの **olm.*** 接頭辞をもう一度使用して、いくつかのプロパティータイプを定義します。

2.2.2.3.1. olm.package プロパティー

olm.package プロパティーは、パッケージ名とバージョンを定義します。これはバンドルの必須プロパティーであり、これらのプロパティーが 1 つ必要です。**packageName** フィールドはバンドルのファーストクラス **package** フィールドと同じでなければならず、**version** フィールドは有効なセマン

ティクスバージョンである必要があります。

例2.5 **olm.package** プロパティー

```
#PropertyPackage: {
  type: "olm.package"
  value: {
    packageName: string & !=""
    version: string & !=""
  }
}
```

2.2.2.3.2. olm.gvk プロパティー

olm.gvk プロパティーは、このバンドルで提供される Kubernetes API の group/version/kind(GVK) を定義します。このプロパティーは、OLM が使用して、必須の API と同じ GVK をリストする他のバンドルの依存関係として、このプロパティーでバンドルを解決します。GVK は Kubernetes GVK の検証に準拠する必要があります。

例2.6 **olm.gvk** プロパティー

```
#PropertyGVK: {
  type: "olm.gvk"
  value: {
    group: string & !=""
    version: string & !=""
    kind: string & !=""
  }
}
```

2.2.2.3.3. olm.package.required

olm.package.required プロパティーは、このバンドルが必要な別のパッケージのパッケージ名とバージョン範囲を定義します。バンドルにリストされている必要なパッケージプロパティーごとに、OLM は、リストされているパッケージのクラスターに必要なバージョン範囲で Operator がインストールされていることを確認します。**versionRange** フィールドは有効なセマンティクスバージョン (semver) の範囲である必要があります。

例2.7 **olm.package.required** プロパティー

```
#PropertyPackageRequired: {
  type: "olm.package.required"
  value: {
    packageName: string & !=""
    versionRange: string & !=""
  }
}
```

2.2.2.3.4. olm.gvk.required

olm.gvk.required プロパティーは、このバンドルが必要とする Kubernetes API の group/version/kind(GVK) を定義します。バンドルにリストされている必要な GVK プロパティーごとに、OLM は、提供する Operator がクラスターにインストールされていることを確認します。GVK は Kubernetes GVK の検証に準拠する必要があります。

例2.8 **olm.gvk.required** プロパティー

```
#PropertyGVKRequired: {
  type: "olm.gvk.required"
  value: {
    group: string & !=""
    version: string & !=""
    kind: string & !=""
  }
}
```

2.2.2.4. カタログの例

ファイルベースのカタログを使用すると、カタログメンテナーは Operator のキュレーションおよび互換性に集中できます。Operator の作成者は Operator 用に Operator 固有のカタログをすでに生成しているので、カタログメンテナーは、各 Operator カタログをカタログのルートディレクトリーのサブディレクトリーにレンダリングしてビルドできます。

ファイルベースのカタログをビルドする方法は多数あります。以下の手順は、単純なアプローチの概要を示しています。

1. カタログの設定ファイルを 1 つ維持し、カタログ内に Operator ごとにイメージの参照を含めます。

   カタログ設定ファイルのサンプル

   ```
   name: community-operators
   repo: quay.io/community-operators/catalog
   tag: latest
   references:
   - name: etcd-operator
     image: quay.io/etcd-
   operator/index@sha256:5891b5b522d5df086d0ff0b110fbd9d21bb4fc7163af34d08286a2e846f
   6be03
   - name: prometheus-operator
     image: quay.io/prometheus-
   operator/index@sha256:e258d248fda94c63753607f7c4494ee0fcbe92f1a76bfdac795c9d84101
   eb317
   ```

2. 設定ファイルを解析し、その参照から新規カタログを作成するスクリプトを実行します。

   スクリプトの例

   ```
   name=$(yq eval '.name' catalog.yaml)
   mkdir "$name"
   yq eval '.name + "/" + .references[].name' catalog.yaml | xargs mkdir
   for I in $(yq e '.name as $catalog | .references[] | .image + "|" + $catalog + "/" + .name +
   "/index.yaml"' catalog.yaml); do
   ```

```
  image=$(echo $l | cut -d'|' -f1)
  file=$(echo $l | cut -d'|' -f2)
  opm render "$image" > "$file"
done
opm generate dockerfile "$name"
indexImage=$(yq eval '.repo + ":" + .tag' catalog.yaml)
docker build -t "$indexImage" -f "$name.Dockerfile" .
docker push "$indexImage"
```

2.2.2.5. ガイドライン

ファイルベースのカタログを維持する場合には、以下のガイドラインを考慮してください。

2.2.2.5.1. イミュータブルなバンドル

Operator Lifecycle Manager(OLM) に関する一般的なアドバイスとして、バンドルイメージとそのメタデータをイミュータブルとして処理する必要がある点があります。

破損したバンドルがカタログにプッシュされている場合には、少なくとも 1 人のユーザーがそのバンドルにアップグレードしたと想定する必要があります。この仮定に基づいて、破損したバンドルがインストールされたユーザーがアップグレードを受信できるように、破損したバンドルから、アップグレードエッジが含まれる別のバンドルをリリースする必要があります。OLM は、カタログでバンドルの内容が更新された場合に、インストールされたバンドルは再インストールされません。

ただし、カタログメタデータの変更が推奨される場合があります。

- チャネルプロモーション: バンドルをすでにリリースし、後で別のチャネルに追加することにした場合は、バンドルのエントリーを別の **olm.channel** Blob に追加できます。
- 新規アップグレードエッジ:**1.2.z** バンドルバージョンを新たにリリースしたが (例:**1.2.4**)、**1.3.0** がすでにリリースされている場合は、**1.2.4** をスキップするように **1.3.0** のカタログメタデータを更新できます。

2.2.2.5.2. ソース制御

カタログメタデータはソースコントロールに保存され、信頼できる情報源として処理される必要があります。以下の手順で、カタログイメージを更新する必要があります。

1. ソース制御されたカタログディレクトリーを新規コミットを使用して更新します。
2. カタログイメージをビルドし、プッシュします。ユーザーがカタログが利用可能になり次第更新を受信できるように、一貫性のあるタグ付け (**:latest** or **:<target_cluster_version>**) を使用します。

2.2.2.6. CLI の使用

opm CLI を使用してファイルベースのカタログを作成する方法は、カスタムカタログの管理 を参照してください。

ファイルベースのカタログの管理に関連する **opm** CLI コマンドについての参考情報は、CLI ツール を参照してください。

2.2.2.7. 自動化

Operator の作成者およびカタログメンテナーは、CI/CD ワークフローを使用してカタログのメンテナンスを自動化することが推奨されます。カタログメンテナーは、GitOps 自動化をビルドして以下のタスクを実行し、これをさらに向上させることができます。

- パッケージのイメージ参照の更新など、プル要求 (PR) の作成者が要求された変更を実行できることを確認します。
- カタログの更新で **opm validate** コマンドが指定されていることを確認します。
- 更新されたバンドルまたはカタログイメージの参照が存在し、カタログイメージがクラスターで正常に実行され、そのパッケージの Operator が正常にインストールされることを確認します。
- 以前のチェックに合格した bmcs を自動的にマージします。
- カタログイメージを自動的にもう一度ビルドして公開します。

2.3. OPERATOR FRAMEWORK の一般的な用語の用語集

このトピックでは、パッケージ形式についての Operator Lifecycle Manager (OLM) および Operator SDK を含む、Operator Framework に関連する一般的な用語の用語集を提供します。

2.3.1. Common Operator Framework の一般的な用語

2.3.1.1. バンドル

Bundle Format では、バンドル は Operator CSV、マニフェスト、およびメタデータのコレクションです。さらに、それらはクラスターにインストールできる一意のバージョンの Operator を形成します。

2.3.1.2. バンドルイメージ

Bundle Format では、バンドルイメージ は Operator マニフェストからビルドされ、1 つのバンドルが含まれるコンテナーイメージです。バンドルイメージは、Quay.io または DockerHub などの Open Container Initiative (OCI) 仕様コンテナーレジストリーによって保存され、配布されます。

2.3.1.3. カタログソース

カタログソース は、OLM が Operator およびそれらの依存関係を検出し、インストールするためにクエリーできるメタデータのストアを表します。

2.3.1.4. Channel

チャネル は Operator の更新ストリームを定義し、サブスクライバーの更新をロールアウトするために使用されます。ヘッドはそのチャネルの最新バージョンを参照します。たとえば **stable** チャネルには、Operator のすべての安定したバージョンが最も古いものから最新のものへと編成されます。

Operator には複数のチャネルを含めることができ、特定のチャネルへのサブスクリプションのバインドはそのチャネル内の更新のみを検索します。

2.3.1.5. チャネルヘッド

チャネルヘッド は、特定のチャネル内の最新の既知の更新を指します。

2.3.1.6. クラスターサービスバージョン

クラスターサービスバージョン **(CSV)** は、クラスターでの Operator の実行に使用される 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イスにロゴ、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に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。

2.3.1.7. 依存関係

Operator はクラスターに存在する別の Operator への 依存関係 を持つ場合があります。たとえば、Vault Operator にはそのデータ永続層について etcd Operator への依存関係があります。

OLM は、インストールフェーズで指定されたすべてのバージョンの Operator および CRD がクラスターにインストールされていることを確認して依存関係を解決します。この依存関係は、必要な CRD API を満たすカタログの Operator を検索し、インストールすることで解決され、パッケージまたはバンドルには関連しません。

2.3.1.8. インデックスイメージ

Bundle Format で、インデックスイメージ は、すべてのバージョンの CSV および CRD を含む Operator バンドルについての情報が含まれるデータベースのイメージ (データベーススナップショット) を指します。このインデックスは、クラスターで Operator の履歴をホストでき、**opm** CLI ツールを使用して Operator を追加または削除することで維持されます。

2.3.1.9. インストール計画

インストール計画 は、CSV を自動的にインストールするか、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です。

2.3.1.10. マルチテナントへの対応

OpenShift Dedicated の テナント は、デプロイされた一連のワークロードに対する共通のアクセス権と権限を共有するユーザーまたはユーザーのグループであり、通常は namespace またはプロジェクトで表されます。テナントを使用して、異なるグループまたはチーム間に一定レベルの分離を提供できます。

クラスターが複数のユーザーまたはグループによって共有されている場合、マルチテナント クラスターと見なされます。

2.3.1.11. Operator グループ

Operator グループ は、 **OperatorGroup** オブジェクトと同じ namespace にデプロイされたすべての Operator を、namespace のリストまたはクラスター全体でそれらの CR を監視できるように設定します。

2.3.1.12. Package

Bundle Format で、パッケージ は Operator のリリースされたすべての履歴をそれぞれのバージョンで囲むディレクトリーです。Operator のリリースされたバージョンは、CRD と共に CSV マニフェストに記述されます。

2.3.1.13. レジストリー

レジストリー は、Operator のバンドルイメージを保存するデータベースで、それぞれにすべてのチャネルの最新バージョンおよび過去のバージョンすべてが含まれます。

2.3.1.14. サブスクリプション

サブスクリプション は、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。

2.3.1.15. 更新グラフ

更新グラフ は、他のパッケージ化されたソフトウェアの更新グラフと同様に、CSV の複数のバージョンを 1 つにまとめます。Operator を順番にインストールすることも、特定のバージョンを省略することもできます。更新グラフは、新しいバージョンが追加されている状態でヘッドでのみ拡張することが予想されます。

2.4. OPERATOR LIFECYCLE MANAGER (OLM)

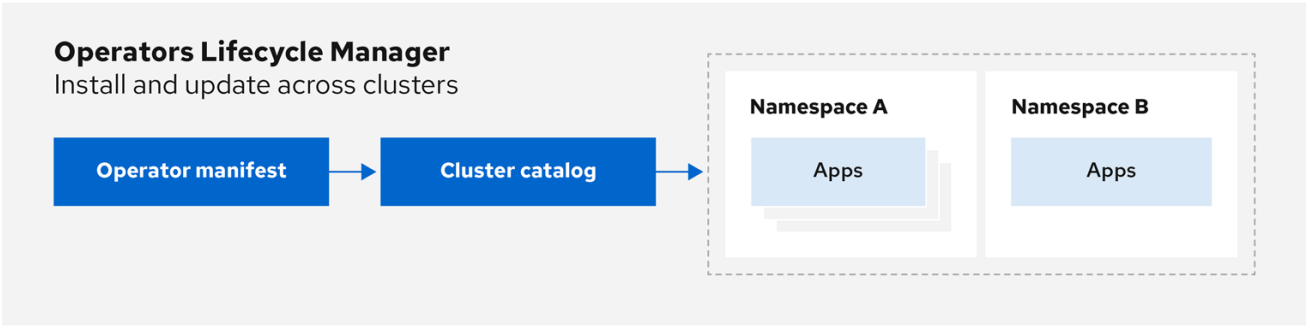
2.4.1. Operator Lifecycle Manager の概念およびリソース

このガイドでは、OpenShift Dedicated の Operator Lifecycle Manager (OLM) を支える概念の概要を説明します。

2.4.1.1. Operator Lifecycle Manager について

Operator Lifecycle Manager (OLM) を使用することにより、ユーザーは Kubernetes ネイティブアプリケーション (Operator) および OpenShift Dedicated クラスター全体で実行される関連サービスについてインストール、更新、およびそのライフサイクルの管理を実行できます。これは、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



OpenShift_43_1019

OLM は OpenShift Dedicated 4 でデフォルトで実行されます。これは、**dedicated-admin** ロールを持つ管理者が、クラスターで実行されている Operator のインストール、アップグレード、アクセス付与を行う際に役立ちます。OpenShift Dedicated Web コンソールは、**dedicated-admin** 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。

2.4.1.2. OLM リソース

以下のカスタムリソース定義 (CRD) は Operator Lifecycle Manager (OLM) によって定義され、管理されます。

表2.2 OLM およびカタログ Operator で管理される CRD

リソース	短縮名	説明
ClusterServiceVersion (CSV)	**csv**	アプリケーションメタデータ:例: 名前、バージョン、アイコン、必須リソース。
CatalogSource	**catsrc**	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。
サブスクリプション	**sub**	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。
InstallPlan	**ip**	CSV を自動的にインストールするか、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。
OperatorGroup	**og**	**OperatorGroup** オブジェクトと同じ namespace にデプロイされたすべての Operator を、namespace のリストまたはクラスター全体でカスタムリソース (CR) を監視できるように設定します。
OperatorConditions	-	OLM とそれが管理する Operator との間で通信チャネルを作成します。Operator は **Status.Conditions** 配列に書き込みを行い、複雑な状態を OLM と通信できます。

2.4.1.2.1. クラスターサービスバージョン

クラスターサービスバージョン (CSV) は、OpenShift Dedicated クラスター上で実行中の Operator の特定バージョンを表します。これは、クラスターでの Operator Lifecycle Manager (OLM) の Operator の実行に使用される Operator メタデータから作成される YAML マニフェストです。

OLM は Operator についてのこのメタデータを要求し、これがクラスターで安全に実行できるようにし、Operator の新規バージョンが公開される際に更新を適用する方法についての情報を提供します。これは従来のオペレーティングシステムのソフトウェアのパッケージに似ています。OLM のパッケージ手順を、**rpm**、**dep**、または **apk** バンドルを作成するステージとして捉えることができます。

CSV には、ユーザーインターフェイスに名前、バージョン、説明、ラベル、リポジトリーリンクおよびロゴ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に伴うメタデータが含まれます。

CSV は、Operator が管理したり、依存したりするカスタムリソース (CR)、RBAC ルール、クラスター要件、およびインストールストラテジーなどの Operator の実行に必要な技術情報の情報源でもあります。この情報は OLM に対して必要なリソースの作成方法と、Operator をデプロイメントとしてセットアップする方法を指示します。

2.4.1.2.2. カタログソース

カタログソース は、通常コンテナーレジストリーに保存されている インデックスイメージ を参照して

メタデータのストアを表します。Operator Lifecycle Manager(OLM) はカタログソースをクエリーし、Operator およびそれらの依存関係を検出してインストールします。OpenShift Dedicated Web コンソールの OperatorHub には、カタログソースによって提供される Operator も表示されます。

ヒント

クラスター管理者は、Web コンソールの **Administration → Cluster Settings → Configuration → OperatorHub** ページを使用して、クラスターで有効なログソースにより提供される Operator の詳細一覧を表示できます。

CatalogSource オブジェクトの **spec** は、Pod の構築方法、または Operator レジストリー gRPC API を提供するサービスとの通信方法を示します。

例2.9 **CatalogSource** オブジェクトの例

```
apiVersion: operators.coreos.com/v1alpha1
kind: CatalogSource
metadata:
  generation: 1
  name: example-catalog (1)
  namespace: openshift-marketplace (2)
  annotations:
    olm.catalogImageTemplate: (3)
      "quay.io/example-org/example-catalog:v{kube_major_version}.{kube_minor_version}.
{kube_patch_version}"
spec:
  displayName: Example Catalog (4)
  image: quay.io/example-org/example-catalog:v1 (5)
  priority: -400 (6)
  publisher: Example Org
  sourceType: grpc (7)
  grpcPodConfig:
    securityContextConfig: <security_mode> (8)
    nodeSelector: (9)
      custom_label: <label>
    priorityClassName: system-cluster-critical (10)
    tolerations: (11)
      - key: "key1"
        operator: "Equal"
        value: "value1"
        effect: "NoSchedule"
  updateStrategy:
    registryPoll: (12)
      interval: 30m0s
status:
  connectionState:
    address: example-catalog.openshift-marketplace.svc:50051
    lastConnect: 2021-08-26T18:14:31Z
    lastObservedState: READY (13)
  latestImageRegistryPoll: 2021-08-26T18:46:25Z (14)
  registryService: (15)
    createdAt: 2021-08-26T16:16:37Z
    port: 50051
```

```
  protocol: grpc
  serviceName: example-catalog
  serviceNamespace: openshift-marketplace
```

1. **CatalogSource** オブジェクトの名前。この値は、要求された namespace で作成される、関連の Pod 名の一部としても使用されます。

2. カタログを作成する namespace。カタログを全 namespace のクラスター全体で利用可能にするには、この値を **openshift-marketplace** に設定します。Red Hat が提供するデフォルトのカタログソースも **openshift-marketplace** namespace を使用します。それ以外の場合は、値を特定の namespace に設定し、Operator をその namespace でのみ利用可能にします。

3. 任意: クラスターのアップグレードにより、Operator のインストールがサポートされていない状態になったり、更新パスが継続されなかったりする可能性を回避するために、クラスターのアップグレードの一環として、Operator カタログのインデックスイメージのバージョンを自動的に変更するように有効化することができます。

   olm.catalogImageTemplate アノテーションをインデックスイメージ名に設定し、イメージタグのテンプレートを作成する際に、1 つ以上の Kubernetes クラスターバージョン変数を使用します。アノテーションは、実行時に **spec.image** フィールドを上書きします。詳細は、カスタムカタログソースのイメージテンプレートのセクションを参照してください。

4. Web コンソールおよび CLI でのカタログの表示名。

5. カタログのインデックスイメージ。オプションで、**olm.catalogImageTemplate** アノテーションを使用して実行時のプル仕様を設定する場合には、省略できます。

6. カタログソースの重み。OLM は重みを使用して依存関係の解決時に優先順位付けします。重みが大きい場合は、カタログが重みの小さいカタログよりも優先されることを示します。

7. ソースタイプには以下が含まれます。

   - **image** 参照のある **grpc**: OLM はイメージをポーリングし、Pod を実行します。これにより、準拠 API が提供されることが予想されます。
   - **address** フィールドのある **grpc**: OLM は所定アドレスでの gRPC API へのアクセスを試行します。これはほとんどの場合使用することができません。
   - **ConfigMap**: OLM は設定マップデータを解析し、gRPC API を提供できる Pod を実行します。

8. **legacy** または **restricted** の値を指定します。フィールドが設定されていない場合、デフォルト値は **legacy** です。今後の OpenShift Dedicated リリースでは、デフォルト値が **restricted** になる予定です。**restricted** 権限でカタログを実行できない場合は、このフィールドを手動で **legacy** に設定することを推奨します。

9. オプション: **grpc** タイプのカタログソースの場合は、**spec.image** でコンテンツを提供する Pod のデフォルトのノードセレクターをオーバーライドします (定義されている場合)。

10. オプション: **grpc** タイプのカタログソースの場合は、**spec.image** でコンテンツを提供する Pod のデフォルトの優先度クラス名をオーバーライドします (定義されている場合)。Kubernetes は、デフォルトで優先度クラス **system-cluster-critical** および **system-node-critical** を提供します。フィールドを空 (**""**) に設定すると、Pod にデフォルトの優先度が割り当てられます。他の優先度クラスは、手動で定義できます。

11. オプション: **grpc** タイプのカタログソースの場合は、**spec.image** でコンテンツを提供する Pod のデフォルトの Toleration をオーバーライドします (定義されている場合)。

12 最新の状態を維持するために、特定の間隔で新しいバージョンの有無を自動的にチェックします。

13 カタログ接続が最後に監視された状態。以下に例を示します。

- **READY**: 接続が正常に確立されました。
- **CONNECTING**: 接続が確立中です。
- **TRANSIENT_FAILURE**: タイムアウトなど、接続の確立時一時的な問題が発生しました。状態は最終的に **CONNECTING** に戻り、再試行されます。

詳細は、gRPC ドキュメントの 接続の状態 を参照してください。

14 カタログイメージを保存するコンテナーレジストリーがポーリングされ、イメージが最新の状態であることを確認します。

15 カタログの Operator レジストリーサービスのステータス情報。

サブスクリプションの **CatalogSource** オブジェクトの **name** を参照すると、要求された Operator を検索する場所を、OLM に指示します。

例2.10 カタログソースを参照する **Subscription** オブジェクトの例

```
apiVersion: operators.coreos.com/v1alpha1
kind: Subscription
metadata:
  name: example-operator
  namespace: example-namespace
spec:
  channel: stable
  name: example-operator
  source: example-catalog
  sourceNamespace: openshift-marketplace
```

関連情報

- OperatorHub について
- Red Hat が提供する Operator カタログ
- クラスターへのカタログソースの追加
- カタログの優先順位
- CLI を使用した Operator カタログソースのステータス表示
- カタログソース Pod のスケジューリング

2.4.1.2.2.1. カスタムカタログソースのイメージテンプレート

基礎となるクラスターとの Operator との互換性は、さまざまな方法でカタログソースにより表現でき

ます。1つの方法は、特定のプラットフォームリリース (OpenShift Dedicated 4 など) 用に特別に作成されたインデックスイメージのイメージタグを特定することです。この方法は、Red Hat 提供のデフォルトのカタログソースに使用されています。

クラスターのアップグレード時に、Red Hat が提供するデフォルトのカタログソースのインデックスイメージのタグは、Operator Lifecycle Manager (OLM) が最新版のカタログをプルするように、Cluster Version Operator (CVO) により自動更新されます。たとえば、OpenShift Dedicated 4.14 から 4.15 へのアップグレード時に、**redhat-operators** カタログの **CatalogSource** オブジェクトの **spec.image** フィールドは次のように更新されます。

```
registry.redhat.io/redhat/redhat-operator-index:v4.14
```

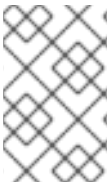
更新後は次のようになります。

```
registry.redhat.io/redhat/redhat-operator-index:v4.15
```

ただし、CVO ではカスタムカタログのイメージタグは自動更新されません。クラスターのアップグレード後、ユーザーが互換性があり、サポート対象の Operator のインストールを確実に行えるようにするには、カスタムカタログも更新して、更新されたインデックスイメージを参照する必要があります。

OpenShift Dedicated 4.9 以降、クラスター管理者は、カスタムカタログの **CatalogSource** オブジェクトの **olm.catalogImageTemplate** アノテーションを、テンプレートを含むイメージ参照に追加できます。以下の Kubernetes バージョン変数は、テンプレートで使用できるようにサポートされています。

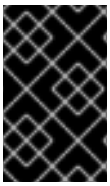
- **kube_major_version**
- **kube_minor_version**
- **kube_patch_version**



注記

OpenShift Dedicated クラスターのバージョンではなく、Kubernetes クラスターのバージョンを指定する必要があります。OpenShift Dedicated クラスターのバージョンは、現在テンプレートに使用できないためです。

更新された Kubernetes バージョンを指定するタグでインデックスイメージを作成してプッシュしている場合に、このアノテーションを設定すると、カスタムカタログのインデックスイメージのバージョンがクラスターのアップグレード後に自動的に変更されます。アノテーションの値は、**CatalogSource** オブジェクトの **spec.image** フィールドでイメージ参照を設定したり、更新したりするために使用されます。こうすることで、サポートなしの状態や、継続する更新パスなしの状態で Operator がインストールされないようにします。



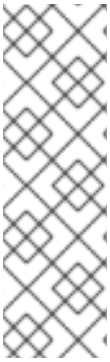
重要

格納されているレジストリーがどれであっても、クラスターのアップグレード時に、クラスターが、更新されたタグを含むインデックスイメージにアクセスできるようにする必要があります。

例2.11 イメージテンプレートを含むカタログソースの例

```
apiVersion: operators.coreos.com/v1alpha1
kind: CatalogSource
```

```
metadata:
  generation: 1
  name: example-catalog
  namespace: openshift-marketplace
  annotations:
    olm.catalogImageTemplate:
      "quay.io/example-org/example-catalog:v{kube_major_version}.{kube_minor_version}"
spec:
  displayName: Example Catalog
  image: quay.io/example-org/example-catalog:v1.28
  priority: -400
  publisher: Example Org
```



注記

spec.image フィールドおよび **olm.catalogImageTemplate** アノテーションの両方が設定されている場合には、**spec.image** フィールドはアノテーションから解決された値で上書きされます。アノテーションが使用可能なプル仕様に対して解決されない場合は、カタログソースは **spec.image** 値にフォールバックします。

spec.image フィールドが設定されていない場合に、アノテーションが使用可能なプル仕様に対して解決されない場合は、OLM はカタログソースの調整を停止し、人間が判読できるエラー条件に設定します。

Kubernetes 1.28 を使用する OpenShift Dedicated 4 クラスターの場合、前の例の **olm.catalogImageTemplate** アノテーションは次のイメージ参照に解決されます。

```
quay.io/example-org/example-catalog:v1.28
```

OpenShift Dedicated の今後のリリースに備えて、新しい OpenShift Dedicated バージョンで使用される新しい Kubernetes バージョンを対象とした、カスタムカタログの更新済みインデックスイメージを作成できます。アップグレード前に **olm.catalogImageTemplate** アノテーションを設定してから、クラスターを新しい OpenShift Dedicated バージョンにアップグレードすると、カタログのインデックスイメージも自動的に更新されます。

2.4.1.2.2.2. カタログの正常性要件

クラスター上の Operator カタログは、インストール解決の観点から相互に置き換え可能です。**Subscription** オブジェクトは特定のカタログを参照する場合がありますが、依存関係はクラスターのすべてのカタログを使用して解決されます。

たとえば、カタログ A が正常でない場合、カタログ A を参照するサブスクリプションはカタログ B の依存関係を解決する可能性があります。通常、B のカタログ優先度は A よりも低いため、クラスター管理者はこれおを想定していない可能性があります。

その結果、OLM では、特定のグローバル namespace (デフォルトの **openshift-marketplace** namespace やカスタムグローバル namespace など) を持つすべてのカタログが正常であることが必要になります。カタログが正常でない場合、その共有グローバル namespace 内のすべての Operator のインストールまたは更新操作は、**CatalogSourcesUnhealthy** 状態で失敗します。正常でない状態でこれらの操作が許可されている場合、OLM はクラスター管理者が想定しない解決やインストールを決定する可能性があります。

クラスター管理者が、カタログが正常でないことを確認し、無効とみなして Operator インストールを

クラスター管理者が、カタログが正常でないことを確認し、無効とみなして Operator インストールを再開する必要がある場合は、「カスタムカタログの削除」または「デフォルトの OperatorHub カタログソースの無効化」セクションで、正常でないカタログの削除について確認してください。

2.4.1.2.3. Subscription

サブスクリプション は、**Subscription** オブジェクトによって定義され、Operator をインストールする意図を表します。これは、Operator をカタログソースに関連付けるカスタムリソースです。

サブスクリプションは、サブスクライブする Operator パッケージのチャネルや、更新を自動または手動で実行するかどうかを記述します。サブスクリプションが自動に設定された場合、Operator Lifecycle Manager (OLM) が Operator を管理し、アップグレードして、最新バージョンがクラスター内で常に実行されるようにします。

Subscription オブジェクトの例

```
apiVersion: operators.coreos.com/v1alpha1
kind: Subscription
metadata:
  name: example-operator
  namespace: example-namespace
spec:
  channel: stable
  name: example-operator
  source: example-catalog
  sourceNamespace: openshift-marketplace
```

この **Subscription** オブジェクトは、Operator の名前および namespace および Operator データのあるカタログを定義します。**alpha**、**beta**、または **stable** などのチャネルは、カタログソースからインストールする必要のある Operator ストリームを判別するのに役立ちます。

サブスクリプションのチャネルの名前は Operator 間で異なる可能性がありますが、命名スキームは指定された Operator 内の一般的な規則に従う必要があります。たとえば、チャネル名は Operator によって提供されるアプリケーションのマイナーリリース更新ストリーム (**1.2**、**1.3**) またはリリース頻度 (**stable**、**fast**) に基づく可能性があります。

関連するサブスクリプションのステータスを調べると、Operator の新しいバージョンが利用可能になったことを確認できます。これは、OpenShift Dedicated Web コンソールからも簡単に確認できます。**currentCSV** フィールドに関連付けられる値は OLM に認識される最新のバージョンであり、**installedCSV** はクラスターにインストールされるバージョンです。

関連情報

- CLI を使用した Operator サブスクリプションステータスの表示

2.4.1.2.4. インストール計画

InstallPlan オブジェクトによって定義される インストール計画 は、Operator Lifecycle Manager(OLM) が特定バージョンの Operator をインストールまたはアップグレードするために作成するリソースのセットを記述します。バージョンはクラスターサービスバージョン (CSV) で定義されます。

Operator、クラスター管理者、または Operator インストールパーミッションが付与されているユーザーをインストールするには、まず **Subscription** オブジェクトを作成する必要があります。サブスクリプションでは、カタログソースから利用可能なバージョンの Operator のストリームにサブスクライ

ブする意図を表します。次に、サブスクリプションは **InstallPlan** オブジェクトを作成し、Operator のリソースのインストールを容易にします。

その後、インストール計画は、以下の承認ストラテジーのいずれかをもとに承認される必要があります。

- サブスクリプションの **spec.installPlanApproval** フィールドが **Automatic** に設定されている場合には、インストール計画は自動的に承認されます。
- サブスクリプションの **spec.installPlanApproval** フィールドが **Manual** に設定されている場合には、インストール計画はクラスター管理者または適切なパーミッションが割り当てられたユーザーによって手動で承認する必要があります。

インストール計画が承認されると、OLM は指定されたリソースを作成し、サブスクリプションで指定された namespace に Operator をインストールします。

例2.12 **InstallPlan** オブジェクトの例

```
apiVersion: operators.coreos.com/v1alpha1
kind: InstallPlan
metadata:
  name: install-abcde
  namespace: operators
spec:
  approval: Automatic
  approved: true
  clusterServiceVersionNames:
    - my-operator.v1.0.1
  generation: 1
status:
  ...
  catalogSources: []
  conditions:
    - lastTransitionTime: '2021-01-01T20:17:27Z'
      lastUpdateTime: '2021-01-01T20:17:27Z'
      status: 'True'
      type: Installed
  phase: Complete
  plan:
    - resolving: my-operator.v1.0.1
      resource:
        group: operators.coreos.com
        kind: ClusterServiceVersion
        manifest: >-
        ...
        name: my-operator.v1.0.1
        sourceName: redhat-operators
        sourceNamespace: openshift-marketplace
        version: v1alpha1
      status: Created
    - resolving: my-operator.v1.0.1
      resource:
        group: apiextensions.k8s.io
        kind: CustomResourceDefinition
        manifest: >-
        ...
```

```
      name: webservers.web.servers.org
      sourceName: redhat-operators
      sourceNamespace: openshift-marketplace
      version: v1beta1
    status: Created
  - resolving: my-operator.v1.0.1
    resource:
      group: ''
      kind: ServiceAccount
      manifest: >-
      ...
      name: my-operator
      sourceName: redhat-operators
      sourceNamespace: openshift-marketplace
      version: v1
    status: Created
  - resolving: my-operator.v1.0.1
    resource:
      group: rbac.authorization.k8s.io
      kind: Role
      manifest: >-
      ...
      name: my-operator.v1.0.1-my-operator-6d7cbc6f57
      sourceName: redhat-operators
      sourceNamespace: openshift-marketplace
      version: v1
    status: Created
  - resolving: my-operator.v1.0.1
    resource:
      group: rbac.authorization.k8s.io
      kind: RoleBinding
      manifest: >-
      ...
      name: my-operator.v1.0.1-my-operator-6d7cbc6f57
      sourceName: redhat-operators
      sourceNamespace: openshift-marketplace
      version: v1
    status: Created
    ...
```

2.4.1.2.5. Operator グループ

Operator グループ は、 **OperatorGroup** リソースによって定義され、マルチテナント設定を OLM でインストールされた Operator に提供します。Operator グループは、そのメンバー Operator に必要な RBAC アクセスを生成するために使用するターゲット namespace を選択します。

ターゲット namespace のセットは、クラスターサービスバージョン (CSV) の **olm.targetNamespaces** アノテーションに保存されるコ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。

関連情報

- Operator グループ

2.4.1.2.6. Operator 条件

Operator のライフサイクル管理のロールの一部として、Operator Lifecycle Manager (OLM) は、Operator を定義する Kubernetes リソースの状態から Operator の状態を推測します。このアプローチでは、Operator が特定の状態にあることをある程度保証しますが、推測できない情報を Operator が OLM と通信して提供する必要がある場合も多々あります。続いて、OLM がこの情報を使用して、Operator のライフサイクルをより適切に管理することができます。

OLM は、Operator が OLM に条件について通信できる **OperatorCondition** というカスタムリソース定義 (CRD) を提供します。**OperatorCondition** リソースの **Spec.Conditions** 配列にある場合に、OLM による Operator の管理に影響するサポートされる条件のセットがあります。

注記

デフォルトでは、 **Spec.Conditions**配列は、ユーザーによって追加されるか、カスタム Operator ロジックの結果として追加されるまで、 **Operator Condition**オブジェクトに存在しません。

関連情報

- Operator 条件

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

ここでは、OpenShift Dedicated における Operator Lifecycle Manager (OLM) のコンポーネントアーキテクチャーの概要を説明します。

2.4.2.1. コンポーネントのロール

Operator Lifecycle Manager (OLM) は、OLM Operator および Catalog Operator の 2 つの Operator で設定されています。

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2.3 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)	**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、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。

リソース	短縮名	所有する Operator	説明
サブスクリプション	**sub**	カタログ	パッケージのチャネルを追跡して CSV を最新の状態に保つために使用されます。
OperatorGroup	**og**	OLM	**OperatorGroup** オブジェクトと同じ namespace にデプロイされたすべての Operator を、namespace のリストまたはクラスター全体でカスタムリソース (CR) を監視できるように設定します。

これらの Operator のそれぞれは以下のリソースの作成も行います。

表2.4 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
Deployments	OLM
ServiceAccounts	
(Cluster)Role	
(Cluster)RoleBinding	
CustomResourceDefinitions (CRDs)	カタログ
ClusterServiceVersions	

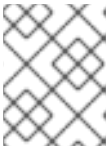
2.4.2.2. OLM Operator

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 またはカタログ Operator を使用してこれらのリソースを手動で作成することを選択できます。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 は以下のワークフローを使用します。

1. namespace でクラスターサービスバージョン (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。
2. 要件が満たされている場合、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ために Operator グループのアクティブなメンバーである必要があります。

2.4.2.3. カタログ Operator

カタログ Operator はクラスターサービスバージョン (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、カタログソースでチャネル内のパッケージへの更新の有無を確認し、必要な場合はそれらを利用可能な最新バージョンに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ために、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** オブジェクトを設定して **Subscription** オブジェクトを作成できます。更新が見つかると、ユーザーに代わって適切な **InstallPlan** オブジェクトの namespace への書き込みが行われます。

カタログ Operator は以下のワークフローを使用します。

1. クラスターの各カタログソースに接続します。
2. ユーザーによって作成された未解決のインストール計画の有無を確認し、これがあった場合は以下を実行します。
    a. 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。
    b. マネージドまたは必須の 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。
    c. 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。
3. 解決済みのインストール計画の有無を確認し、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)。
4. カタログソースおよびサブスクリプションの有無を確認し、それらに基づいてインストール計画を作成します。

2.4.2.4. 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)。

2.4.3. Operator Lifecycle Manager ワークフロー

ここでは、OpenShift Dedicated における Operator Lifecycle Manager (OLM) のワークフローの概要を説明します。

2.4.3.1. OLM での Operator のインストールおよびアップグレードのワークフロー

Operator Lifecycle Manager (OLM) エコシステムでは、以下のリソースを使用して Operator インストールおよびアップグレードを解決します。

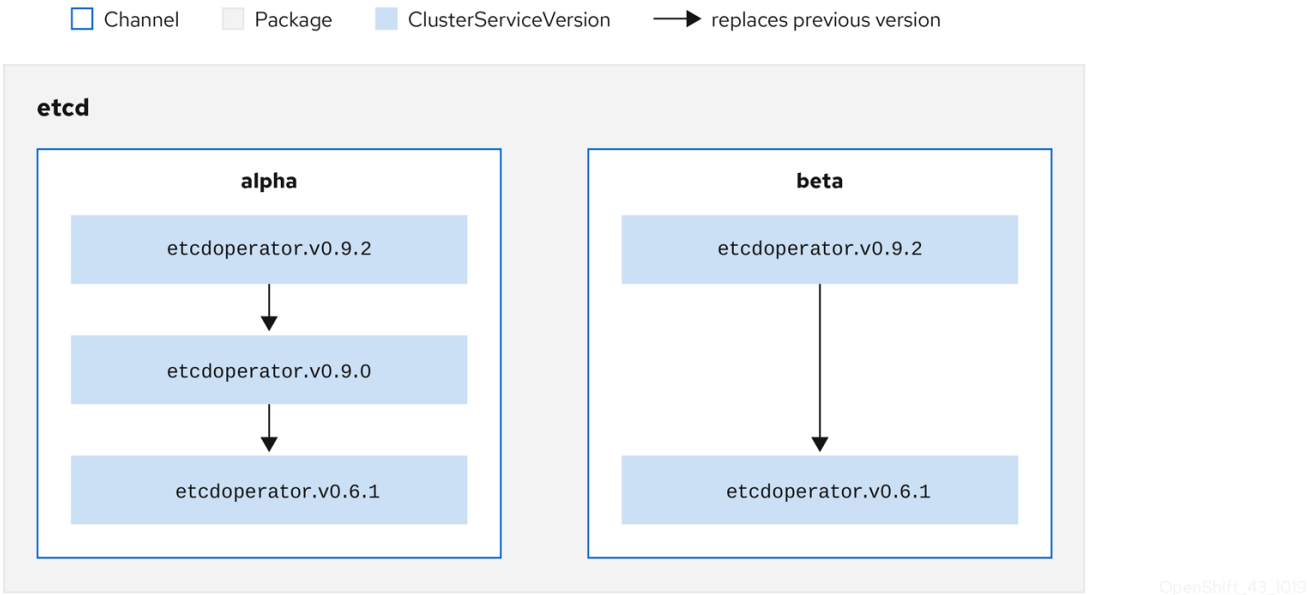
- **ClusterServiceVersion** (CSV)
- **CatalogSource**
- サブスクリプション

CSV で定義される Operator メタデータは、カタログソースというコレクションに保存できます。OLM はカタログソースを使用します。これは Operator Registry API を使用して利用可能な Operator やインストールされた Operator のアップグレードについてクエリーします。

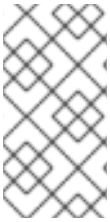
図2.3 カタログソースの概要

カタログソース内では、Operator は パッケージ と、チャネル と呼ばれる更新のストリームに編成されています。これは、OpenShift Dedicated や、Web ブラウザーなどの継続的なリリースサイクルを持つその他のソフトウェアでもよく見られる更新パターンです。

図2.4 カタログソースのパッケージおよびチャネル



ユーザーは サブスクリプション の特定のカタログソースの特定のパッケージおよびチャネルを指定できます (例: **etcd** パッケージおよびその **alpha** チャネル)。サブスクリプションが namespace にインストールされていないパッケージに対して作成されると、そのパッケージの最新 Operator がインストールされます。

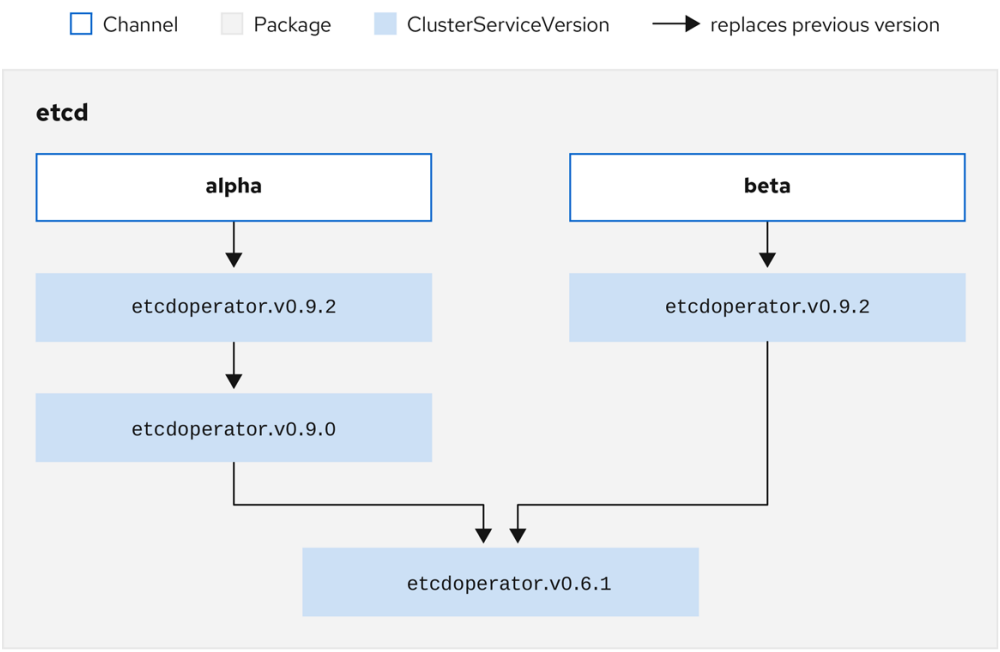


注記

OLM では、バージョンの比較が意図的に避けられます。そのため、所定の **catalog → channel → package** パスから利用可能な latest または newest Operator が必ずしも最も高いバージョン番号である必要はありません。これは Git リポジトリーの場合と同様に、チャネルの **Head** リファレンスとして見なされます。

各 CSV には、これが置き換える Operator を示唆する **replaces** パラメーターがあります。これにより、OLM でクエリー可能な CSV のグラフが作成され、更新がチャネル間で共有されます。チャネルは、更新グラフのエントリーポイントと見なすことができます。

図2.5 利用可能なチャネル更新についての OLM グラフ



パッケージのチャネルの例

```
packageName: example
channels:
- name: alpha
  currentCSV: example.v0.1.2
- name: beta
  currentCSV: example.v0.1.3
defaultChannel: alpha
```

カタログソース、パッケージ、チャネルおよび CSV がある状態で、OLM が更新のクエリーを実行できるようにするには、カタログが入力された CSV の置き換え (**replaces**) を実行する単一 CSV を明確にかつ確定的に返すことができる必要があります。

2.4.3.1.1. アップグレードパスの例

アップグレードシナリオのサンプルについて、CSV バージョン **0.1.1** に対応するインストールされた Operator について見てみましょう。OLM はカタログソースをクエリーし、新規 CSV バージョン **0.1.3** についてサブスクライブされたチャネルのアップグレードを検出します。これは、古いバージョンでインストールされていない CSV バージョン **0.1.2** を置き換えます。その後、さらに古いインストールされた CSV バージョン **0.1.1** を置き換えます。

OLM は、チャネルヘッドから CSV で指定された **replaces** フィールドで以前のバージョンに戻り、アップグレードパス **0.1.3 → 0.1.2 → 0.1.1** を判別します。矢印の方向は前者が後者を置き換えることを示します。OLM は、チャネルヘッドに到達するまで Operator を 1 バージョンずつアップグレードします。

このシナリオでは、OLM は Operator バージョン **0.1.2** をインストールし、既存の Operator バージョン **0.1.1** を置き換えます。その後、Operator バージョン **0.1.3** をインストールし、直前にインストールされた Operator バージョン **0.1.2** を置き換えます。この時点で、インストールされた Operator のバージョン **0.1.3** はチャネルヘッドに一致し、アップグレードは完了します。

2.4.3.1.2. アップグレードの省略

OLM のアップグレードの基本パスは以下の通りです。

- カタログソースは Operator への 1 つ以上の更新によって更新されます。
- OLM は、カタログソースに含まれる最新バージョンに到達するまで、Operator のすべてのバージョンを横断します。

ただし、この操作の実行は安全でない場合があります。公開されているバージョンの Operator がクラスターにインストールされていない場合、そのバージョンによって深刻な脆弱性が導入される可能性があるなどの理由でその Operator をがクラスターにインストールできないことがあります。

この場合、OLM は以下の 2 つのクラスターの状態を考慮に入れて、それらの両方に対応する更新グラフを提供する必要があります。

- 問題のある中間 Operator がクラスターによって確認され、かつインストールされている。
- 問題のある中間 Operator がクラスターにまだインストールされていない。

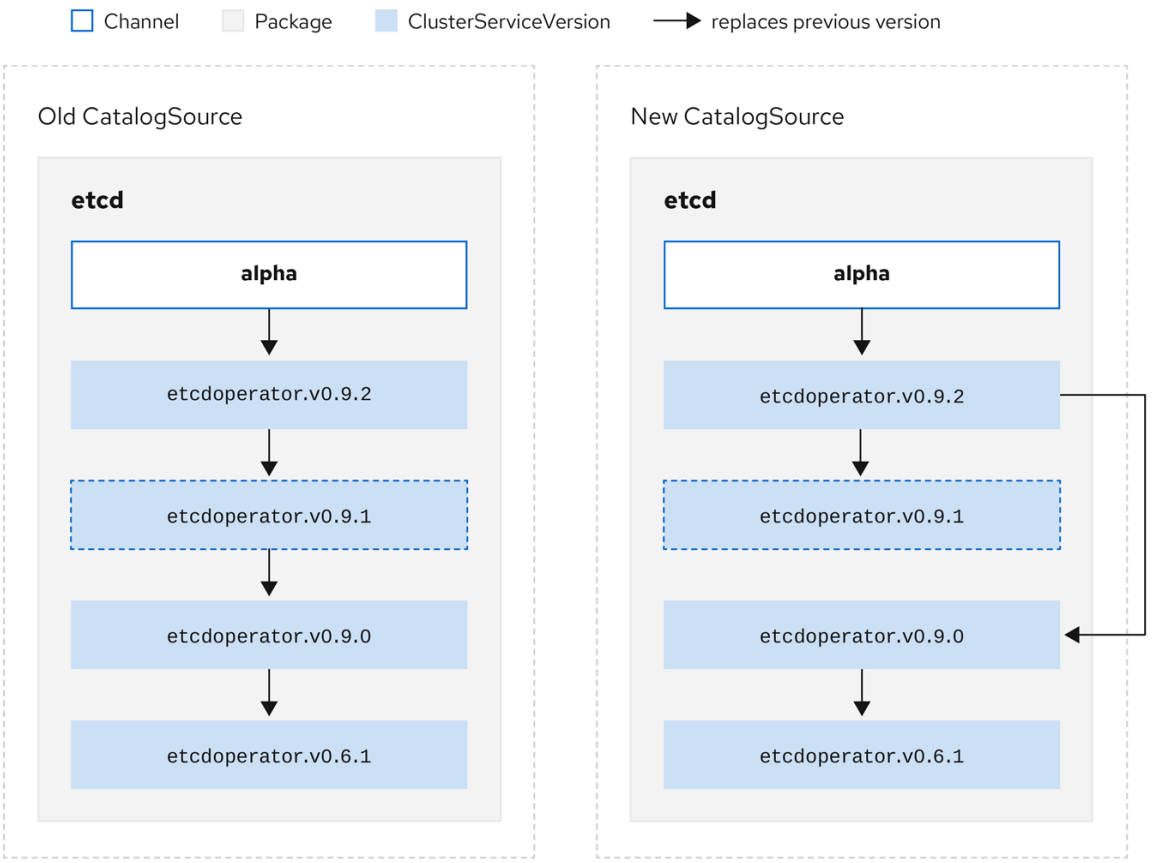
OLM は、新規カタログを送り、省略されたリリースを追加することで、クラスターの状態や問題のある更新が発見されたかどうかにかかわらず、単一の固有の更新を常に取得することができます。

省略されたリリースの CSV 例

```
apiVersion: operators.coreos.com/v1alpha1
kind: ClusterServiceVersion
metadata:
  name: etcdoperator.v0.9.2
  namespace: placeholder
  annotations:
spec:
    displayName: etcd
    description: Etcd Operator
    replaces: etcdoperator.v0.9.0
    skips:
    - etcdoperator.v0.9.1
```

古い **CatalogSource** および 新規 **CatalogSource** についての以下の例を見てみましょう。

図2.6 更新のスキップ



このグラフは、以下を示しています。

- 古い **CatalogSource** の Operator には、 新規 **CatalogSource** の単一の置き換えがある。
- 新規 **CatalogSource** の Operator には、 新規 **CatalogSource** の単一の置き換えがある。
- 問題のある更新がインストールされていない場合、これがインストールされることはない。

2.4.3.1.3. 複数 Operator の置き換え

説明されているように 新規 **CatalogSource** を作成するには、1 つの Operator を置き換える (置き換える) が、複数バージョンを省略 (**skip**) できる CSV を公開する必要があります。これは、**skipRange** アノテーションを使用して実行できます。

```
olm.skipRange: <semver_range>
```

ここで **<semver_range>** には、semver ライブラリー でサポートされるバージョン範囲の形式が使用されます。

カタログで更新を検索する場合、チャネルのヘッドに **skipRange** アノテーションがあり、現在インストールされている Operator にその範囲内のバージョンフィールドがある場合、OLM はチャネル内の最新エントリーに対して更新されます。

以下は動作が実行される順序になります。

1. サブスクリプションの **sourceName** で指定されるソースのチャネルヘッド (省略する他の条件が満たされている場合)。

2. **sourceName** で指定されるソースの現行バージョンを置き換える次の Operator。
3. サブスクリプションに表示される別のソースのチャネルヘッド (省略する他の条件が満たされている場合)。
4. サブスクリプションに表示されるソースの現行バージョンを置き換える次の Operator。

skipRange を含む CSV の例

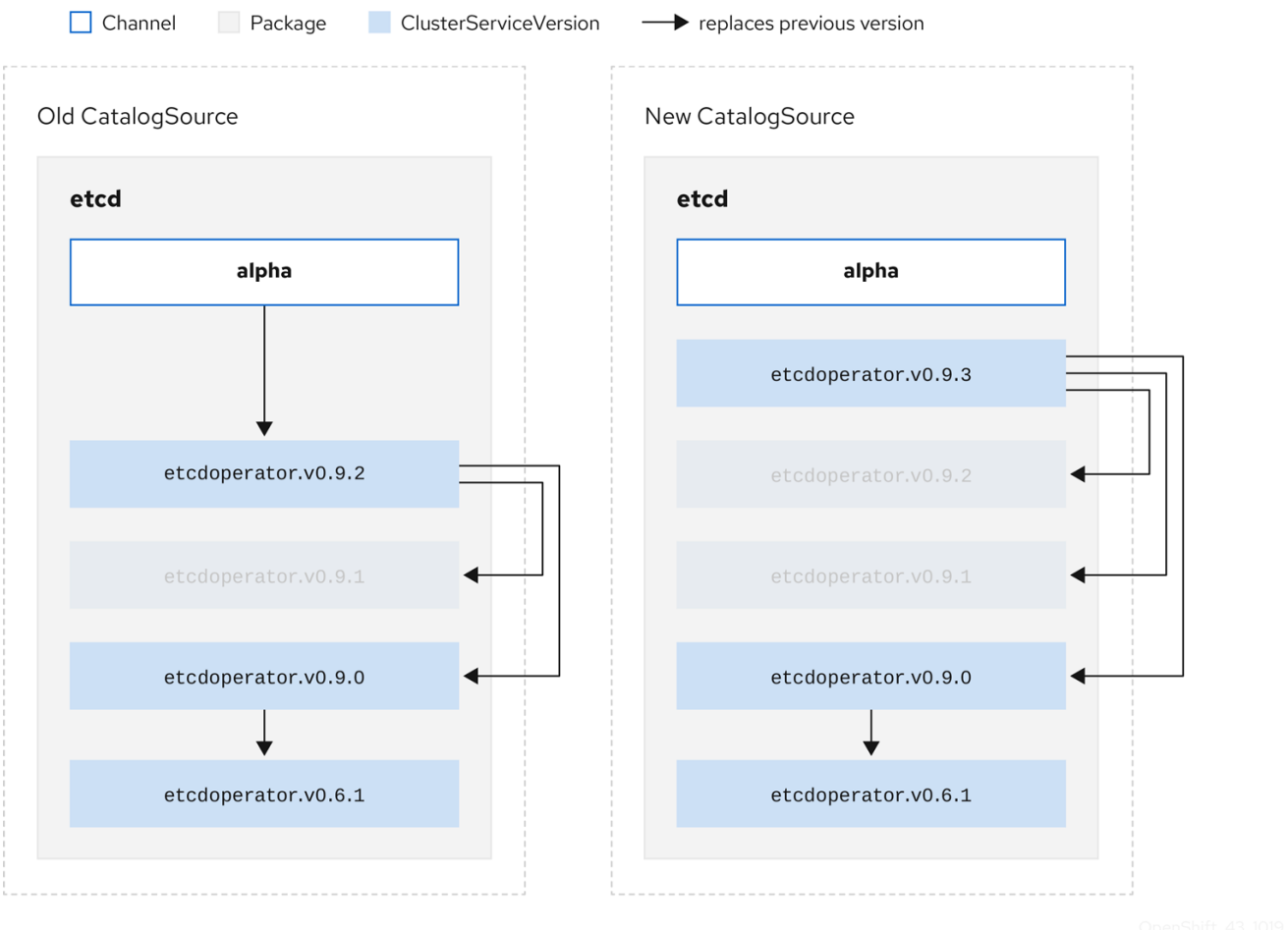
```
apiVersion: operators.coreos.com/v1alpha1
kind: ClusterServiceVersion
metadata:
  name: elasticsearch-operator.v4.1.2
  namespace: <namespace>
  annotations:
    olm.skipRange: '>=4.1.0 <4.1.2'
```

2.4.3.1.4. z-stream サポート

z-streamまたはパッチリリースは、同じマイナーバージョンの以前のすべての z-stream リリースを置き換える必要があります。OLM は、メジャー、マイナーまたはパッチバージョンを考慮せず、カタログ内で正確なグラフのみを作成する必要があります。

つまり、OLM では 古い **CatalogSource** のようにグラフを使用し、以前と同様に 新規 **CatalogSource** にあるようなグラフを生成する必要があります。

図2.7 複数 Operator の置き換え



このグラフは、以下を示しています。

- 古い **CatalogSource** の Operator には、 新規 **CatalogSource** の単一の置き換えがある。
- 新規 **CatalogSource** の Operator には、 新規 **CatalogSource** の単一の置き換えがある。
- 古い **CatalogSource** の z-stream リリースは、 新規 **CatalogSource** の最新 z-stream リリースに更新される。
- 使用不可のリリースは仮想グラフノードと見なされる。それらのコンテンツは存在する必要がなく、レジストリーはグラフが示すように応答することのみが必要になります。

2.4.4. Operator Lifecycle Manager の依存関係の解決

ここでは、OpenShift Dedicated における Operator Lifecycle Manager (OLM) を使用した依存関係の解決とカスタムリソース定義 (CRD) のアップグレードライフサイクルの概要を説明します。

2.4.4.1. 依存関係の解決

Operator Lifecycle Manager (OLM) は、実行中の Operator の依存関係の解決とアップグレードのライフサイクルを管理します。多くの場合、OLM が直面する問題は、 **yum**や**rpm**などの他のシステムまたは言語パッケージマネージャーと同様です。

ただし、OLM にはあるものの、通常同様のシステムにはない 1 つの制約があります。Opearator は常に実行されており、OLM は相互に機能しない Operator のセットの共存を防ごうとします。

その結果、以下のシナリオで OLM を使用しないでください。

- 提供できない API を必要とする Operator のセットのインストール
- Operator と依存関係のあるものに障害を発生させる仕方での Operator の更新

これは、次の 2 種類のデータで可能になります。

プロパティー	Operator に関する型付きのメタデータ。これは、依存関係のリゾルバーで Operator の公開インターフェイスを設定します。例としては、Operator が提供する API の group/version/kind (GVK) や Operator のセマンティックバージョン (semver) などがあります。
制約または依存関係	ターゲットクラスターにすでにインストールされているかどうかに関係なく、他の Operator が満たす必要のある Operator の要件。これらは、使用可能なすべての Operator に対するクエリーまたはフィルターとして機能し、依存関係の解決およびインストール中に選択を制限します。クラスターで特定の API が利用できる状態にする必要がある場合や、特定のバージョンに特定の Operator をインストールする必要がある場合など、例として挙げられます。

OLM は、これらのプロパティーと制約をブール式のシステムに変換して SAT ソルバーに渡します。これは、ブールの充足可能性を確立するプログラムであり、インストールする Operator を決定する作業を行います。

2.4.4.2. Operator のプロパティー

カタログ内の Operator にはすべて、次のプロパティーが含まれます。

olm.package

パッケージの名前と Operator のバージョンを含めます。

olm.gvk

クラスターサービスバージョン (CSV) から提供された API ごとに 1 つのプロパティー

追加のプロパティーは、Operator バンドルの **metadata**/ディレクトリーに**properties.yaml**ファイルを追加して、Operator 作成者が直接宣言することもできます。

任意のプロパティーの例

```
properties:
- type: olm.kubeversion
  value:
    version: "1.16.0"
```

2.4.4.2.1. 任意のプロパティー

Operator の作成者は、Operator バンドルの**metadata**/ ディレクトリーにある**properties.yaml**ファイルで任意のプロパティーを宣言できます。これらのプロパティーは、実行時に Operator Lifecycle Manager (OLM) リゾルバーへの入力として使用されるマップデータ構造に変換されます。

これらのプロパティーはリゾルバーには不透明です。リゾルバーはプロパティーについて理解しませんが、これらのプロパティーに対する一般的な制約を評価して、プロパティーリストを指定することで制約を満たすことができるかどうかを判断します。

任意のプロパティーの例

```
properties:
  - property:
      type: color
      value: red
  - property:
      type: shape
      value: square
  - property:
      type: olm.gvk
      value:
        group: olm.coreos.io
        version: v1alpha1
        kind: myresource
```

この構造を使用して、ジェネリック制約の Common Expression Language (CEL) 式を作成できます。

関連情報

- Common Expression Language (CEL) の制約

2.4.4.3. Operator の依存関係

Operator の依存関係は、バンドルの **metadata**/ フォルダー内の **dependencies.yaml** ファイルに一覧表示されます。このファイルはオプションであり、現時点では明示的な Operator バージョンの依存関係を指定するためにのみ使用されます。

依存関係の一覧には、依存関係の内容を指定するために各項目の **type** フィールドが含まれます。次のタイプの Operator 依存関係がサポートされています。

olm.package

このタイプは、特定の Operator バージョンの依存関係であることを意味します。依存関係情報には、パッケージ名とパッケージのバージョンを semver 形式で含める必要があります。たとえば、**0.5.2** などの特定バージョンや **>0.5.1** などのバージョンの範囲を指定することができます。

olm.gvk

このタイプの場合、作成者は CSV の既存の CRD および API ベースの使用方法と同様に group/version/kind (GVK) 情報で依存関係を指定できます。これは、Operator の作成者がすべての依存関係、API または明示的なバージョンを同じ場所に配置できるようにするパスです。

olm.constraint

このタイプは、任意の Operator プロパティーに対するジェネリック制約を宣言します。

以下の例では、依存関係は Prometheus Operator および etcd CRD について指定されます。

dependencies.yaml ファイルの例

```
dependencies:
  - type: olm.package
    value:
      packageName: prometheus
      version: ">0.27.0"
  - type: olm.gvk
    value:
      group: etcd.database.coreos.com
      kind: EtcdCluster
      version: v1beta2
```

2.4.4.4. 一般的な制約

olm.constraintプロパティーは、特定のタイプの依存関係制約を宣言し、非制約プロパティーと制約プロパティーを区別します。その値フィールドは、制約メッセージの文字列表現を保持する**failure Message**フィールドを含むオブジェクトです。このメッセージは、実行時に制約が満たされない場合に、ユーザーへの参考のコメントとして表示されます。

次のキーは、使用可能な制約タイプを示します。

gvk

値と解釈が **olm.gvk** タイプと同じタイプ

package

値と解釈が**olm.package**タイプと同じタイプ

cel

任意のバンドルプロパティーとクラスター情報に対して Operator Lifecycle Manager (OLM) リゾルバーによって実行時に評価される Common Expression Language (CEL) 式

all、any、not

gvkやネストされた複合制約など、1 つ以上の具体的な制約を含む、論理積、論理和、否定の制約。

2.4.4.4.1. Common Expression Language (CEL) の制約

cel制約型は、式言語として[Common Expression Language (CEL)]を[サポート]しており、**cel**構造に

cel 制約型は、式言語として Common Expression Language (CEL)をサポートしています。**cel** 構造には、Operator が制約を満たしているかどうかを判断するために、実行時に Operator プロパティーに対して評価される CEL 式文字列を含む **rule** フィールドがあります。

cel 制約の例

```
type: olm.constraint
value:
  failureMessage: 'require to have "certified"'
  cel:
    rule: 'properties.exists(p, p.type == "certified")'
```

CEL 構文は、**AND** や **OR** などの幅広い論理演算子をサポートします。その結果、単一の CEL 式は、これらの論理演算子で相互にリンクされる複数の条件に対して複数のルールを含めることができます。これらのルールは、バンドルまたは任意のソースからの複数の異なるプロパティーのデータセットに対して評価され、出力は、単一の制約内でこれらのルールのすべてを満たす単一のバンドルまたは Operator に対して解決されます。

複数のルールが指定されたcel制約の例

```
type: olm.constraint
value:
  failureMessage: 'require to have "certified" and "stable" properties'
  cel:
    rule: 'properties.exists(p, p.type == "certified") && properties.exists(p, p.type == "stable")'
```

2.4.4.4.2. 複合制約 (all, any, not)

複合制約タイプは、論理定義に従って評価されます。

以下は、2 つのパッケージと 1 つの GVK の接続制約 (**all**) の例です。つまり、インストールされたバンドルがすべての制約を満たす必要があります。

all制約の例

```
schema: olm.bundle
name: red.v1.0.0
properties:
- type: olm.constraint
  value:
    failureMessage: All are required for Red because...
    all:
      constraints:
      - failureMessage: Package blue is needed for...
        package:
          name: blue
          versionRange: '>=1.0.0'
      - failureMessage: GVK Green/v1 is needed for...
        gvk:
          group: greens.example.com
          version: v1
          kind: Green
```

以下は、同じGVKの2つのバージョンの選言的制約（or）の例です。一つまたはインストールされたバ

以下は、同じ GVK の 3 つのバージョンの選言的制約 (**any**) の例です。つまり、インストールされたバンドルが少なくとも 1 つの制約を満たす必要があります。

any 制約の例

```
schema: olm.bundle
name: red.v1.0.0
properties:
- type: olm.constraint
  value:
    failureMessage: Any are required for Red because...
    any:
      constraints:
      - gvk:
          group: blues.example.com
          version: v1beta1
          kind: Blue
      - gvk:
          group: blues.example.com
          version: v1beta2
          kind: Blue
      - gvk:
          group: blues.example.com
          version: v1
          kind: Blue
```

以下は、GVK の 1 つのバージョンの否定制約 (**not**) の例です。つまり、この結果セットのバンドルでは、この GVK を提供できません。

not の制約例

```
schema: olm.bundle
name: red.v1.0.0
properties:
- type: olm.constraint
  value:
  all:
    constraints:
    - failureMessage: Package blue is needed for...
      package:
        name: blue
        versionRange: '>=1.0.0'
    - failureMessage: Cannot be required for Red because...
      not:
        constraints:
        - gvk:
            group: greens.example.com
            version: v1alpha1
            kind: greens
```

否定のセマンティクスは、**not**制約のコンテキストで不明確であるように見える場合があります。つまり、この否定では、特定の GVK、あるバージョンのパッケージを含むソリューション、または結果セットからの子の複合制約を満たすソリューションを削除するように、リゾルバーに対して指示を出しています。

当然の結果として、最初に可能な依存関係のセットを選択せずに否定することは意味がないため、複合では**not**制約は**all**または**any**制約内でのみ使用する必要があります。

2.4.4.4.3. ネストされた複合制約

ネストされた複合制約 (少なくとも 1 つの子複合制約と 0 個以上の単純な制約を含む制約) は、前述の各制約タイプの手順に従って、下から上に評価されます。

以下は、接続詞の論理和の例で、one、the other、または both が制約を満たすことができます。

ネストされた複合制約の例

```
schema: olm.bundle
name: red.v1.0.0
properties:
- type: olm.constraint
  value:
    failureMessage: Required for Red because...
    any:
      constraints:
      - all:
          constraints:
          - package:
              name: blue
              versionRange: '>=1.0.0'
          - gvk:
              group: blues.example.com
              version: v1
              kind: Blue
      - all:
          constraints:
          - package:
              name: blue
              versionRange: '<1.0.0'
          - gvk:
              group: blues.example.com
              version: v1beta1
              kind: Blue
```

注記

olm.constraintタイプの最大 raw サイズは 64KB に設定されており、リソース枯渇攻撃を制限しています。

2.4.4.5. 依存関係の設定

Operator の依存関係を同等に満たすオプションが多数ある場合があります。Operator Lifecycle Manager (OLM) の依存関係リゾルバーは、要求された Operator の要件に最も適したオプションを判別します。Operator の作成者またはユーザーとして、依存関係の解決が明確になるようにこれらの選択方法を理解することは重要です。

2.4.4.5.1. カタログの優先順位

OpenShift Dedicated クラスターでは、OLM がカタログソースを読み取り、どの Operator がインストール可能であるかを確認します。

CatalogSource オブジェクトの例

```
apiVersion: "operators.coreos.com/v1alpha1"
kind: "CatalogSource"
metadata:
  name: "my-operators"
  namespace: "operators"
spec:
  sourceType: grpc
  grpcPodConfig:
    securityContextConfig: <security_mode> 1
  image: example.com/my/operator-index:v1
  displayName: "My Operators"
  priority: 100
```

1 **legacy** または **restricted** の値を指定します。フィールドが設定されていない場合、デフォルト値は **legacy** です。今後の OpenShift Dedicated リリースでは、デフォルト値が **restricted** になる予定です。**restricted** 権限でカタログを実行できない場合は、このフィールドを手動で **legacy** に設定することを推奨します。

CatalogSource オブジェクトには **priority** フィールドがあります。このフィールドは、依存関係のオプションを優先する方法を把握するためにリゾルバーによって使用されます。

カタログ設定を規定する 2 つのルールがあります。

- 優先順位の高いカタログにあるオプションは、優先順位の低いカタログのオプションよりも優先されます。
- 依存オブジェクトと同じカタログにあるオプションは他のカタログよりも優先されます。

2.4.4.5.2. チャネルの順序付け

カタログ内の Operator パッケージは、OpenShift Dedicated クラスターでユーザーがサブスクライブできる更新チャネルのコレクションです。チャネルは、マイナーリリース (**1.2**、**1.3**) またはリリース頻度 (**stable**、**fast**) についての特定の更新ストリームを提供するために使用できます。

同じパッケージの Operator によって依存関係が満たされる可能性がありますが、その場合、異なるチャネルの Operator のバージョンによって満たされる可能性があります。たとえば、Operator のバージョン **1.2** は **stable** および **fast** チャネルの両方に存在する可能性があります。

それぞれのパッケージにはデフォルトのチャネルがあり、これは常にデフォルト以外のチャネルよりも優先されます。デフォルトチャネルのオプションが依存関係を満たさない場合には、オプションは、チャネル名の辞書式順序 (lexicographic order) で残りのチャネルから検討されます。

2.4.4.5.3. チャネル内での順序

ほとんどの場合、単一のチャネル内に依存関係を満たすオプションが複数あります。たとえば、1 つのパッケージおよびチャネルの Operator は同じセットの API を提供します。

ユーザーがサブスクリプションを作成すると、それらはどのチャネルから更新を受け取るかを示唆します。これにより、すぐにその 1 つのチャネルだけに検索が絞られます。ただし、チャネル内では、多くの Operator が依存関係を満たす可能性があります。

チャネル内では、更新グラフでより上位にある新規 Operator が優先されます。チャネルのヘッドが依存関係を満たす場合、これがまず試行されます。

2.4.4.5.4. その他の制約

OLM には、パッケージの依存関係で指定される制約のほかに、必要なユーザーの状態を表し、常にメンテナンスする必要のある依存関係の解決を適用するための追加の制約が含まれます。

2.4.4.5.4.1. サブスクリプションの制約

サブスクリプションの制約は、サブスクリプションを満たすことのできる Operator のセットをフィルターします。サブスクリプションは、依存関係リゾルバーについてのユーザー指定の制約です。それらは、クラスター上にない場合は新規 Operator をインストールすることを宣言するか、既存 Operator の更新された状態を維持することを宣言します。

2.4.4.5.4.2. パッケージの制約

namespace 内では、2 つの Operator が同じパッケージから取得されることはありません。

2.4.4.5.5. 関連情報

- カタログの正常性要件

2.4.4.6. CRD のアップグレード

OLM は、単一のクラスターサービスバージョン (CSV) によって所有されている場合にはカスタムリソース定義 (CRD) をすぐにアップグレードします。CRD が複数の CSV によって所有されている場合、CRD は、以下の後方互換性の条件のすべてを満たす場合にアップグレードされます。

- 現行 CRD の既存の有効にされたバージョンすべてが新規 CRD に存在する。
- 検証が新規 CRD の検証スキーマに対して行われる場合、CRD の提供バージョンに関連付けられる既存インスタンスまたはカスタムリソースすべてが有効である。

関連情報

- 新規 CRD バージョンの追加
- CRD バージョンの非推奨または削除

2.4.4.7. 依存関係のベストプラクティス

依存関係を指定する際には、ベストプラクティスを考慮する必要があります。

Operator の API または特定のバージョン範囲によって異なります。

Operator は API をいつでも追加または削除できます。Operator が必要とする API に **olm.gvk** 依存関係を常に指定できます。これの例外は、**olm.package** 制約を代わりに指定する場合です。

最小バージョンの設定

API の変更に関する Kubernetes ドキュメントでは、Kubernetes 形式の Operator で許可される変更について説明しています。これらのバージョン管理規則により、Operator は API バージョンに後方互換性がある限り、API バージョンに影響を与えずに API を更新することができます。
Operator の依存関係の場合、依存関係の API バージョンを把握するだけでは、依存する Operator が確実に意図された通りに機能することを確認できないことを意味します。

以下に例を示します。

- TestOperator v1.0.0 は、v1alpha1 API バージョンの **MyObject** リソースを提供します。
- TestOperator v1.0.1 は新しいフィールド **spec.newfield** を **MyObject** に追加しますが、v1alpha1 のままになります。

Operator では、**spec.newfield** を **MyObject** リソースに書き込む機能が必要になる場合があります。**olm.gvk** 制約のみでは、OLM で TestOperator v1.0.0 ではなく TestOperator v1.0.1 が必要であると判断することはできません。

可能な場合には、API を提供する特定の Operator が事前に分かっている場合、最小値を設定するために追加の **olm.package** 制約を指定します。

最大バージョンを省略するか、幅広いバージョンを許可します。

Operator は API サービスや CRD などのクラスタースコープのリソースを提供するため、依存関係に小規模な範囲を指定する Operator は、その依存関係の他のコンシューマーの更新に不要な制約を加える可能性があります。
可能な場合は、最大バージョンを設定しないでください。または、他の Operator との競合を防ぐために、幅広いセマンティクスの範囲を設定します。例: **>1.0.0 <2.0.0**

従来のパッケージマネージャーとは異なり、Operator の作成者は更新が OLM のチャネルで更新を安全に行われるように Operator を明示的にエンコードします。更新が既存のサブスクリプションで利用可能な場合、Operator の作成者がこれが以前のバージョンから更新できることを示唆していることが想定されます。依存関係の最大バージョンを設定すると、特定の上限で不必要な切り捨てが行われることにより、作成者の更新ストリームが上書きされます。

注記

クラスター管理者は、Operator の作成者が設定した依存関係を上書きすることはできません。

ただし、回避する必要がある非互換性があることが分かっている場合は、最大バージョンを設定でき、およびこれを設定する必要があります。特定のバージョンは、バージョン範囲の構文 (例: **1.0.0 !1.2.1**) で省略できます。

関連情報

- Kubernetes ドキュメント: Changing the API

2.4.4.8. 依存関係に関する注意事項

依存関係を指定する際には、考慮すべき注意事項があります。

複合制約がない (AND)

現時点で、制約の間に AND 関係を指定する方法はありません。つまり、ある Operator が、所定の API を提供し、バージョン **>1.1.0** を持つ別の Operator に依存するように指定することはできません。
依存関係を指定すると、以下のようになります。

```
dependencies:
- type: olm.package
  value:
```

```
    packageName: etcd
    version: ">3.1.0"
- type: olm.gvk
  value:
    group: etcd.database.coreos.com
    kind: EtcdCluster
    version: v1beta2
```

OLM は EtcdCluster を提供する Operator とバージョン **>3.1.0** を持つ Operator の 2 つの Operator で、上記の依存関係の例の条件を満たすことができる可能性があります。その場合や、または両方の制約を満たす Operator が選択されるかどうかは、選択できる可能性のあるオプションが参照される順序によって変わります。依存関係の設定および順序のオプションは十分に定義され、理にかなったものであると考えられますが、Operator は継続的に特定のメカニズムをベースとする必要があります。

namespace 間の互換性

OLM は namespace スコープで依存関係の解決を実行します。ある namespace での Operator の更新が別の namespace の Operator の問題となる場合、更新のデッドロックが生じる可能性があります。

2.4.4.9. 依存関係解決のシナリオ例

以下の例で、プロバイダー は CRD または API サービスを所有する Operator です。

例: 依存 **API** を非推奨にする
A および B は API (CRD):

- A のプロバイダーは B によって異なる。
- B のプロバイダーにはサブスクリプションがある。
- B のプロバイダーは C を提供するように更新するが、B を非推奨にする。

この結果は以下のようになります。

- B にはプロバイダーがなくなる。
- A は機能しなくなる。

これは OLM がアップグレードストラテジーで回避するケースです。

例: バージョンのデッドロック
A および B は API である:

- A のプロバイダーは B を必要とする。
- B のプロバイダーは A を必要とする。
- A のプロバイダーは (A2 を提供し、B2 を必要とするように) 更新し、A を非推奨にする。
- B のプロバイダーは (B2 を提供し、A2 を必要とするように) 更新し、B を非推奨にする。

OLM が B を同時に更新せずに A を更新しようとする場合や、その逆の場合、OLM は、新しい互換性のあるセットが見つかったとしても Operator の新規バージョンに進むことができません。

これは OLM がアップグレードストラテジーで回避するもう 1 つのケースです。

2.4.5. Operator グループ

ここでは、OpenShift Dedicated における Operator Lifecycle Manager (OLM) での Operator グループの使用について概説します。

2.4.5.1. Operator グループについて

Operator グループ は、 **OperatorGroup** リソースによって定義され、マルチテナント設定を OLM でインストールされた Operator に提供します。Operator グループは、そのメンバー Operator に必要な RBAC アクセスを生成するために使用するターゲット namespace を選択します。

ターゲット namespace のセットは、クラスターサービスバージョン (CSV) の **olm.targetNamespaces** アノテーションに保存されるコ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。

2.4.5.2. Operator グループメンバーシップ

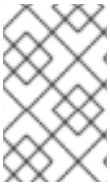
Operator は、以下の条件が true の場合に Operator グループの メンバー とみなされます。

- Operator の CSV が Operator グループと同じ namespace にある。
- Operator の CSV のインストールモードは Operator グループがターゲットに設定する namespace のセットをサポートする。

CSV のインストールモードは **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ます。CSV の仕様には、4 つの固有の **InstallModeTypes** のインストールモードのセットを含めることができます。

表2.5 インストールモードおよびサポートされる **Operator** グループ

InstallMode タイプ	説明
OwnNamespace	Operator は、独自の namespace を選択する Operator グループ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 Operator グループ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 Operator グループ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 Operator グループのメンバーにすることができます (設定されるターゲット namespace 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。

2.4.5.3. ターゲット namespace の選択

spec.targetNamespaces パラメーターを使用して Operator グループのターゲット namespace に名前を明示的に指定することができます。

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

または、**spec.selector** パラメーターでラベルセレクターを使用して namespace を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:
  selector:
    cool.io/prod: "true"
```

重要

spec.targetNamespaces で複数の namespace をリスト表示したり、**spec.selector** でラベルセレクターを使用したりすることは推奨されません。Operator グループの複数のターゲット namespace のサポートは今後のリリースで取り除かれる可能性があります。

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** Operator グループを指定できます。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
```

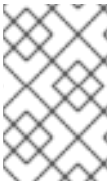
選択された namespace の解決済みのセットは Operator グループの **status.namespaces** パラメーターに表示されます。グローバル Operator グループの **status.namespace** には空の文字列 (**""**) が含まれます。 これは、消費する Operator に対し、すべての namespace を監視するように示唆します。

2.4.5.4. Operator グループの CSV アノテーション

Operator グループのメンバー 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>	Operator グループの名前が含まれます。

アノテーション	説明
olm.operatorNamespace=<group_namespace>	Operator グループの namespace が含まれます。
olm.targetNamespaces=<target_namespaces>	Operator グループのターゲット namespace 選択をリスト表示するコ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。

2.4.5.5. 提供される API アノテーション

group/version/kind(GVK) は Kubernetes API の一意の識別子です。Operator グループによって提供される 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、コンマで区切られた **<kind>.<version>.<group>** で構成される文字列です。Operator グループ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 API サービスの GVK が含まれます。

PackageManifest リースを提供する単一のアクティブメンバー CSV を含む **OperatorGroup** オブジェクト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
```

2.4.5.6. ロールベースのアクセス制御

Operator グループの作成時に、3 つのクラスターロールが生成されます。それぞれには、以下に示すようにクラスターロールセレクターがラベルに一致するように設定された単一の集計ルールが含まれます。

クラスターロール	一致するラベル
olm.og.<operatorgroup_name>-admin-<hash_value>	**olm.opgroup.permissions/aggregate-to-admin: <operatorgroup_name>**
olm.og.<operatorgroup_name>-edit-<hash_value>	**olm.opgroup.permissions/aggregate-to-edit: <operatorgroup_name>**
olm.og.<operatorgroup_name>-view-<hash_value>	**olm.opgroup.permissions/aggregate-to-view: <operatorgroup_name>**

以下の RBAC リソースは、CSV が **AllNamespaces** インストールモードのあるすべての namespace を監視しており、理由が **InterOperatorGroupOwnerConflict** の失敗状態にない限り、CSV が Operator グループ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 リソースのクラスターロール
- API サービスからの各 API リソースのクラスターロール
- 追加のロールおよびロールバインディング

表2.6 CRD からの各 API リソース用に生成されたクラスターロール

クラスターロール	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>**

クラスターロール	設定
<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2.7 API サービスから各 API リソース用に生成されたクラスターロール

クラスターロール	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>**

クラスターロール	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>**

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、クラスターロールと対応するクラスターロールバインディングが CSV の **permissions** 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。

2.4.5.7. コピーされる CSV

OLM は、それぞれの Operator グループのターゲット namespace の Operator グループ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、それらのソース CSV が属する Operator グループが、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。

注記

デフォルトでは、**disableCopiedCSVs** フィールドは無効になっています。**disableCopiedCSVs** フィールドを有効にすると、OLM はクラスター上の既存のコピーされた CSV を削除します。**disableCopiedCSVs** フィールドが無効になると、OLM はコピーされた CSV を再度追加します。

- **disableCopiedCSVs** フィールドを無効にします。

```
$ cat << EOF | oc apply -f -
apiVersion: operators.coreos.com/v1
kind: OLMConfig
metadata:
  name: cluster
spec:
  features:
    disableCopiedCSVs: false
EOF
```

- **disableCopiedCSVs** フィールドを有効にします。

```
$ cat << EOF | oc apply -f -
apiVersion: operators.coreos.com/v1
kind: OLMConfig
metadata:
  name: cluster
spec:
  features:
    disableCopiedCSVs: true
EOF
```

2.4.5.8. 静的 Operator グループ

Operator グループはその **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 Operator グループの **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 Operator グループを使用して 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 Operator グループの例です。

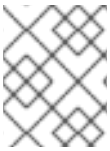
```
apiVersion: operators.coreos.com/v1
kind: OperatorGroup
metadata:
  name: cluster-monitoring
```

```
  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"
```

2.4.5.9. Operator グループの交差部分

2 つの Operator グループは、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 Operator グループは、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、Operator グループの 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。

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 Operator グループとそれ以外のすべての間での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 Operator グループ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 グループのサブセット ではない 場合:
  - Operator グループが静的である場合:
    - 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。
  - Operator グループが静的 ではない 場合:
    - Operator グループの **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 Operator グループのサブセット 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 の指定 API が Operator グループのサブセットである場合:

- Operator グループが静的である場合:
  - 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。
- Operator グループが静的 ではない 場合:
  - Operator グループの **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

Operator グループ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、Operator グループ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。

2.4.5.10. マルチテナント Operator 管理の制限事項

OpenShift Dedicated では、同じクラスターに異なる Operator のバージョンを同時にインストールした場合、サポートが限定的になります。Operator Lifecycle Manager (OLM) は、Operator を異なる namespace に複数回インストールします。その 1 つの制約として、Operator の API バージョンは同じである必要があります。

Operator は、Kubernetes のグローバルリソースである **CustomResourceDefinition** オブジェクト (CRD) を使用するため、コントロールプレーンの拡張機能です。多くの場合、Operator の異なるメジャーバージョンには互換性のない CRD があります。これにより、クラスター上の異なる namespace に同時にインストールするのに互換性がなくなります。

すべてのテナントまたは namespace がクラスターの同じコントロールプレーンを共有します。したがって、マルチテナントクラスター内のテナントはグローバル CRD も共有するため、同じクラスターで同じ Operator の異なるインスタンスを並行して使用できるシナリオが制限されます。

サポートされているシナリオは次のとおりです。

- まったく同じ CRD 定義を提供する異なるバージョンの Operator (バージョン管理された CRD の場合は、まったく同じバージョンのセット)
- CRD を同梱せず、代わりに OperatorHub の別のバンドルで CRD を利用できる異なるバージョンの Operator

他のすべてのシナリオはサポートされていません。これは、異なる Operator バージョンからの複数の競合または重複する CRD が同じクラスター上で調整される場合、クラスターデータの整合性が保証されないためです。

関連情報

- マルチテナントクラスター内の Operator

2.4.5.11. Operator グループのトラブルシューティング

メンバーシップ

- インストールプランの namespace には、Operator グループを 1 つだけ含める必要があります。namespace でクラスターサービスバージョン (CSV) を生成しようとすると、インストールプランでは、以下のシナリオの Operator グループが無効であると見なされます。
  - インストールプランの namespace に Operator グループが存在しない。
  - インストールプランの namespace に複数の Operator グループが存在する。
  - Operator グループに、正しくないサービスアカウント名または存在しないサービスアカウント名が指定されている。

  インストールプランで無効な Operator グループが検出された場合には、CSV は生成されず、**InstallPlan** リソースは関連するメッセージを出力して、インストールを続行します。たとえば、複数の Operator グループが同じ namespace に存在する場合に以下のメッセージが表示されます。

  ```
  attenuated service account query failed - more than one operator group(s) are managing this namespace count=2
  ```

  ここでは、**count=** は、namespace 内の Operator グループの数を指します。

- CSV のインストールモードがその namespace で Operator グループのターゲット 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、Operator グループのターゲット namespace の選択がサポートされる設定に変更されるか、CSV のインストールモードがターゲット 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。

2.4.6. マルチテナント対応と Operator のコロケーション

このガイドでは、Operator Lifecycle Manager (OLM) のマルチテナント対応と Operator のコロケーションについて説明します。

2.4.6.1. Colocation of Operators in a namespace

Operator Lifecycle Manager (OLM) は、同じ namespace にインストールされている OLM 管理の Operator を処理します。つまり、それらの **Subscription** リソースは、関連する Operator として同じ namespaceに配置されます。それらが実際には関連していなくても、いずれかが更新されると、OLM はバージョンや更新ポリシーなどの状態を考慮します。

このデフォルトの動作は、次の 2 つの方法で現れます。

- 保留中の更新の **InstallPlan** リソースには、同じ namespace にある他のすべての Operator の **ClusterServiceVersion** (CSV) リソースが含まれます。
- 同じ namespace 内のすべての Operator は、同じ更新ポリシーを共有します。たとえば、1 つの Operator が手動更新に設定されている場合、他のすべての Operator の更新ポリシーも手動に設定されます。

これらのシナリオは、次の問題につながる可能性があります。

- 更新された Operator だけでなく、より多くのリソースが定義されているため、Operator 更新のインストール計画について推論するのは難しくなります。
- namespace 内の一部の Operator を自動で更新し、他の Operator を手動で更新するという、多くのクラスター管理者が求めていることが不可能になります。

通常、これらの問題が明らかになるのは、OpenShift Dedicated を使用して Operator をインストールするときに、デフォルトの動作により、**All namespaces** インストールモードをサポートする Operator が、デフォルトの **openshift-operators** グローバル namespace にインストールされるためです。

dedicated-admin ロールを持つ管理者は、次のワークフローを使用して、このデフォルトの動作を手動で回避できます。

1. Operator をインストールするためのプロジェクトを作成します。
2. すべての namespace を監視する Operator グループである、カスタム グローバル **Operator group** を作成します。この Operator グループを作成した namespace に関連付けることで、インストール namespace がグローバル namespace になり、そこにインストールされた Operator がすべての namespace で使用できるようになります。
3. 必要な Operator をインストール namespace にインストールします。

Operator に依存関係がある場合、依存関係は事前に作成された namespace に自動的にインストールされます。その結果、依存関係 Operator が同じ更新ポリシーと共有インストールプランを持つことが有効になります。詳細な手順については、カスタム namespace へのグローバル Operator のインストールを参照してください。

関連情報

- Installing global Operators in custom namespaces
- マルチテナントクラスター内の Operator

2.4.7. Operator 条件

以下では、Operator Lifecycle Manager (OLM) による Operator 条件の使用方法について説明します。

2.4.7.1. Operator 条件について

Operator のライフサイクル管理のロールの一部として、Operator Lifecycle Manager (OLM) は、Operator を定義する Kubernetes リソースの状態から Operator の状態を推測します。このアプローチでは、Operator が特定の状態にあることをある程度保証しますが、推測できない情報を Operator が OLM と通信して提供する必要がある場合も多々あります。続いて、OLM がこの情報を使用して、Operator のライフサイクルをより適切に管理することができます。

OLM は、Operator が OLM に条件について通信できる **OperatorCondition** というカスタムリソース定義 (CRD) を提供します。**OperatorCondition** リソースの **Spec.Conditions** 配列にある場合に、OLM による Operator の管理に影響するサポートされる条件のセットがあります。

注記

デフォルトでは、 **Spec.Conditions**配列は、ユーザーによって追加されるか、カスタム Operator ロジックの結果として追加されるまで、 **Operator Condition**オブジェクトに存在しません。

2.4.7.2. サポートされる条件

Operator Lifecycle Manager (OLM) は、以下の Operator 条件をサポートします。

2.4.7.2.1. アップグレード可能な条件

Upgradeable Operator 条件は、既存のクラスターサービスバージョン (CSV) が、新規の CSV バージョンに置き換えられることを阻止します。この条件は、以下の場合に役に立ちます。

- Operator が重要なプロセスを開始するところで、プロセスが完了するまでアップグレードしてはいけない場合
- Operator が、Operator のアップグレードの準備ができる前に完了する必要のあるカスタムリソース (CR) の移行を実行している場合

重要

Upgradeable Operator の条件を **False** 値に設定しても、Pod の中断は回避できません。Pod が中断されないようにする必要がある場合は、「追加リソース」セクションの「Pod 中断バジェットを使用して稼働させなければならないPodの数を指定する」と「正常な終了」を参照してください。

Upgradeable Operator 条件の例

```
apiVersion: operators.coreos.com/v1
kind: OperatorCondition
metadata:
  name: my-operator
  namespace: operators
spec:
  conditions:
  - type: Upgradeable (1)
    status: "False" (2)
    reason: "migration"
    message: "The Operator is performing a migration."
    lastTransitionTime: "2020-08-24T23:15:55Z"
```

(1) 条件の名前。

(2) **False** 値は、Operator のアップグレードの準備ができていないことを示します。OLM は、Operator の既存の CSV を置き換える CSV が **Pending** フェーズでなくなることを阻止します。**False** 値はクラスターのアップグレードをブロックしません。

2.4.7.3. 関連情報

- Operator 条件の管理
- Operator 条件の有効化

2.4.8. Operator Lifecycle Manager メトリクス

2.4.8.1. 公開されるメトリクス

Operator Lifecycle Manager (OLM) は、Prometheus ベースの OpenShift Dedicated 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2.8 OLM によって公開されるメトリクス

名前	説明
catalog_source_count	カタログソースの数。
catalogsource_ready	カタログソースの状態。値 **1** は、カタログソースが **READY** 状態であることを示します。値 **0** は、カタログソースが **READY** 状態ではないことを示します。
csv_abnormal	クラスターサービスバージョン (CSV) を調整する際に、(インストールされていない場合など) CSV バージョンが **Succeeded** 以外の状態にあることを表します。**name**、**namespace**、 **phase**、**reason**、および **version** ラベルが含まれます。Prometheus アラートは、このメトリクスが存在する場合に作成されます。
csv_count	正常に登録された CSV の数。
csv_succeeded	CSV を調整する際に、CSV バージョンが **Succeeded** 状態 (値 **1**) にあるか、そうでないか (値 **0**) を表します。**name**、**namespace**、および **version** ラベルが含まれます。
csv_upgrade_count	CSV アップグレードの単調 (monotonic) カウント。
install_plan_count	インストール計画の数。
installplan_warnings_total	インストール計画に含まれる非推奨のリソースなど、リソースによって生成される警告の個数。
olm_resolution_duration_seconds	依存関係解決の試行期間。
subscription_count	サブスクリプションの数。
subscription_sync_total	サブスクリプション同期の単調 (monotonic) カウント。**channel**、**installed** CSV、およびサブスクリプション **name** ラベルが含まれます。

2.4.9. Operator Lifecycle Manager での Webhook の管理

Webhook により、リソースがオブジェクトストアに保存され、Operator コントローラーによって処理される前に、Operator の作成者はリソースのインターセプト、変更、許可、および拒否を実行することができます。Operator Lifecycle Manager (OLM) は、Operator と共に提供される際にこれらの Webhook のライフサイクルを管理できます。

Operator 開発者が自分の Operator に Webhook を定義する方法の詳細と、OLM で実行する場合の注意事項は、クラスターサービスのバージョン (CSV) を定義する を参照してください。

2.4.9.1. 関連情報

- Kubernetes ドキュメント:
  - 検証用の受付 Webhook
  - 変更用の受付 Webhook
  - 変換 Webhook

2.5. OPERATORHUB について

2.5.1. OperatorHub について

OperatorHub は OpenShift Dedicated の Web コンソールインターフェイスです。クラスター管理者はこれを使用して Operator を検出し、インストールします。1 回のクリックで、Operator をクラスター外のソースからプルし、クラスター上でインストールおよびサブスクライブして、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全体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カタログ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。
Red Hat Marketplace	Red Hat Marketplace から購入できる認定ソフトウェア。
コミュニティー Operator	redhat-openshift-ecosystem/community-operators-prod/operators GitHub リポジトリーで関連する担当者によって保守されているオプションで表示可能なソフトウェア。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 を追加していない場合、**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 の Operator は OLM で実行されるようにパッケージ化されます。これには、Operator のインストールおよびセキュアな実行に必要なすべての CRD、RBAC ルール、デプロイメント、およびコンテナーイメージが含まれるクラスターサービスバージョン (CSV) という YAML ファイルが含まれます。また、機能の詳細やサポートされる Kubernetes バージョンなどのユーザーに表示される情報も含まれます。

Operator SDK は、開発者が OLM および OperatorHub で使用するために Operator のパッケージ化することを支援するために使用できます。お客様によるアクセスが可能な商用アプリケーションがある場合、Red Hat Partner Connect ポータル (connect.redhat.com) で提供される認定ワークフローを使用し

てこれを組み込むようにしてください。

2.5.2. OperatorHub アーキテクチャー

OperatorHub UI コンポーネントは、デフォルトで、OpenShift Dedicated の **openshift-marketplace** namespace で Marketplace Operator によって実行されます。

2.5.2.1. OperatorHub カスタムリソース

Marketplace Operator は、OperatorHub で提供されるデフォルトの **CatalogSource** オブジェクトを管理する **cluster** という名前の **OperatorHub** カスタムリソース (CR) を管理します。

2.5.3. 関連情報

- カタログソース
- Operator SDK について
- クラスターサービスバージョン (CSV) の定義
- OLM での Operator のインストールおよびアップグレードのワークフロー
- Red Hat Partner Connect
- Red Hat Marketplace

2.6. RED HAT が提供する OPERATOR カタログ

Red Hat は、OpenShift Dedicated にデフォルトで含まれる複数の Operator カタログを提供します。

重要

OpenShift Dedicated 4.11 以降、デフォルトの Red Hat 提供の Operator カタログはファイルベースのカタログ形式でリリースされます。OpenShift Dedicated 4.6 ~ 4.10 用のデフォルトの Red Hat 提供 Operator カタログは、非推奨の SQLite データベース形式でリリースされました。

opm サブコマンド、フラグ、および SQLite データベース形式に関連する機能も非推奨となり、今後のリリースで削除されます。機能は引き続きサポートされており、非推奨の SQLite データベース形式を使用するカタログに使用する必要があります。

opm index prune などの SQLite データベース形式を使用する **opm** サブコマンドおよびフラグの多くは、ファイルベースのカタログ形式では機能しません。ファイルベースのカタログを使用する方法の詳細は、カスタムカタログの管理 および Operator Framework パッケージ形式 を参照してください。

2.6.1. Operator カタログについて

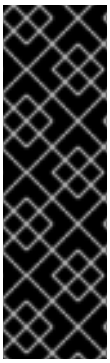
Operator カタログは、Operator Lifecycle Manager (OLM) がクエリーを行い、Operator およびそれらの依存関係をクラスターで検出し、インストールできるメタデータのリポジトリーです。OLM は最新バージョンのカタログから Operator を常にインストールします。

Operator Bundle Format に基づくインデックスイメージは、カタログのコンテナー化されたスナップショットです。これは、Operator マニフェストコンテンツのセットへのポインターのデータベースが

含まれるイミュータブルなアーティファクトです。カタログはインデックスイメージを参照し、クラスター上の OLM のコンテンツを調達できます。

カタログが更新されると、Operator の最新バージョンが変更され、それ以前のバージョンが削除または変更される可能性があります。また、制限されたネットワーク環境の OpenShift Dedicated クラスター上で OLM が実行されている場合、インターネットからカタログに直接アクセスして最新のコンテンツをプルすることはできません。

クラスター管理者は、Red Hat が提供するカタログをベースとして使用して、またはゼロから独自のカスタムインデックスイメージを作成できます。これを使用して、クラスターのカタログコンテンツを調達できます。独自のインデックスイメージの作成および更新により、クラスターで利用可能な Operator のセットをカスタマイズする方法が提供され、また前述のネットワークが制限された環境の問題を回避することができます。



重要

Kubernetes は定期的に特定の API を非推奨とし、後続のリリースで削除します。そのため、OpenShift Dedicated のバージョンで、API が削除された Kubernetes バージョンが採用されると、Operator がその API を使用できなくなります。

クラスターでカスタムカタログを使用している場合は、OpenShift Dedicated のバージョンと Operator の互換性の管理 を参照し、Operator 作成者がプロジェクトを更新する際にワークロードの問題や互換性のないアップグレードを避ける方法について詳細を確認してください。

注記

Operator のレガシー パッケージマニフェスト形式 (レガシー形式を使用していたカスタムカタログを含む) のサポートは、OpenShift Dedicated 4.8 以降で削除されました。

カスタムカタログイメージを作成する場合、OpenShift Dedicated 4 の以前のバージョンでは **oc adm category build** コマンドを使用する必要がありました。このコマンドはいくつかのリリースで非推奨となり、現在は削除されています。OpenShift Dedicated 4.6 以降、Red Hat 提供のインデックスイメージが利用可能になったため、カタログ作成者は **opm index** コマンドを使用してインデックスイメージを管理する必要があります。

関連情報

- カスタムカタログの管理
- パッケージ形式

2.6.2. Red Hat が提供する Operator カタログについて

Red Hat が提供するカタログソースは、デフォルトで **openshift-marketplace** namespace にインストールされます。これにより、すべての namespace でクラスター全体でカタログを利用できるようになります。

以下の Operator カタログは Red Hat によって提供されます。

カタログ	インデックスイメージ	説明

カタログ	インデックスイメージ	説明
redhat-operators	**registry.redhat.io/redhat/redhat-operator-index:v4**	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。
certified-operators	**registry.redhat.io/redhat/certified-operator-index:v4**	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。
redhat-marketplace	**registry.redhat.io/redhat/redhat-marketplace-index:v4**	Red Hat Marketplace から購入できる認定ソフトウェア。
community-operators	**registry.redhat.io/redhat/community-operator-index:v4**	redhat-openshift-ecosystem/community-operators-prod/operators GitHub リポジトリーで、関連する担当者によって保守されているソフトウェア。正式なサポートはありません。

クラスターのアップグレード時に、Red Hat が提供するデフォルトのカタログソースのインデックスイメージのタグは、Operator Lifecycle Manager (OLM) が最新版のカタログをプルするように、Cluster Version Operator (CVO) により自動更新されます。たとえば、OpenShift Dedicated 4.8 から 4.9 へのアップグレード時に、**redhat-operators** カタログの **CatalogSource** オブジェクトの **spec.image** フィールドは次のように更新されます。

```
registry.redhat.io/redhat/redhat-operator-index:v4.8
```

更新後は次のようになります。

```
registry.redhat.io/redhat/redhat-operator-index:v4.9
```

2.7. マルチテナントクラスター内の OPERATOR

Operator Lifecycle Manager (OLM) のデフォルトの動作は、Operator のインストール時に簡素化することを目的としています。ただし、この動作は、特にマルチテナントクラスターでは柔軟性に欠ける場合があります。OpenShift Dedicated クラスター上の複数のテナントが Operator を使用するには、OLM のデフォルトの動作では、管理者が Operator を **All namespaces** モードでインストールする必要がありますが、これは最小特権の原則に違反すると考えられます。

以下のシナリオを考慮して、環境と要件に最適な Operator インストールワークフローを決定してください。

関連情報

- Common terms: Multitenant
- マルチテナント Operator 管理の制限事項

2.7.1. デフォルトの Operator インストールモードと動作

管理者として Web コンソールを使用して Operator をインストールする場合、通常、Operator の機能に応じて、インストールモードに 2 つの選択肢があります。

単一の **namespace**

選択した単一の namespace に Operator をインストールし、Operator が要求するすべての権限をその namespace で使用できるようにします。

すべての **namespace**

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。Operator が要求するすべてのアクセス許可をすべての namespace で使用できるようにします。場合によっては、Operator の作成者はメタデータを定義して、その Operator が提案する namespace の 2 番目のオプションをユーザーに提供できます。

この選択は、影響を受ける namespace のユーザーが、namepsace でのロールに応じて、所有するカスタムリソース (CR) を活用できる Operators API にアクセスできることも意味します。

- **namespace-admin** および **namespace-edit** ロールは、Operator API の読み取り/書き込みが可能です。つまり、Operator API を使用できます。
- **namespace-view** ロールは、その Operator の CR オブジェクトを読み取ることができます。

Single namespace モードの場合、Operator 自体が選択したnamespaceにインストールされるため、その Pod とサービスアカウントもそこに配置されます。**All namespaces** モードの場合、Operator の権限はすべて自動的にクラスターロールに昇格されます。つまり、Operator はすべてのnamespaceでこれらの権限を持ちます。

関連情報

- Operator のクラスターへの追加
- Install modes types
- 推奨される namespace の設定

2.7.2. マルチテナントクラスターの推奨ソリューション

Multinamespace インストールモードは存在しますが、サポートされている Operator はほとんどありません。標準 **All namespaces** と **Single namespace** インストールモードの中間的なソリューションとして、次のワークフローを使用して、テナントごとに 1 つずつ、同じ Operator の複数のインスタンスをインストールできます。

1. テナントの namespace とは別のテナント Operator の namespace を作成します。これは、プロジェクトを作成することで実行できます。

2. テナントの namepsace のみを対象とするテナント Operator の Operator グループを作成します。

3. テナント Operator namespace に Operator をインストールします。

その結果、Operator はテナントの Operator namespace に存在し、テナントの namepsace を監視しますが、Operator の Pod もそのサービスアカウントも、テナントによって表示または使用できません。

このソリューションは、より優れたテナント分離、リソースの使用を犠牲にした最小特権の原則、および制約が確実に満たされるようにするための追加のオーケストレーションを提供します。詳細な手順については、マルチテナントクラスター用の Operator の複数インスタンスの準備を参照してください。

制限および考慮事項

このソリューションは、次の制約が満たされている場合にのみ機能します。

- 同じ Operator のすべてのインスタンスは、同じバージョンである必要があります。
- Operator は、他の Operator に依存することはできません。
- Operator は CRD 変換 Webhook を出荷できません。

重要

同じクラスターで同じ Operator の異なるバージョンを使用することはできません。最終的に、Operator の別のインスタンスのインストールは、以下の条件を満たす場合にブロックされます。

- インスタンスは Operator の最新バージョンではありません。
- インスタンスは、クラスターですでに使用されている新しいリビジョンに含まれる情報またはバージョンを欠いている CRD の古いリビジョンを出荷します。

関連情報

- マルチテナントクラスター用の Operator の複数インスタンスの準備

2.7.3. Operator のコロケーションと Operator グループ

Operator Lifecycle Manager (OLM) は、同じ namespace にインストールされている OLM 管理の Operator を処理します。つまり、それらの **Subscription** リソースは、関連する Operator として同じ namespaceに配置されます。それらが実際には関連していなくても、いずれかが更新されると、OLM はバージョンや更新ポリシーなどの状態を考慮します。

Operator のコロケーションと Operator グループの効果的な使用の詳細は、Operator Lifecycle Manager (OLM) → マルチテナント対応と Operator のコロケーション を参照してください。

2.8. CRD

2.8.1. カスタムリソース定義か定義定義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。

2.8.1.1. 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 **Pods** 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。

2.8.1.2. 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。

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されます。

   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. CRD からグループ名および API バージョン (name/version) 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
   ```

2.8.1.3. 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>
```

以下に例を示します。

```
$ oc get crontab
```

出力例

```
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。以下に例を示します。

```
$ 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
```

以下に例を示します。

```
$ oc get ct -o yaml
```

出力例

```
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 ユーザータスク

3.1. 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 Dedicated Web コンソールを使用して、インストールされた Operator からアプリケーションを作成する例を示します。

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

手順

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

   ヒント

   以下を使用して、CLI でこのリストを取得できます。

   ```
   $ oc get csv
   ```

3. **Installed Operators** ページで、etcd Operator をクリックして詳細情報および選択可能なアクションを表示します。
   Provided APIs に表示されているように、この Operator は 3 つの新規リソースタイプを利用可能にします。これには、**etcd** クラスター (**EtcdCluster** リソース) のタイプが含まれます。これらのオブジェクトは、**Deployment** または **ReplicaSet** などの組み込み済みのネイティブ Kubernetes オブジェクトと同様に機能しますが、これらには etcd を管理するための固有のロジックが含まれます。
4. 新規 etcd クラスターを作成します。
   a. **etcd Cluster** API ボックスで、**Create instance** をクリックします。
   b. 次のページでは、**EtcdCluster** オブジェクト (クラスターのサイズなど) のテンプレートを起動する最小条件を変更できます。ここでは **Create** をクリックして確定します。これにより、Operator がトリガーされ、Pod、サービス、および新規 etcd クラスターの他のコンポーネントが起動します。
5. **example** etcd クラスター、**Resources** タブの順にクリックし、Operator が自動的に作成および設定した多数のリソースが含まれていることを確認します。

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

6. 所定プロジェクトで **edit** ロールを持つすべてのユーザーは、クラウドサービスのようにセルフサービス方式でプロジェクトにすでに作成されている 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。最も重要なことは、適切なアクセス権を持つ dedicated-admin または開発者が、アプリケーションでデータベースを簡単に使用できるようになった点です。

第4章 管理者タスク

4.1. OPERATOR のクラスターへの追加

dedicated-admin ロールを持つ管理者は、Operator Lifecycle Manager (OLM) を使用して、OLM ベースの Operator を OpenShift Dedicated クラスターにインストールできます。

注記

OLM が同一 namespace に配置されたインストール済み Operator の更新を処理する方法や、カスタムグローバル Operator グループで Operator をインストールする別の方法は、マルチテナント対応と Operator のコロケーション を参照してください。

4.1.1. OperatorHub を使用した Operator のインストールについて

OperatorHub は Operator を検出するためのユーザーインターフェイスです。これは Operator Lifecycle Manager (OLM) と連携し、クラスター上で Operator をインストールし、管理します。

dedicated-admin は、OpenShift Dedicated 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。Operator を 1 つまたは複数の namespace にサブスクライブし、Operator を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、(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 をすべてのユーザーおよびプロジェクトで利用可能にします。

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、これをリストから選択します。

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選択されたチャネルで利用可能になると、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。

手動更新を選択した場合、新しいバージョンの Operator が利用可能になると、OLM によって更新要求が作成されます。その後、**dedicated-admin** として、その更新リクエストを手動で承認して、Operator を新しいバージョンに更新する必要があります。

関連情報

- OperatorHub について

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

OpenShift Dedicated Web コンソールを使用して、OperatorHub から Operator をインストールしてサブスクライブできます。

前提条件

- **dedicated-admin** ロールを持つアカウントを使用して、OpenShift Dedicated クラスターにアクセスできる。

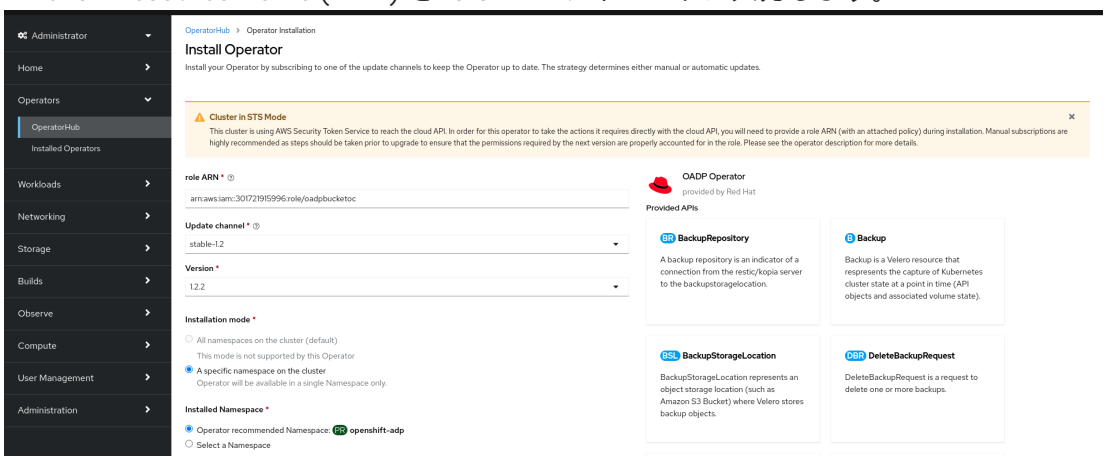
手順

1. Web コンソールで、**Operators → OperatorHub** ページに移動します。
2. スクロールするか、キーワードを **Filter by keyword** ボックスに入力し、必要な Operator を見つけます。たとえば、Advanced Cluster Management for Kubernetes Operator を検索するには **advanced** を入力します。
   また、インフラストラクチャー機能 でオプションをフィルターすることもできます。たとえば、非接続環境 (ネットワークが制限された環境ともしても知られる) で機能する Operator を表示するには、**Disconnected** を選択します。
3. Operator を選択して、追加情報を表示します。

注記

コミュニティー Operator を選択すると、Red Hat がコミュニティー Operator を認定していないことを警告します。続行する前に警告を確認する必要があります。

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. **Install Operator** ページで以下を行います。
   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. トークン認証が有効になっているクラウドプロバイダー上のクラスターの場合:
      - クラスターが AWS STS モードの場合は、サービスアカウントの AWS IAM ロールの Amazon Resource Name (ARN) を **role ARN** フィールドに入力します。



ロールの ARN を作成するには、AWS アカウントの準備 で説明されている手順に従います。

- クラスターが Azure AD Workload Identity モードの場合は、クライアント ID、テナント ID、およびサブスクリプション ID を適切なフィールドに追加します。

c. 複数の更新チャネルが利用可能な場合は、**Update channel** を選択します。

d. 前述のように、**Automatic** または **Manual** 承認ストラテジーを選択します。

重要

Web コンソールにクラスターが AWS STS または Azure AD Workload Identity モードであることが示された場合は、**Update approval** を **Manual** に設定する必要があります。

更新前に権限の変更が必要になる可能性があるため、自動更新承認のあるサブスクリプションは推奨できません。手動更新承認付きのサブスクリプションにより、管理者は新しいバージョンの権限を確認し、更新前に必要な手順を実行する機会が確保されます。

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

a. 手動 の承認ストラテジーを選択している場合、サブスクリプションのアップグレードステータスは、そのインストール計画を確認し、承認するまで **Upgrading** のままになります。
Install Plan ページでの承認後に、サブスクリプション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 の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。

7. 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、**Operators → Installed Operators** を選択し、インストールされた Operator のクラスターサービスバージョン (CSV) が表示されることを確認します。その **Status** は最終的に関連する namespace で **InstallSucceeded** に解決するはずです。

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、以下を実行します。

a. さらにトラブルシューティングを行うために問題を報告している **Workloads → Pods** ページで、**openshift-operators** プロジェクト (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他の関連の namespace) の Pod のログを確認します。

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

OpenShift Dedicated 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、**Subscription** オブジェクトを作成または更新します。

前提条件

- **dedicated-admin** ロールを持つアカウントを使用して、OpenShift Dedicated クラスターにアクセスできる。
- OpenShift CLI (**oc**) がインストールされている。

手順

1. OperatorHub からクラスターで利用できる Operator のリストを表示します。

```
$ oc get packagemanifests -n openshift-marketplace
```

出力例

```
NAME                              CATALOG               AGE
3scale-operator                   Red Hat Operators     91m
advanced-cluster-management       Red Hat Operators     91m
amq7-cert-manager                 Red Hat Operators     91m
...
couchbase-enterprise-certified    Certified Operators   91m
crunchy-postgres-operator         Certified Operators   91m
mongodb-enterprise                Certified Operators   91m
...
etcd                              Community Operators   91m
jaeger                            Community Operators   91m
kubefed                           Community Operators   91m
...
```

必要な Operator のカタログをメモします。

2. 必要な Operator を検査して、サポートされるインストールモードおよび利用可能なチャネルを確認します。

```
$ oc describe packagemanifests <operator_name> -n openshift-marketplace
```

3. **OperatorGroup** で定義される Operator グループは、Operator グループと同じ namespace 内のすべての Operator に必要な RBAC アクセスを生成するターゲット namespace を選択します。
Operator をサブスクライブする namespace には、Operator のインストールモードに一致する Operator グループが必要になります (**AllNamespaces** または **SingleNamespace** モードのいずれか)。インストールする Operator が **AllNamespaces** を使用する場合、 **openshift-operators** namespace には適切な Operator グループがすでに配置されます。

ただし、Operator が **SingleNamespace** モードを使用し、適切な Operator グループがない場合、それらを作成する必要があります。

注記

この手順の Web コンソールバージョンでは、**SingleNamespace** モードを選択する際に、**OperatorGroup** および **Subscription** オブジェクトの作成を背後で自動的に処理します。

a. **OperatorGroup** オブジェクト YAML ファイルを作成します (例: **operatorgroup.yaml**)。

OperatorGroup オブジェクトのサンプル

```
apiVersion: operators.coreos.com/v1
kind: OperatorGroup
metadata:
  name: <operatorgroup_name>
  namespace: <namespace>
spec:
  targetNamespaces:
  - <namespace>
```

b. **OperatorGroup** オブジェクトを作成します。

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

4. **Subscription** オブジェクトの YAML ファイルを作成し、namespace を Operator にサブスクライブします (例: **sub.yaml**)。

Subscription オブジェクトの例

```
apiVersion: operators.coreos.com/v1alpha1
kind: Subscription
metadata:
  name: <subscription_name>
  namespace: openshift-operators (1)
spec:
  channel: <channel_name> (2)
  name: <operator_name> (3)
  source: redhat-operators (4)
  sourceNamespace: openshift-marketplace (5)
  config:
    env: (6)
    - name: ARGS
      value: "-v=10"
    envFrom: (7)
    - secretRef:
        name: license-secret
    volumes: (8)
    - name: <volume_name>
      configMap:
        name: <configmap_name>
    volumeMounts: (9)
    - mountPath: <directory_name>
      name: <volume_name>
    tolerations: (10)
    - operator: "Exists"
    resources: (11)
      requests:
        memory: "64Mi"
        cpu: "250m"
      limits:
        memory: "128Mi"
```

```
      cpu: "500m"
  nodeSelector: (12)
    foo: bar
```

(1) デフォルトの **AllNamespaces** インストールモードの使用については、**openshift-operators** namespace を指定します。カスタムグローバル namespace を作成している場合はこれを指定できます。それ以外の場合は、**SingleNamespace** インストールモードの使用について関連する単一の namespace を指定します。

(2) サブスクライブするチャネルの名前。

(3) サブスクライブする Operator の名前。

(4) Operator を提供するカタログソースの名前。

(5) カタログソースの namespace。デフォルトの OperatorHub カタログソースには **openshift-marketplace** を使用します。

(6) **env** パラメーターは、OLM によって作成される Pod のすべてのコンテナーに存在する必要がある環境変数の一覧を定義します。

(7) **envFrom** パラメーターは、コンテナーの環境変数に反映するためのソースの一覧を定義します。

(8) **volumes** パラメーターは、OLM によって作成される Pod に存在する必要があるボリュームの一覧を定義します。

(9) **volumeMounts** パラメーターは、OLM によって作成される Pod のすべてのコンテナーに存在する必要があるボリュームマウントの一覧を定義します。**volumeMount** が存在しない ボリューム を参照する場合、OLM は Operator のデプロイに失敗します。

(10) **tolerations** パラメーターは、OLM によって作成される Pod の容認の一覧を定義します。

(11) **resources** パラメーターは、OLM によって作成される Pod のすべてのコンテナーのリソース制約を定義します。

(12) **nodeSelector** パラメーターは、OLM によって作成される Pod の ノードセレクター を定義します。

5. トークン認証が有効になっているクラウドプロバイダー上のクラスターの場合:

   a. **Subscription** オブジェクトが手動更新承認に設定されていることを確認します。

```
kind: Subscription
# ...
spec:
  installPlanApproval: Manual (1)
```

(1) 更新前に権限の変更が必要になる可能性があるため、自動更新承認のあるサブスクリプションは推奨できません。手動更新承認付きのサブスクリプションにより、管理者は新しいバージョンの権限を確認し、更新前に必要な手順を実行する機会が確保されます。

   b. 関連するクラウドプロバイダー固有のフィールドを **Subscription** オブジェクトの **config** セクションに含めます。

- クラスターが AWS STS モードの場合は、次のフィールドを含めます。

```
kind: Subscription
# ...
spec:
  config:
    env:
    - name: ROLEARN
      value: "<role_arn>" (1)
```

(1) ロール ARN の詳細を含めます。

- クラスターが Azure AD Workload Identity モードの場合は、次のフィールドを含めます。

```
kind: Subscription
# ...
spec:
 config:
   env:
   - name: CLIENTID
     value: "<client_id>" (1)
   - name: TENANTID
     value: "<tenant_id>" (2)
   - name: SUBSCRIPTIONID
     value: "<subscription_id>" (3)
```

(1) クライアント ID を含めます。

(2) テナント ID を含めます。

(3) サブスクリプション ID を含めます。

6. **Subscription** オブジェクトを作成します。

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

この時点で、OLM は選択した Operator を認識します。Operator のクラスターサービスバージョン (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。

関連情報

- Operator グループについて

4.1.4. Operator の特定バージョンのインストール

Subscription オブジェクトにクラスターサービスバージョン (CSV) を設定して Operator の特定バージョンをインストールできます。

前提条件

- **dedicated-admin** ロールを持つアカウントを使用して、OpenShift Dedicated クラスターにアクセスできる。
- OpenShift CLI (**oc**) がインストールされている。

手順

1. 次のコマンドを実行して、インストールする Operator の利用可能なバージョンとチャネルを検索します。

   コマンド構文

   ```
   $ oc describe packagemanifests <operator_name> -n <catalog_namespace>
   ```

   たとえば、次のコマンドは、OperatorHub から Red Hat Quay Operator の利用可能なチャネルとバージョンを出力します。

   コマンドの例

   ```
   $ oc describe packagemanifests quay-operator -n openshift-marketplace
   ```

   例**4.1** 出力例

   ```
   Name:         quay-operator
   Namespace:    operator-marketplace
   Labels:       catalog=redhat-operators
                 catalog-namespace=openshift-marketplace
                 hypershift.openshift.io/managed=true
                 operatorframework.io/arch.amd64=supported
                 operatorframework.io/os.linux=supported
                 provider=Red Hat
                 provider-url=
   Annotations:  <none>
   API Version:  packages.operators.coreos.com/v1
   Kind:         PackageManifest
   ...
     Current CSV:  quay-operator.v3.7.11
   ...
     Entries:
       Name:      quay-operator.v3.7.11
       Version:   3.7.11
       Name:      quay-operator.v3.7.10
       Version:   3.7.10
       Name:      quay-operator.v3.7.9
       Version:   3.7.9
       Name:      quay-operator.v3.7.8
       Version:   3.7.8
       Name:      quay-operator.v3.7.7
       Version:   3.7.7
       Name:      quay-operator.v3.7.6
       Version:   3.7.6
       Name:      quay-operator.v3.7.5
       Version:   3.7.5
       Name:      quay-operator.v3.7.4
   ```

```
      Version:    3.7.4
      Name:       quay-operator.v3.7.3
      Version:    3.7.3
      Name:       quay-operator.v3.7.2
      Version:    3.7.2
      Name:       quay-operator.v3.7.1
      Version:    3.7.1
      Name:       quay-operator.v3.7.0
      Version:    3.7.0
    Name:         stable-3.7
...
  Current CSV:  quay-operator.v3.8.5
...
  Entries:
    Name:         quay-operator.v3.8.5
    Version:      3.8.5
    Name:         quay-operator.v3.8.4
    Version:      3.8.4
    Name:         quay-operator.v3.8.3
    Version:      3.8.3
    Name:         quay-operator.v3.8.2
    Version:      3.8.2
    Name:         quay-operator.v3.8.1
    Version:      3.8.1
    Name:         quay-operator.v3.8.0
    Version:      3.8.0
  Name:           stable-3.8
 Default Channel:  stable-3.8
 Package Name:     quay-operator
```

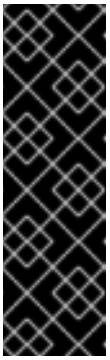
ヒント

次のコマンドを実行すると、Operator のバージョンとチャネル情報を YAML 形式で出力できます。

```
$ oc get packagemanifests <operator_name> -n <catalog_namespace> -o yaml
```

- namespace に複数のカタログがインストールされている場合は、次のコマンドを実行して、特定のカタログから Operator の使用可能なバージョンとチャネルを検索します。

```
$ oc get packagemanifest \
  --selector=catalog=<catalogsource_name> \
  --field-selector metadata.name=<operator_name> \
  -n <catalog_namespace> -o yaml
```



重要

Operator のカタログを指定しない場合、**oc get packagemanifest** および **oc description packagemanifest** コマンドを実行すると、次の条件が満たされると予期しないカタログからパッケージが返される可能性があります。

- 複数のカタログが同じ namespace にインストールされます。
- カタログには、同じ Operator、または同じ名前の Operator が含まれています。

2. **OperatorGroup** オブジェクトによって定義される Operator グループは、Operator グループと同じ namespace 内のすべての Operator に必要なロールベースのアクセス制御 (RBAC) アクセスを生成するターゲットの namespace を選択します。
Operator をサブスクライブする namespace には、Operator のインストールモードに一致する Operator グループが必要になります (**AllNamespaces** または **SingleNamespace** モードのいずれか)。インストールしようとしている Operator が **AllNamespaces** モードを使用する場合、**openshift-operators** namespace にはすでに適切な Operator グループが存在します。

   ただし、Operator が **SingleNamespace** モードを使用し、適切な Operator グループがない場合、それらを作成する必要があります。

   a. **OperatorGroup** オブジェクト YAML ファイルを作成します (例: **operatorgroup.yaml**)。

   OperatorGroup オブジェクトのサンプル

```
apiVersion: operators.coreos.com/v1
kind: OperatorGroup
metadata:
  name: <operatorgroup_name>
  namespace: <namespace>
spec:
  targetNamespaces:
  - <namespace>
```

   b. **OperatorGroup** オブジェクトを作成します。

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

3. **startingCSV** フィールドを設定し、特定バージョンの Operator に namespace をサブスクライブする **Subscription** オブジェクト YAML ファイルを作成します。**installPlanApproval** フィールドを **Manual** に設定し、Operator の新しいバージョンがカタログに存在する場合に Operator が自動的にアップグレードされないようにします。
たとえば、以下の **sub.yaml** ファイルを使用して、バージョン 3.7.10 に固有の Red Hat Quay Operator をインストールすることができます。

   最初にインストールする特定の Operator バージョンのあるサブスクリプション

```
apiVersion: operators.coreos.com/v1alpha1
kind: Subscription
metadata:
  name: quay-operator
  namespace: quay
spec:
  channel: quay-operator.v3.7.10
```

```
installPlanApproval: Manual (1)
name: quay-operator
source: redhat-operators
sourceNamespace: openshift-marketplace
startingCSV: quay-operator.v3.7.10 (2)
```

(1) 指定したバージョンがカタログの新しいバージョンに置き換えられる場合に備えて、承認ストラテジーを **Manual** に設定します。これにより、新しいバージョンへの自動アップグレードが阻止され、最初の CSV のインストールが完了する前に手動での承認が必要となります。

(2) Operator CSV の特定バージョンを設定します。

4. **Subscription** オブジェクトを作成します。

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

5. 保留中のインストール計画を手動で承認し、Operator のインストールを完了します。

関連情報

- 保留中の Operator 更新の手動による承認
- Installing global Operators in custom namespaces

4.1.5. Web コンソールでの Operator の特定のバージョンのインストール

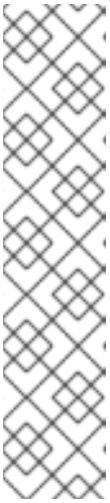
Web コンソールの OperatorHub を使用して、特定のバージョンの Operator をインストールできます。Operator のすべてのチャネルから Operator のさまざまなバージョンを参照し、そのチャネルとバージョンのメタデータを表示して、インストールする正確なバージョンを選択できます。

前提条件

- 管理者権限を持っている。

手順

1. Web コンソールで **Operators → OperatorHub** をクリックします。
2. インストールする Operator を選択します。
3. 選択した Operator から、リストから チャンネル と バージョン を選択できます。



注記

バージョン選択のデフォルトは、選択したチャネルの最新バージョンです。チャネルの最新バージョンが選択されている場合は、自動承認戦略がデフォルトで有効になります。それ以外の場合、選択したチャネルの最新バージョンをインストールしない場合は、手動による承認が必要です。

手動承認は、namespace にインストールされているすべての Operator に適用されます。

手動承認を使用して Operator をインストールすると、namespace 内にインストールされたすべての Operator が手動承認戦略で機能し、すべての Operator が一緒に更新されます。Operator を個別に更新するには、別の namespace にインストールします。

4. **Install** をクリックします。

検証

- Operator をインストールすると、メタデータに、インストールされているチャネルとバージョンが示されます。

注記

チャネルとバージョンのドロップダウンメニューは、このカタログコンテキストで他のバージョンのメタデータを表示するために引き続き使用できます。

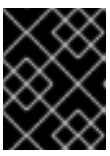
4.1.6. マルチテナントクラスター用の Operator の複数インスタンスの準備

dedicated-admin ロールを持つ管理者は、マルチテナントクラスターで使用する Operator のインスタンスを複数追加できます。これは、最小特権の原則に違反していると見なされる標準の **All namespaces** インストールモード、または広く採用されていない **Multinamespace** モードのいずれかを使用する代替ソリューションです。詳細は、「マルチテナントクラスター内の Operator」を参照してください。

次の手順では、テナントは、デプロイされた一連のワークロードに対する共通のアクセス権と特権を共有するユーザーまたはユーザーのグループです。テナント **Operator** は、そのテナントのみによる使用を意図した Operator のインスタンスです。

前提条件

- **dedicated-admin** ロールを持つユーザーとしてクラスターにアクセスできる。
- インストールする Operator のすべてのインスタンスは、特定のクラスター全体で同じバージョンである必要があります。



重要

この制限およびその他の制限の詳細は、「マルチテナントクラスター内の Operator」を参照してください。

手順

1. Operator をインストールする前に、テナントの namepsace とは別のテナント Operator の namespace を作成します。これは、プロジェクトを作成することで実行できます。たとえば、テナントの namespace が **team1** の場合、**team1-operator** プロジェクトを作成します。

```
$ oc new-project team1-operator
```

2. **spec.targetNamespaces** リストにその 1 つの namespace エントリーのみを使用して、テナントのnamespaceをスコープとするテナント Operator の Operator グループを作成します。

   a. **OperatorGroup** リソースを定義し、YAML ファイル (例: **team1-operatorgroup.yaml)** を保存します。

```
apiVersion: operators.coreos.com/v1
kind: OperatorGroup
metadata:
  name: team1-operatorgroup
  namespace: team1-operator
spec:
  targetNamespaces:
  - team1 1
```

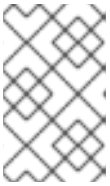
   1 **spec.targetNamespaces** リストでテナントのnamespaceのみを定義します。

   b. 以下のコマンドを実行して Operator グループを作成します。

```
$ oc create -f team1-operatorgroup.yaml
```

次のステップ

- テナント Operator namespace に Operator をインストールします。このタスクは、CLI の代わりに Web コンソールで OperatorHub を使用することにより、より簡単に実行できます。詳細な手順は、Web コンソールを使用した OperatorHub からのインストール を参照してください。



注記

Operator のインストールが完了すると、Operator はテナントの Operator namespaceに存在し、テナントのnamespaceを監視しますが、Operator の Pod もそのサービスアカウントも、テナントによって表示または使用されません。

関連情報

- マルチテナントクラスター内の Operator

4.1.7. Installing global Operators in custom namespaces

OpenShift Dedicated を使用して Operator をインストールする場合、デフォルトの動作では、**All namespaces** インストールモードをサポートする Operator がデフォルトの **openshift-operators** グローバル namespace にインストールされます。これにより、namespace内のすべての Operator 間で共有インストールプランと更新ポリシーに関連する問題が発生する可能性があります。これらの制限について、詳しくは「マルチテナント対応と Operator のコロケーション」を参照してください。

dedicated-admin ロールを持つ管理者は、カスタムのグローバル namespace を作成し、その namespace を使用して個別の Operator またはスコープ指定した Operator のセットとその依存関係をインストールすることで、このデフォルトの動作を手動で回避できます。

前提条件

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

手順

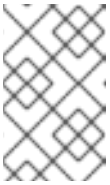
1. Operator をインストールする前に、目的の Operator をインストールするための namespace を作成します。これは、プロジェクトを作成することで実行できます。このプロジェクトの namespace は、カスタムのグローバル namespace になります。

```
$ oc new-project global-operators
```

2. すべてのnamespaceを監視する Operator グループである、カスタム **global Operator group** を作成します。

   a. **OperatorGroup** リソースを定義し、**global-operatorgroup.yaml** などの YAML ファイルを保存します。**spec.selector** フィールドと **spec.targetNamespaces** フィールドの両方を省略して、すべてのnamespaceを選択する **global Operator group** にします。

```
apiVersion: operators.coreos.com/v1
kind: OperatorGroup
metadata:
  name: global-operatorgroup
  namespace: global-operators
```



注記

作成されたグローバル Operator グループの **status.namespaces** には、空の文字列 (**""**) が含まれています。これは、すべてのnamespaceを監視する必要があることを消費する Operator に通知します。

   b. 以下のコマンドを実行して Operator グループを作成します。

```
$ oc create -f global-operatorgroup.yaml
```

次のステップ

- 必要な Operator をカスタムグローバル namespace にインストールします。Web コンソールは、Operator のインストール時にカスタムグローバル namespace で **Installed Namespace** メニューを設定しないため、このタスクは OpenShift CLI (**oc**) でのみ実行できます。詳細な手順は、CLI を使用した OperatorHub からのインストール を参照してください。

注記

Operator のインストールを開始すると、Operator に依存関係がある場合、その依存関係もカスタムグローバルnamespaceに自動的にインストールされます。その結果、依存関係 Operator が同じ更新ポリシーと共有インストールプランを持つことが有効になります。

関連情報

- マルチテナント対応と Operator のコロケーション

4.1.8. Operator ワークロードの Pod の配置

デフォルトで、Operator Lifecycle Manager (OLM) は、Operator のインストールまたはオペランドのワークロードのデプロイ時に Pod を任意のワーカーノードに配置します。管理者は、ノードセレクター、テイント、および容認 (Toleration) の組み合わせを持つプロジェクトを使用して、Operator およびオペランドの特定のノードへの配置を制御できます。

Operator およびオペランドワークロードの Pod 配置の制御には以下の前提条件があります。

1. 要件に応じて Pod のターゲットとするノードまたはノードのセットを判別します。利用可能な場合は、単数または複数のノードを特定する **node-role.kubernetes.io/app** などの既存ラベルをメモします。それ以外の場合は、コンピュートマシンセットを使用するか、ノードを直接編集して、**myoperator** などのラベルを追加します。このラベルは、後のステップでプロジェクトのノードセレクターとして使用します。

2. 関連しないワークロードを他のノードに向けつつ、特定のラベルの付いた Pod のみがノードで実行されるようにする必要がある場合、コンピュートマシンセットを使用するか、ノードを直接編集してテイントをノードに追加します。テイントに一致しない新規 Pod がノードにスケジュールされないようにする effect を使用します。たとえば、**myoperator:NoSchedule** テイントは、テイントに一致しない新規 Pod がノードにスケジュールされないようにしますが、ノードの既存 Pod はそのまま残ります。

3. デフォルトのノードセレクターで設定され、テイントを追加している場合に一致する容認を持つプロジェクトを作成します。

この時点で、作成したプロジェクトでは、以下のシナリオの場合に指定されたノードに Pod を導くことができます。

Operator Pod の場合

管理者は、次のセクションで説明するように、プロジェクトに **Subscription** オブジェクトを作成できます。その結果、Operator Pod は指定されたノードに配置されます。

オペランド **Pod** の場合

インストールされた Operator を使用して、ユーザーはプロジェクトにアプリケーションを作成できます。これにより、Operator が所有するカスタムリソース (CR) がプロジェクトに置かれます。その結果、Operator が他の namespace にクラスター全体のオブジェクトまたはリソースをデプロイしない限り、オペランド Pod は指定されたノードに配置されます。この場合、このカスタマイズされた Pod の配置は適用されません。

関連情報

- ノードに手動で、または コンピュートマシンセットを使用 してテイントと容認を追加する
- プロジェクトスコープのノードセレクターの作成
- ノードセレクターおよび容認を使用したプロジェクトの作成

4.1.9. Operator のインストール場所の制御

デフォルトでは、Operator をインストールすると、OpenShift Dedicated は Operator Pod をワーカーノードの 1 つにランダムにインストールします。ただし、特定のノードまたはノードのセットでその Pod をスケジュールする必要がある場合があります。

以下の例では、Operator Pod を特定のノードまたはノードのセットにスケジュールする状況について説明します。

- 同じホストまたは同じラックに配置されたホストでスケジュールされた一緒に動作する Operator が必要な場合
- ネットワークまたはハードウェアの問題によるダウンタイムを回避するために、Operator をインフラストラクチャー全体に分散させたい場合

Operator の **Subscription** オブジェクトにノードアフィニティー、Pod アフィニティー、または Pod 非アフィニティー制約を追加することで、Operator Pod がインストールされる場所を制御できます。ノードアフィニティーは、Pod の配置場所を判別するためにスケジューラーによって使用されるルールのセットです。Pod アフィニティーを使用すると、関連する Pod が同じノードにスケジュールされていることを確認できます。Pod 非アフィニティーを使用すると、ノードで Pod がスケジュールされないようにすることができます。

次の例は、ノードアフィニティーまたは Pod 非アフィニティーを使用して、Custom Metrics Autoscaler Operator のインスタンスをクラスター内の特定のノードにインストールする方法を示しています。

Operator Pod を特定のノードに配置するノードアフィニティーの例

```
apiVersion: operators.coreos.com/v1alpha1
kind: Subscription
metadata:
  name: openshift-custom-metrics-autoscaler-operator
  namespace: openshift-keda
spec:
  name: my-package
  source: my-operators
  sourceNamespace: operator-registries
  config:
    affinity:
      nodeAffinity: (1)
        requiredDuringSchedulingIgnoredDuringExecution:
          nodeSelectorTerms:
          - matchExpressions:
            - key: kubernetes.io/hostname
              operator: In
              values:
              - ip-10-0-163-94.us-west-2.compute.internal
#...
```

(1) Operator の Pod を **ip-10-0-163-94.us-west-2.compute.internal** という名前のノードでスケジュールする必要があるノードアフィニティー。

Operator Pod を特定のプラットフォームのノードに配置するノードアフィニティーの例

```
apiVersion: operators.coreos.com/v1alpha1
kind: Subscription
metadata:
  name: openshift-custom-metrics-autoscaler-operator
  namespace: openshift-keda
spec:
```

```
  name: my-package
  source: my-operators
  sourceNamespace: operator-registries
  config:
    affinity:
      nodeAffinity: (1)
        requiredDuringSchedulingIgnoredDuringExecution:
          nodeSelectorTerms:
          - matchExpressions:
            - key: kubernetes.io/arch
              operator: In
              values:
              - arm64
            - key: kubernetes.io/os
              operator: In
              values:
              - linux
#...
```

(1) Operator の Pod を **kubernetes.io/arch=arm64** および **kubernetes.io/os=linux** ラベルを持つノードでスケジュールする必要があるノードアフィニティー。

Operator Pod を 1 つ以上の特定のノードに配置する Pod アフィニティーの例

```
apiVersion: operators.coreos.com/v1alpha1
kind: Subscription
metadata:
  name: openshift-custom-metrics-autoscaler-operator
  namespace: openshift-keda
spec:
  name: my-package
  source: my-operators
  sourceNamespace: operator-registries
  config:
    affinity:
      podAffinity: (1)
        requiredDuringSchedulingIgnoredDuringExecution:
        - labelSelector:
            matchExpressions:
            - key: app
              operator: In
              values:
              - test
          topologyKey: kubernetes.io/hostname
#...
```

(1) **app=test** ラベルを持つ Pod を持つノードに Operator の Pod を配置する Pod アフィニティー。

Operator Pod が 1 つ以上の特定のノードからアクセスできないようにする Pod 非アフィニティーの例

```
apiVersion: operators.coreos.com/v1alpha1
kind: Subscription
```

```
metadata:
  name: openshift-custom-metrics-autoscaler-operator
  namespace: openshift-keda
spec:
  name: my-package
  source: my-operators
  sourceNamespace: operator-registries
  config:
    affinity:
      podAntiAffinity: (1)
        requiredDuringSchedulingIgnoredDuringExecution:
        - labelSelector:
            matchExpressions:
            - key: cpu
              operator: In
              values:
              - high
          topologyKey: kubernetes.io/hostname
#...
```

(1) Operator の Pod が **cpu=high** ラベルの Pod を持つノードでスケジュールされないようにする Pod 非アフィニティー。

手順

Operator Pod の配置を制御するには、次の手順を実行します。

1. 通常どおり Operator をインストールします。
2. 必要に応じて、ノードがアフィニティーに適切に応答するようにラベル付けされていることを確認してください。
3. Operator **Subscription** オブジェクトを編集してアフィニティーを追加します。

```
apiVersion: operators.coreos.com/v1alpha1
kind: Subscription
metadata:
  name: openshift-custom-metrics-autoscaler-operator
  namespace: openshift-keda
spec:
  name: my-package
  source: my-operators
  sourceNamespace: operator-registries
  config:
    affinity: (1)
      nodeAffinity:
        requiredDuringSchedulingIgnoredDuringExecution:
          nodeSelectorTerms:
          - matchExpressions:
            - key: kubernetes.io/hostname
              operator: In
              values:
              - ip-10-0-185-229.ec2.internal
#...
```

1 **nodeAffinity**、**podAffinity**、または **podAntiAffinity** を追加します。アフィニティーの作成については、以下のその他のリソースセクションを参照してください。

検証

- Pod が特定のノードにデプロイされていることを確認するには、次のコマンドを実行します。

```
$ oc get pods -o wide
```

出力例

```
NAME                                          READY   STATUS    RESTARTS   AGE   IP
NODE                          NOMINATED NODE   READINESS GATES
custom-metrics-autoscaler-operator-5dcc45d656-bhshg   1/1     Running   0          50s
10.131.0.20   ip-10-0-185-229.ec2.internal   <none>           <none>
```

関連情報

- Pod のアフィニティーについて
- ノードアフィニティーについて

4.2. インストール済み OPERATOR の更新

dedicated-admin ロールを持つ管理者は、Operator Lifecycle Manager (OLM) を使用して OpenShift Dedicated クラスターに以前インストールした Operator を更新できます。

注記

OLM が同一 namespace に配置されたインストール済み Operator の更新を処理する方法や、カスタムグローバル Operator グループで Operator をインストールする別の方法は、マルチテナント対応と Operator のコロケーション を参照してください。

4.2.1. Operator 更新の準備

インストールされた Operator のサブスクリプションは、Operator の更新を追跡および受信する更新チャネルを指定します。更新チャネルを変更して、新しいチャネルからの更新の追跡と受信を開始できます。

サブスクリプションの更新チャネルの名前は Operator 間で異なる可能性がありますが、命名スキーム通常、特定の Operator 内の共通の規則に従います。たとえば、チャネル名は Operator によって提供されるアプリケーションのマイナーリリース更新ストリーム (**1.2**、**1.3**) またはリリース頻度 (**stable**、**fast**) に基づく可能性があります。

注記

インストールされた Operator は、現在のチャネルよりも古いチャネルに切り換えることはできません。

Red Hat Customer Portal Labs には、管理者が Operator の更新を準備するのに役立つ以下のアプリケーションが含まれています。

- Red Hat OpenShift Container Platform Operator Update Information Checker

このアプリケーションを使用すると、OpenShift Dedicated のさまざまなバージョンを対象に、Operator Lifecycle Manager ベースの Operator を検索し、更新チャネルごとに利用可能な Operator のバージョンを確認できます。Cluster Version Operator ベースの Operator は含まれません。

4.2.2. Operator の更新チャネルの変更

OpenShift Dedicated を使用して、Operator の更新チャネルを変更できます。

ヒント

サブスクリプションの承認ストラテジーが **Automatic** に設定されている場合、アップグレードプロセスは、選択したチャネルで新規 Operator バージョンが利用可能になるとすぐに開始します。承認ストラテジーが **Manual** に設定されている場合は、保留中のアップグレードを手動で承認する必要があります。

前提条件

- Operator Lifecycle Manager (OLM) を使用して以前にインストールされている Operator。

手順

1. Web コンソールの **Administrator** パースペクティブで、**Operators → Installed Operators** に移動します。
2. 更新チャネルを変更する Operator の名前をクリックします。
3. **Subscription** タブをクリックします。
4. **Update channel** の下にある更新チャネルの名前をクリックします。
5. 変更する新しい更新チャネルをクリックし、**Save** をクリックします。
6. **Automatic** 承認ストラテジーのあるサブスクリプションの場合、更新は自動的に開始します。**Operators → Installed Operators** ページに戻り、更新の進捗をモニターします。完了時に、ステータスは **Succeeded** および **Up to date** に変更されます。
   Manual 承認ストラテジーのあるサブスクリプションの場合、**Subscription** タブから更新を手動で承認できます。

4.2.3. 保留中の Operator 更新の手動による承認

インストールされた Operator のサブスクリプションの承認ストラテジーが **Manual** に設定されている場合、新規の更新が現在の更新チャネルにリリースされると、インストールを開始する前に更新を手動で承認する必要があります。

前提条件

- Operator Lifecycle Manager (OLM) を使用して以前にインストールされている Operator。

手順

1. OpenShift Dedicated Web コンソールの **Administrator** パースペクティブで、**Operators → Installed Operators** に移動します。

2. 更新が保留中の Operator は **Upgrade available** のステータスを表示します。更新する Operator の名前をクリックします。

3. **Subscription** タブをクリックします。承認が必要な更新は、**Upgrade status** の横に表示されます。たとえば、**1 requires approval** が表示される可能性があります。

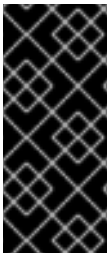
4. **1 requires approval** をクリックしてから、**Preview Install Plan** をクリックします。

5. 更新に利用可能なリソースとして一覧表示されているリソースを確認します。問題がなければ、**Approve** をクリックします。

6. **Operators → Installed Operators** ページに戻り、更新の進捗をモニターします。完了時に、ステータスは **Succeeded** および **Up to date** に変更されます。

4.3. クラスターからの OPERATOR の削除

以下では、Operator Lifecycle Manager (OLM) を使用して OpenShift Dedicated クラスターに以前インストールした Operator を削除またはアンインストールする方法について説明します。



重要

同じ Operator の再インストールを試行する前に、Operator を正常かつ完全にアンインストールする必要があります。Operator を適切かつ完全にアンインストールできていない場合、プロジェクトや namespace などのリソースが "Terminating" ステータスでスタックし、Operator を再インストールしようとすると "error resolving resource" メッセージが表示される可能性があります。

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

クラスター管理者は Web コンソールを使用して、選択した namespace からインストールされた Operator を削除できます。

前提条件

- **dedicated-admin** パーミッションを持つアカウントを使用して OpenShift Dedicated クラスター Web コンソールにアクセスできる。

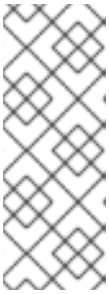
手順

1. **Operators → Installed Operators** ページに移動します。

2. スクロールするか、キーワードを **Filter by name** フィールドに入力して、削除する Operator を見つけます。次に、それをクリックします。

3. **Operator Details** ページの右側で、**Actions** 一覧から **Uninstall Operator** を選択します。**Uninstall Operator?** ダイアログボックスが表示されます。

4. **Uninstall** を選択し、Operator、Operator デプロイメント、および Pod を削除します。このアクションの後には、Operator は実行を停止し、更新を受信しなくなります。



注記

このアクションは、カスタムリソース定義 (CRD) およびカスタムリソース (CR) など、Operator が管理するリソースは削除されません。Web コンソールおよび継続して実行されるクラスター外のリソースによって有効にされるダッシュボードおよびナビゲーションアイテムには、手動でのクリーンアップが必要になる場合があります。Operator のアンインストール後にこれらを削除するには、Operator CRD を手動で削除する必要があります。

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

クラスター管理者は CLI を使用して、選択した namespace からインストールされた Operator を削除できます。

前提条件

- **dedicated-admin** パーミッションを持つアカウントを使用して OpenShift Dedicated クラスターにアクセスできる。
- OpenShift CLI (**oc**) がワークステーションにインストールされている。

手順

1. サブスクライブした Operator の最新バージョン (**serverless-operator** など) が、**currentCSV** フィールドで識別されていることを確認します。

```
$ oc get subscription.operators.coreos.com serverless-operator -n openshift-serverless -o yaml | grep currentCSV
```

出力例

```
  currentCSV: serverless-operator.v1.28.0
```

2. サブスクリプション (**serverless-operator** など) を削除します。

```
$ oc delete subscription.operators.coreos.com serverless-operator -n openshift-serverless
```

出力例

```
subscription.operators.coreos.com "serverless-operator" 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 serverless-operator.v1.28.0 -n openshift-serverless
```

出力例

```
clusterserviceversion.operators.coreos.com "serverless-operator.v1.28.0" deleted
```

4.3.3. 障害のあるサブスクリプションの更新

Operator Lifecycle Manager (OLM) で、ネットワークでアクセスできないイメージを参照する Operator をサブスクライブする場合、以下のエラーを出して失敗した **openshift-marketplace** namespace でジョブを見つけることができます。

出力例

```
ImagePullBackOff for
Back-off pulling image "example.com/openshift4/ose-elasticsearch-operator-
bundle@sha256:6d2587129c846ec28d384540322b40b05833e7e00b25cca584e004af9a1d292e"
```

出力例

```
rpc error: code = Unknown desc = error pinging docker registry example.com: Get
"https://example.com/v2/": dial tcp: lookup example.com on 10.0.0.1:53: no such host
```

その結果、サブスクリプションはこの障害のある状態のままとなり、Operator はインストールまたはアップグレードを実行できません。

サブスクリプション、クラスターサービスバージョン (CSV) その他の関連オブジェクトを削除して、障害のあるサブスクリプションを更新できます。サブスクリプションを再作成した後に、OLM は Operator の正しいバージョンを再インストールします。

前提条件

- アクセス不可能なバンドルイメージをプルできない障害のあるサブスクリプションがある。
- 正しいバンドルイメージにアクセスできることを確認している。

手順

1. Operator がインストールされている namespace から **Subscription** および **ClusterServiceVersion** オブジェクトの名前を取得します。

   ```
   $ oc get sub,csv -n <namespace>
   ```

   出力例

   ```
   NAME                                         PACKAGE                  SOURCE            CHANNEL
   subscription.operators.coreos.com/elasticsearch-operator   elasticsearch-operator   redhat-
   operators   5.0

   NAME                                                  DISPLAY                  VERSION
   REPLACES   PHASE
   clusterserviceversion.operators.coreos.com/elasticsearch-operator.5.0.0-65   OpenShift
   Elasticsearch Operator   5.0.0-65              Succeeded
   ```

2. サブスクリプションを削除します。

   ```
   $ oc delete subscription <subscription_name> -n <namespace>
   ```

3. クラスターサービスバージョンを削除します。

   ```
   $ oc delete csv <csv_name> -n <namespace>
   ```

4. **openshift-marketplace** namespace の失敗したジョブおよび関連する設定マップの名前を取得します。

```
$ oc get job,configmap -n openshift-marketplace
```

出力例

```
NAME                                                                COMPLETIONS   DURATION   AGE
job.batch/1de9443b6324e629ddf31fed0a853a121275806170e34c926d69e53a7fcbccb   1/1
26s        9m30s

NAME                                                                DATA   AGE
configmap/1de9443b6324e629ddf31fed0a853a121275806170e34c926d69e53a7fcbccb   3
9m30s
```

5. ジョブを削除します。

```
$ oc delete job <job_name> -n openshift-marketplace
```

これにより、アクセスできないイメージのプルを試行する Pod は再作成されなくなります。

6. 設定マップを削除します。

```
$ oc delete configmap <configmap_name> -n openshift-marketplace
```

7. Web コンソールの OperatorHub を使用した Operator の再インストール

検証

- Operator が正常に再インストールされていることを確認します。

```
$ oc get sub,csv,installplan -n <namespace>
```

4.4. OPERATOR LIFECYCLE MANAGER でのプロキシーサポートの設定

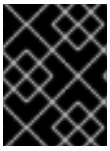
グローバルプロキシーが OpenShift Dedicated クラスターに設定されている場合、Operator Lifecycle Manager (OLM) は、クラスター全体のプロキシーで管理する Operator を自動的に設定します。ただし、インストールされた Operator をグローバルプロキシーを上書きするか、カスタム CA 証明書を挿入するように設定することもできます。

関連情報

- クラスター全体のプロキシーの設定
- Go、Ansible、および Helm のプロキシー設定をサポートする Operator の開発

4.4.1. Operator のプロキシー設定の上書き

クラスター全体の egress プロキシーが設定されている場合、Operator Lifecycle Manager (OLM) を使用して実行する Operator は、デプロイメントでクラスター全体のプロキシー設定を継承します。**dedicated-admin** ロールを持つ管理者は、Operator のサブスクリプションを設定することで、これらのプロキシー設定をオーバーライドすることもできます。



重要

Operator は、マネージドオペランドの Pod でのプロキシー設定の環境変数の設定を処理する必要があります。

前提条件

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

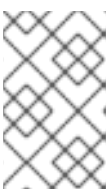
手順

1. Web コンソールで、**Operators → OperatorHub** ページに移動します。
2. Operator を選択し、**Install** をクリックします。
3. **Install Operator** ページで、**Subscription** オブジェクトを変更して以下の 1 つ以上の環境変数を **spec** セクションに組み込みます。
   - **HTTP_PROXY**
   - **HTTPS_PROXY**
   - **NO_PROXY**

   以下に例を示します。

   プロキシー設定の上書きのある Subscription オブジェクト

```
apiVersion: operators.coreos.com/v1alpha1
kind: Subscription
metadata:
  name: etcd-config-test
  namespace: openshift-operators
spec:
  config:
    env:
    - name: HTTP_PROXY
      value: test_http
    - name: HTTPS_PROXY
      value: test_https
    - name: NO_PROXY
      value: test
  channel: clusterwide-alpha
  installPlanApproval: Automatic
  name: etcd
  source: community-operators
  sourceNamespace: openshift-marketplace
  startingCSV: etcdoperator.v0.9.4-clusterwide
```



注記

これらの環境変数については、以前に設定されたクラスター全体またはカスタムプロキシーの設定を削除するために空の値を使用してそれらの設定を解除することもできます。

OLM はこれらの環境変数を単位として処理します。それらの環境変数が 1 つ以上設定されている場合、それらはすべて上書きされているものと見なされ、クラスター全体のデフォルト値はサブスクライブされた Operator のデプロイメントには使用されません。

4. **Install** をクリックし、Operator を選択された namespace で利用可能にします。

5. Operator の CSV が関連する namespace に表示されると、カスタムプロキシーの環境変数がデプロイメントに設定されていることを確認できます。たとえば、CLI を使用します。

```
$ oc get deployment -n openshift-operators \
    etcd-operator -o yaml \
    | grep -i "PROXY" -A 2
```

出力例

```
        - name: HTTP_PROXY
          value: test_http
        - name: HTTPS_PROXY
          value: test_https
        - name: NO_PROXY
          value: test
        image: quay.io/coreos/etcd-
operator@sha256:66a37fd61a06a43969854ee6d3e21088a98b93838e284a6086b13917f96b0
d9c
...
```

4.4.2. カスタム CA 証明書の挿入

dedicated-admin ロールを持つ管理者が config map を使用してカスタム CA 証明書をクラスターに追加すると、Cluster Network Operator がユーザー提供の証明書とシステム CA 証明書を 1 つのバンドルにマージします。このマージされたバンドルを Operator Lifecycle Manager (OLM) で実行されている Operator に挿入することができます。これは、man-in-the-middle HTTPS プロキシーがある場合に役立ちます。

前提条件

- **dedicated-admin** ロールを持つユーザーとして OpenShift Dedicated クラスターにアクセスできる。
- 設定マップを使用してクラスターに追加されたカスタム CA 証明書。
- 必要な Operator が OLM にインストールされ、実行される。

手順

1. Operator のサブスクリプションがある namespace に空の設定マップを作成し、以下のラベルを組み込みます。

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: trusted-ca 1
  labels:
    config.openshift.io/inject-trusted-cabundle: "true" 2
```

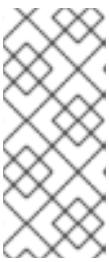
1. 設定マップの名前。
2. Cluster Network Operator に対してマージされたバンドルを挿入するように要求します。

この設定マップの作成後すぐに、設定マップにはマージされたバンドルの証明書の内容が設定されます。

2. **Subscription** オブジェクトを更新し、**trusted-ca** 設定マップをカスタム CA を必要とする Pod 内の各コンテナーにボリュームとしてマウントする **spec.config** セクションを追加します。

```
apiVersion: operators.coreos.com/v1alpha1
kind: Subscription
metadata:
  name: my-operator
spec:
  package: etcd
  channel: alpha
  config: 1
    selector:
      matchLabels:
        <labels_for_pods> 2
    volumes: 3
    - name: trusted-ca
      configMap:
        name: trusted-ca
        items:
          - key: ca-bundle.crt 4
            path: tls-ca-bundle.pem 5
    volumeMounts: 6
    - name: trusted-ca
      mountPath: /etc/pki/ca-trust/extracted/pem
      readOnly: true
```

1. **config** セクションがない場合に、これを追加します。
2. Operator が所有する Pod に一致するラベルを指定します。
3. **trusted-ca** ボリュームを作成します。
4. **ca-bundle.crt** は設定マップキーとして必要になります。
5. **tls-ca-bundle.pem** は設定マップパスとして必要になります。
6. **trusted-ca** ボリュームマウントを作成します。



注記

Operator のデプロイメントは認証局の検証に失敗し、**x509 certificate signed by unknown authority** エラーが表示される可能性があります。このエラーは、Operator のサブスクリプションの使用時にカスタム CA を挿入した後でも発生する可能性があります。この場合、Operator のサブスクリプションを使用して、trusted-ca の **mountPath** を **/etc/ssl/certs** として設定できます。

4.5. OPERATOR ステータスの表示

Operator Lifecycle Manager (OLM) のシステムの状態を理解することは、インストールされた Operator についての問題について意思決定を行い、デバッグを行う上で重要です。OLM は、サブスクリプションおよびそれに関連するカタログソースリソースの状態および実行されたアクションに関する知見を提供します。これは、それぞれの Operator の正常性を把握するのに役立ちます。

4.5.1. Operator サブスクリプションの状態のタイプ

サブスクリプションは状態についての以下のタイプを報告します。

表4.1 サブスクリプションの状態のタイプ

状態	説明
CatalogSourcesUnhealthy	解決に使用される一部のまたはすべてのカタログソースは正常ではありません。
InstallPlanMissing	サブスクリプションのインストール計画がありません。
InstallPlanPending	サブスクリプションのインストール計画はインストールの保留中です。
InstallPlanFailed	サブスクリプションのインストール計画が失敗しました。
ResolutionFailed	サブスクリプションの依存関係の解決に失敗しました。

注記

デフォルトの OpenShift Dedicated クラスター Operator は、Cluster Version Operator (CVO) によって管理されます。この Operator には **Subscription** オブジェクトがありません。アプリケーション Operator は、Operator Lifecycle Manager (OLM) によって管理されます。この Operator には **Subscription** オブジェクトがあります。

関連情報

- 障害のあるサブスクリプションの更新

4.5.2. CLI を使用した Operator サブスクリプションステータスの表示

CLI を使用して Operator サブスクリプションステータスを表示できます。

前提条件

- **dedicated-admin** ロールを持つユーザーとしてクラスターにアクセスできる。
- OpenShift CLI (**oc**) がインストールされている。

手順

1. Operator サブスクリプションをリスト表示します。

```
$ oc get subs -n <operator_namespace>
```

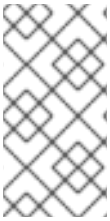
2. **oc describe** コマンドを使用して、**Subscription** リソースを検査します。

```
$ oc describe sub <subscription_name> -n <operator_namespace>
```

3. コマンド出力で、**Conditions** セクションで Operator サブスクリプションの状態タイプのステータスを確認します。以下の例では、利用可能なすべてのカタログソースが正常であるため、**CatalogSourcesUnhealthy** 状態タイプのステータスは **false** になります。

出力例

```
Name:         cluster-logging
Namespace:    openshift-logging
Labels:       operators.coreos.com/cluster-logging.openshift-logging=
Annotations:  <none>
API Version:  operators.coreos.com/v1alpha1
Kind:         Subscription
# ...
Conditions:
  Last Transition Time:  2019-07-29T13:42:57Z
  Message:               all available catalogsources are healthy
  Reason:                AllCatalogSourcesHealthy
  Status:                False
  Type:                  CatalogSourcesUnhealthy
# ...
```



注記

デフォルトの OpenShift Dedicated クラスター Operator は、Cluster Version Operator (CVO) によって管理されます。この Operator には **Subscription** オブジェクトがありません。アプリケーション Operator は、Operator Lifecycle Manager (OLM) によって管理されます。この Operator には **Subscription** オブジェクトがあります。

4.5.3. CLI を使用した Operator カタログソースのステータス表示

Operator カタログソースのステータスは、CLI を使用して確認できます。

前提条件

- **dedicated-admin** ロールを持つユーザーとしてクラスターにアクセスできる。
- OpenShift CLI (**oc**) がインストールされている。

手順

1. namespace のカタログソースをリスト表示します。例えば、クラスター全体のカタログソースに使用されている **openshift-marketplace**namespace を確認することができます。

```
$ oc get catalogsources -n openshift-marketplace
```

出力例

```
NAME              DISPLAY           TYPE   PUBLISHER   AGE
```

```
certified-operators   Certified Operators   grpc   Red Hat     55m
community-operators   Community Operators   grpc   Red Hat     55m
example-catalog       Example Catalog       grpc   Example Org 2m25s
redhat-marketplace    Red Hat Marketplace   grpc   Red Hat     55m
redhat-operators      Red Hat Operators     grpc   Red Hat     55m
```

2. カタログソースの詳細やステータスを確認するには、**oc describe** コマンドを使用します。

```
$ oc describe catalogsource example-catalog -n openshift-marketplace
```

出力例

```
Name:         example-catalog
Namespace:    openshift-marketplace
Labels:       <none>
Annotations:  operatorframework.io/managed-by: marketplace-operator
              target.workload.openshift.io/management: {"effect": "PreferredDuringScheduling"}
API Version:  operators.coreos.com/v1alpha1
Kind:         CatalogSource
# ...
Status:
  Connection State:
    Address:              example-catalog.openshift-marketplace.svc:50051
    Last Connect:         2021-09-09T17:07:35Z
    Last Observed State:  TRANSIENT_FAILURE
  Registry Service:
    Created At:         2021-09-09T17:05:45Z
    Port:               50051
    Protocol:           grpc
    Service Name:       example-catalog
    Service Namespace:  openshift-marketplace
# ...
```

前述の出力例では、最後に観測された状態が **TRANSIENT_FAILURE** となっています。この状態は、カタログソースの接続確立に問題があることを示しています。

3. カタログソースが作成された namespace の Pod をリストアップします。

```
$ oc get pods -n openshift-marketplace
```

出力例

```
NAME                                  READY   STATUS             RESTARTS   AGE
certified-operators-cv9nn             1/1     Running            0          36m
community-operators-6v8lp             1/1     Running            0          36m
marketplace-operator-86bfc75f9b-jkgbc   1/1     Running          0          42m
example-catalog-bwt8z                 0/1     ImagePullBackOff   0          3m55s
redhat-marketplace-57p8c              1/1     Running            0          36m
redhat-operators-smxx8                1/1     Running            0          36m
```

namespace にカタログソースを作成すると、その namespace にカタログソース用の Pod が作成されます。前述の出力例では、**example-catalog-bwt8z** Pod のステータスが **ImagePullBackOff** になっています。このステータスは、カタログソースのインデックスイメージのプルに問題があることを示しています。

4. **oc describe** コマンドを使用して、より詳細な情報を得るために Pod を検査します。

```
$ oc describe pod example-catalog-bwt8z -n openshift-marketplace
```

出力例

```
Name:         example-catalog-bwt8z
Namespace:    openshift-marketplace
Priority:     0
Node:         ci-ln-jyryyg2-f76d1-ggdbq-worker-b-vsxjd/10.0.128.2
...
Events:
  Type     Reason          Age              From               Message
  ----     ------          ----             ----               -------
  Normal   Scheduled       48s              default-scheduler  Successfully assigned openshift-marketplace/example-catalog-bwt8z to ci-ln-jyryyf2-f76d1-fgdbq-worker-b-vsxjd
  Normal   AddedInterface  47s              multus             Add eth0 [10.131.0.40/23] from openshift-sdn
  Normal   BackOff         20s (x2 over 46s)  kubelet          Back-off pulling image "quay.io/example-org/example-catalog:v1"
  Warning  Failed          20s (x2 over 46s)  kubelet          Error: ImagePullBackOff
  Normal   Pulling         8s (x3 over 47s)   kubelet          Pulling image "quay.io/example-org/example-catalog:v1"
  Warning  Failed          8s (x3 over 47s)   kubelet          Failed to pull image "quay.io/example-org/example-catalog:v1": rpc error: code = Unknown desc = reading manifest v1 in quay.io/example-org/example-catalog: unauthorized: access to the requested resource is not authorized
  Warning  Failed          8s (x3 over 47s)   kubelet          Error: ErrImagePull
```

前述の出力例では、エラーメッセージは、カタログソースのインデックスイメージが承認問題のために正常にプルできないことを示しています。例えば、インデックスイメージがログイン認証情報を必要とするレジストリーに保存されている場合があります。

関連情報

- Operator Lifecycle Manager の概念およびリソース → カタログソース
- gRPC ドキュメント:接続性の状態

4.6. OPERATOR 条件の管理

dedicated-admin ロールを持つ管理者は、Operator Lifecycle Manager (OLM) を使用して Operator 条件を管理できます。

4.6.1. Operator 条件のオーバーライド

dedicated-admin ロールを持つ管理者は、Operator によって報告されるサポート対象の Operator 条件を無視することもできます。**Spec.Overrides** 配列に Operator 条件が存在する場合、この条件によって **Spec.Conditions** 配列の条件がオーバーライドされます。これを使用することで、**dedicated-admin** 管理者は、Operator が Operator Lifecycle Manager (OLM) に状態を誤って報告している状況に対処できます。

注記

デフォルトでは、**Spec.Overrides** 配列は、**dedicated-admin** ロールを持つ管理者が追加するまで、**OperatorCondition** オブジェクトに存在しません。**Spec.Conditions** 配列も、ユーザーが追加するか、カスタム Operator ロジックの結果として追加されるまで存在しません。

たとえば、アップグレードできないことを常に通信する Operator の既知のバージョンについて考えてみましょう。この場合、Operator がアップグレードできないと通信していますが、Operator をアップグレードすることを推奨します。これは、条件の **type** および **status** を **OperatorCondition** オブジェクトの **Spec.Overrides** 配列に追加して Operator 条件をオーバーライドすることによって実行できます。

前提条件

- **dedicated-admin** ロールを持つユーザーとしてクラスターにアクセスできる。
- **OperatorCondition** オブジェクトを持つ Operator が OLM を使用してインストールされている。

手順

1. Operator の **OperatorCondition** オブジェクトを編集します。

```
$ oc edit operatorcondition <name>
```

2. **Spec.Overrides** 配列をオブジェクトに追加します。

Operator 条件のオーバーライドの例

```
apiVersion: operators.coreos.com/v1
kind: OperatorCondition
metadata:
  name: my-operator
  namespace: operators
spec:
  overrides:
  - type: Upgradeable (1)
    status: "True"
    reason: "upgradeIsSafe"
    message: "This is a known issue with the Operator where it always reports that it cannot be upgraded."
  conditions:
  - type: Upgradeable
    status: "False"
    reason: "migration"
    message: "The operator is performing a migration."
    lastTransitionTime: "2020-08-24T23:15:55Z"
```

(1) このように編集すると、**dedicated-admin** ユーザーはアップグレードの準備状況を **True** に変更できます。

4.6.2. Operator 条件を使用するための Operator の更新

Operator Lifecycle Manager (OLM) は、調整する **ClusterServiceVersion** リソースごとに **OperatorCondition** リソースを自動的に作成します。CSV のすべてのサービスアカウントには、Operator が所有する **OperatorCondition** と対話するための RBAC が付与されます。

Operator の作成者は、Operator が OLM によってデプロイされた後に、独自の条件を設定できるように Operator を開発し、**operator-lib** ライブラリーを使用することができます。Operator 作成者として Operator 条件を設定する方法の詳細は、Operator 条件の有効化 ページを参照してください。

4.6.2.1. デフォルトの設定

後方互換性を維持するために、OLM は **OperatorCondition** リソースがない状態を条件からのオプトアウトとして扱います。そのため、Operator 条件の使用にオプトインする Operator は、Pod の ready プローブが **true** に設定される前に、デフォルトの条件を設定する必要があります。これにより、Operator には、条件を正しい状態に更新するための猶予期間が与えられます。

4.6.3. 関連情報

- Operator 条件

4.7. カスタムカタログの管理

dedicated-admin ロールを持つ管理者とOperator カタログメンテナーは、OpenShift Dedicated の Operator Lifecycle Manager (OLM) で バンドル形式 を使用してパッケージ化したカスタムカタログを作成および管理できます。

重要

Kubernetes は定期的に特定の API を非推奨とし、後続のリリースで削除します。そのため、OpenShift Dedicated のバージョンで、API が削除された Kubernetes バージョンが採用されると、Operator がその API を使用できなくなります。

クラスターでカスタムカタログを使用している場合は、OpenShift Dedicated のバージョンと Operator の互換性の管理 を参照し、Operator 作成者がプロジェクトを更新する際にワークロードの問題や互換性のないアップグレードを避ける方法について詳細を確認してください。

関連情報

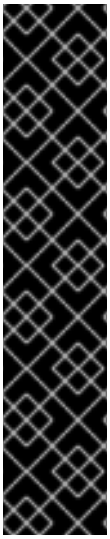
- Red Hat が提供する Operator カタログ

4.7.1. 前提条件

- **opm** CLI がインストールされている。

4.7.2. ファイルベースのカタログ

ファイルベースのカタログは、Operator Lifecycle Manager(OLM) のカタログ形式の最新の反復になります。この形式は、プレーンテキストベース (JSON または YAML) であり、以前の SQLite データベース形式の宣言的な設定の進化であり、完全な下位互換性があります。



重要

OpenShift Dedicated 4.11 以降、デフォルトの Red Hat 提供の Operator カタログはファイルベースのカタログ形式でリリースされます。OpenShift Dedicated 4.6 ~ 4.10 用のデフォルトの Red Hat 提供 Operator カタログは、非推奨の SQLite データベース形式でリリースされました。

opm サブコマンド、フラグ、および SQLite データベース形式に関連する機能も非推奨となり、今後のリリースで削除されます。機能は引き続きサポートされており、非推奨の SQLite データベース形式を使用するカタログに使用する必要があります。

opm index prune などの SQLite データベース形式を使用する **opm** サブコマンドおよびフラグの多くは、ファイルベースのカタログ形式では機能しません。ファイルベースのカタログを使用する方法の詳細は、Operator Framework パッケージ形式 を参照してください。

4.7.2.1. ファイルベースのカタログイメージの作成

opm CLI を使用して、非推奨の SQLite データベース形式を置き換えるプレーンテキストの ファイルベースのカタログ 形式 (JSON または YAML) を使用するカタログイメージを作成できます。

前提条件

- **opm** CLI がインストールされている。
- **podman** バージョン 1.9.3 以降がある。
- バンドルイメージがビルドされ、Docker v2-2 をサポートするレジストリーにプッシュされている。

手順

1. カタログを初期化します。
   a. 次のコマンドを実行して、カタログ用のディレクトリーを作成します。

```
$ mkdir <catalog_dir>
```

   b. **opm generated dockerfile** コマンドを実行して、カタログイメージを構築できる Dockerfile を生成します。

```
$ opm generate dockerfile <catalog_dir> \
    -i registry.redhat.io/openshift4/ose-operator-registry:v4 1
```

   1 **-i** フラグを使用して公式の Red Hat ベースイメージを指定します。それ以外の場合、Dockerfile はデフォルトのアップストリームイメージを使用します。

   Dockerfile は、直前の手順で作成したカタログディレクトリーと同じ親ディレクトリーに存在する必要があります。

   ディレクトリー構造の例

```
. 1
├── <catalog_dir> 2
└── <catalog_dir>.Dockerfile 3
```

1. 親ディレクトリー
2. カタログディレクトリー
3. **opm generate dockerfile** コマンドによって生成された Dockerfile

c. **opm init** コマンドを実行して、カタログに Operator のパッケージ定義を追加します。

```
$ opm init <operator_name> \ 1
    --default-channel=preview \ 2
    --description=./README.md \ 3
    --icon=./operator-icon.svg \ 4
    --output yaml \ 5
    > <catalog_dir>/index.yaml 6
```

1. Operator、またはパッケージ、名前。
2. 指定されていない場合にサブスクリプションがデフォルトで使用するチャネル
3. Operator の **README.md** またはその他のドキュメントへのパス。
4. Operator のアイコンへのパス。
5. 出力形式: JSON または YAML。
6. カタログ設定ファイルを作成するパス。

このコマンドは、指定されたカタログ設定ファイルに **olm.package** 宣言型設定 blob を生成します。

2. **opm render** コマンドを実行して、バンドルをカタログに追加します。

```
$ opm render <registry>/<namespace>/<bundle_image_name>:<tag> \ 1
    --output=yaml \
    >> <catalog_dir>/index.yaml 2
```

1. バンドルイメージのプル仕様。
2. カタログ設定ファイルへのパス。

注記

チャネルには、1 つ以上のバンドルが含まれる必要があります。

3. バンドルのチャネルエントリーを追加します。たとえば、次の例を仕様に合わせて変更し、**<catalog_dir>/index.yaml** ファイルに追加します。

チャネルエントリーの例

チャネルエントリーの例

```
---
schema: olm.channel
package: <operator_name>
name: preview
entries:
  - name: <operator_name>.v0.1.0 (1)
```

(1) **<operator_name>** の後、かつ、バージョンの **v** の前に、ピリオド (**.**) を追加するようにしてください。それ以外の場合、エントリーが **opm validate** コマンドに合格できません。

4. ファイルベースのカタログを検証します。
   a. カタログディレクトリーに対して **opm validate** コマンドを実行します。

   ```
   $ opm validate <catalog_dir>
   ```

   b. エラーコードが **0** であることを確認します。

   ```
   $ echo $?
   ```

   出力例

   ```
   0
   ```

5. **podman build** コマンドを実行して、カタログイメージをビルドします。

```
$ podman build . \
    -f <catalog_dir>.Dockerfile \
    -t <registry>/<namespace>/<catalog_image_name>:<tag>
```

6. カタログイメージをレジストリーにプッシュします。
   a. 必要に応じて、**podman login** コマンドを実行してターゲットレジストリーで認証します。

   ```
   $ podman login <registry>
   ```

   b. **podman push** コマンドを実行して、カタログイメージをプッシュします。

   ```
   $ podman push <registry>/<namespace>/<catalog_image_name>:<tag>
   ```

関連情報

- **opm** CLI リファレンス

4.7.2.2. ファイルベースのカタログイメージの更新またはフィルタリング

opm CLI を使用して、ファイルベースのカタログ形式を使用するカタログイメージを更新またはフィルタリングできます。既存のカタログイメージのコンテンツを抽出すると、必要に応じてカタログを変更できます。たとえば、以下を実行できます。

- パッケージの追加
- パッケージの削除
- 既存のパッケージエントリーの更新
- パッケージ、チャネル、バンドルごとの非推奨メッセージの記載

その後、イメージをカタログの更新バージョンとして再構築できます。

前提条件

- ワークステーションに以下が含まれている。
  - **opm** CLI。
  - **podman** version 1.9.3+。
  - ファイルベースのカタログイメージ。
  - このカタログに関連するワークステーションで最近初期化されたカタログディレクトリー構造。
    初期化されたカタログディレクトリーがない場合は、ディレクトリーを作成し、Dockerfile を生成します。詳細は、「ファイルベースのカタログイメージの作成」手順の「カタログの初期化」手順を参照してください。

手順

1. カタログイメージのコンテンツを YAML 形式でカタログディレクトリーの **index.yaml** ファイルに展開します。

```
$ opm render <registry>/<namespace>/<catalog_image_name>:<tag> \
    -o yaml > <catalog_dir>/index.yaml
```

注記

または、**-o json** フラグを使用して JSON 形式で出力することもできます。

2. 作成された **index.yaml** ファイルの内容を仕様に合わせて変更します。

重要

バンドルがカタログに公開されたら、いずれかのユーザーがバンドルをインストールしていると想定します。カタログ内で以前に公開されたすべてのバンドルに、現在または新しいチャネルヘッドへの更新パスが設定されていることを確認し、そのバージョンがインストールされているユーザーが立ち往生するのを防ぎます。

- Operator を追加するには、「ファイルベースのカタログイメージの作成」手順のパッケージ、バンドル、およびチャネルエントリーを作成する手順に従います。
- Operator を削除するには、パッケージに関連する **olm.package**、**olm.channel**、および **olm.bundle** Blob のセットを削除します。次の例は、カタログから **example-operator** パッケージを削除するために削除する必要があるセットを示しています。

例4.2 削除されたエントリーの例

```
---
defaultChannel: release-2.7
icon:
  base64data: <base64_string>
  mediatype: image/svg+xml
name: example-operator
schema: olm.package
---
entries:
- name: example-operator.v2.7.0
  skipRange: '>=2.6.0 <2.7.0'
- name: example-operator.v2.7.1
  replaces: example-operator.v2.7.0
  skipRange: '>=2.6.0 <2.7.1'
- name: example-operator.v2.7.2
  replaces: example-operator.v2.7.1
  skipRange: '>=2.6.0 <2.7.2'
- name: example-operator.v2.7.3
  replaces: example-operator.v2.7.2
  skipRange: '>=2.6.0 <2.7.3'
- name: example-operator.v2.7.4
  replaces: example-operator.v2.7.3
  skipRange: '>=2.6.0 <2.7.4'
name: release-2.7
package: example-operator
schema: olm.channel
---
image: example.com/example-inc/example-operator-bundle@sha256:<digest>
name: example-operator.v2.7.0
package: example-operator
properties:
- type: olm.gvk
  value:
    group: example-group.example.io
    kind: MyObject
    version: v1alpha1
- type: olm.gvk
  value:
    group: example-group.example.io
    kind: MyOtherObject
    version: v1beta1
- type: olm.package
  value:
    packageName: example-operator
    version: 2.7.0
- type: olm.bundle.object
  value:
    data: <base64_string>
- type: olm.bundle.object
  value:
    data: <base64_string>
relatedImages:
- image: example.com/example-inc/example-related-image@sha256:<digest>
```

```
  name: example-related-image
schema: olm.bundle
---
```

- Operator の非推奨メッセージを追加または更新するには、パッケージの **index.yaml** ファイルと同じディレクトリーに **deprecations.yaml** ファイルがあることを確認してください。**deprecations.yaml** ファイル形式の詳細は、「olm.deprecations スキーマ」を参照してください。

3. 変更を保存します。
4. カタログを検証します。

```
$ opm validate <catalog_dir>
```

5. カタログを再構築します。

```
$ podman build . \
   -f <catalog_dir>.Dockerfile \
   -t <registry>/<namespace>/<catalog_image_name>:<tag>
```

6. 更新されたカタログイメージをレジストリーにプッシュします。

```
$ podman push <registry>/<namespace>/<catalog_image_name>:<tag>
```

検証

1. Web コンソールで、**Administration → Cluster Settings → Configuration** ページで OperatorHub 設定リソースに移動します。
2. カタログソースを追加するか、既存のカタログソースを更新して、更新されたカタログイメージのプル仕様を使用します。
   詳細は、このセクションの関連情報にある「クラスターへのカタログソースの追加」を参照してください。
3. カタログソースが **READY** 状態になったら、**Operators → OperatorHub** ページに移動し、加えた変更が Operator のリストに反映されていることを確認します。

4.7.3. SQLite ベースのカタログ

重要

Operator カタログの SQLite データベース形式は非推奨の機能です。非推奨の機能は依然として OpenShift Dedicated に含まれており、引き続きサポートされますが、この製品の今後のリリースで削除されるため、新規デプロイメントでの使用は推奨されません。

OpenShift Dedicated で非推奨化または削除された主な機能の最新のリストについては、OpenShift Dedicated リリースノートの 非推奨および削除された機能 セクションを参照してください。

4.7.3.1. SQLite ベースのインデックスイメージの作成

opm CLI を使用して、SQLite データベース形式に基づいてインデックスイメージを作成できます。

前提条件

- **opm** CLI がインストールされている。
- **podman** バージョン 1.9.3 以降がある。
- バンドルイメージがビルドされ、Docker v2-2 をサポートするレジストリーにプッシュされている。

手順

1. 新しいインデックスを開始します。

```
$ opm index add \
    --bundles <registry>/<namespace>/<bundle_image_name>:<tag> \ 1
    --tag <registry>/<namespace>/<index_image_name>:<tag> \ 2
    [--binary-image <registry_base_image>] 3
```

1. インデックスに追加するバンドルイメージのコンマ区切りのリスト。
2. インデックスイメージで使用するイメージタグ。
3. オプション: カタログを提供するために使用する代替レジストリーベースイメージ。

2. インデックスイメージをレジストリーにプッシュします。
   a. 必要な場合は、ターゲットレジストリーで認証します。

```
$ podman login <registry>
```

   b. インデックスイメージをプッシュします。

```
$ podman push <registry>/<namespace>/<index_image_name>:<tag>
```

4.7.3.2. SQLite ベースのインデックスイメージの更新

dedicated-admin ロールを持つ管理者は、カスタムインデックスイメージを参照するカタログソースを使用するように OperatorHub を設定した後、インデックスイメージにバンドルイメージを追加することで、クラスター上で利用可能な Operator を最新の状態に保つことができます。

opm index add コマンドを使用して既存インデックスイメージを更新できます。

前提条件

- **opm** CLI がインストールされている。
- **podman** バージョン 1.9.3 以降がある。
- インデックスイメージがビルドされ、レジストリーにプッシュされている。
- インデックスイメージを参照する既存のカタログソースがある。

手順

1. バンドルイメージを追加して、既存のインデックスを更新します。

```
$ opm index add \
    --bundles <registry>/<namespace>/<new_bundle_image>@sha256:<digest> \ 1
    --from-index <registry>/<namespace>/<existing_index_image>:<existing_tag> \ 2
    --tag <registry>/<namespace>/<existing_index_image>:<updated_tag> \ 3
    --pull-tool podman 4
```

1. **--bundles**フラグは、インデックスに追加する他のバンドルイメージのコンマ区切りリストを指定します。
2. **--from-index**フラグは、以前にプッシュされたインデックスを指定します。
3. **--tag**フラグは、更新されたインデックスイメージに適用するイメージタグを指定します。
4. **--pull-tool**フラグは、コンテナーイメージのプルに使用されるツールを指定します。

ここでは、以下のようになります。

<registry>
: **quay.io**や**mirror.example.com**などのレジストリーのホスト名を指定します。

<namespace>
: **ocs-dev**や**abc**など、レジストリーの namespace を指定します。

<new_bundle_image>
: **ocs-operator**など、レジストリーに追加する新しいバンドルイメージを指定します。

<digest>
: **c7f11097a628f092d8bad148406aa0e0951094a03445fd4bc0775431ef683a41**などのバンドルイメージの SHA イメージ ID またはダイジェストを指定します。

<existing_index_image>
: **abc-redhat-operator-index**など、以前にプッシュされたイメージを指定します。

<existing_tag>
: 以前にプッシュしたイメージのタグ (**4** など) を指定します。

<updated_tag>
: 更新されたインデックスイメージに適用するイメージタグ (**4.1** など) を指定します。

コマンドの例

```
$ opm index add \
    --bundles quay.io/ocs-dev/ocs-
operator@sha256:c7f11097a628f092d8bad148406aa0e0951094a03445fd4bc0775431ef683a
41 \
    --from-index mirror.example.com/abc/abc-redhat-operator-index:4 \
    --tag mirror.example.com/abc/abc-redhat-operator-index:4.1 \
    --pull-tool podman
```

2. 更新されたインデックスイメージをプッシュします。

```
$ podman push <registry>/<namespace>/<existing_index_image>:<updated_tag>
```

3. Operator Lifecycle Manager (OLM) がカタログソースで参照されるインデックスイメージを一定間隔で自動的にポーリングした後に、新規パッケージが正常に追加されたことを確認します。

```
$ oc get packagemanifests -n openshift-marketplace
```

4.7.3.3. SQLite ベースのインデックスイメージのフィルタリング

Operator Bundle Format に基づくインデックスイメージは、Operator カタログのコンテナー化されたスナップショットです。パッケージの指定された一覧以外のすべてのインデックスをプルーニングできます。これにより、必要な Operator のみが含まれるソースインデックスのコピーを作成できます。

前提条件

- **podman** バージョン 1.9.3 以降がある。
- **grpurl** (サードパーティーのコマンドラインツール) がある。
- **opm** CLI がインストールされている。
- Docker v2-2 をサポートするレジストリーにアクセスできる。

手順

1. ターゲットレジストリーで認証します。

```
$ podman login <target_registry>
```

2. プルーニングされたインデックスに追加するパッケージのリストを判別します。

   a. コンテナーでプルーニングするソースインデックスイメージを実行します。以下に例を示します。

```
$ podman run -p50051:50051 \
    -it registry.redhat.io/redhat/redhat-operator-index:v4
```

   出力例

```
Trying to pull registry.redhat.io/redhat/redhat-operator-index:v4...
Getting image source signatures
Copying blob ae8a0c23f5b1 done
...
INFO[0000] serving registry                 database=/database/index.db port=50051
```

   b. 別のターミナルセッションで、**grpcurl** コマンドを使用して、インデックスが提供するパッケージのリストを取得します。

```
$ grpcurl -plaintext localhost:50051 api.Registry/ListPackages > packages.out
```

   c. **packages.out** ファイルを検査し、プルーニングされたインデックスに保持したいパッケージ名をこのリストから特定します。以下に例を示します。

パッケージリストのスニペットの例

```
...
{
  "name": "advanced-cluster-management"
}
...
{
  "name": "jaeger-product"
}
...
{
{
  "name": "quay-operator"
}
...
```

d. **podman run** コマンドを実行したターミナルセッションで、**Ctrl** と **C** を押してコンテナープロセスを停止します。

3. 以下のコマンドを実行して、指定したパッケージ以外のすべてのパッケージのソースインデックスをプルーニングします。

```
$ opm index prune \
   -f registry.redhat.io/redhat/redhat-operator-index:v4 \ 1
   -p advanced-cluster-management,jaeger-product,quay-operator \ 2
   [-i registry.redhat.io/openshift4/ose-operator-registry:v4.9] \ 3
   -t <target_registry>:<port>/<namespace>/redhat-operator-index:v4 4
```

1 プルーニングするインデックス。

2 保持するパッケージのコンマ区切りリスト。

3 IBM Power® および IBM Z® イメージにのみ必要: OpenShift Dedicated クラスターのターゲットのメジャーバージョンとマイナーバージョンに一致するタグを持つ Operator Registry ベースイメージ。

4 ビルドされる新規インデックスイメージのカスタムタグ。

4. 以下のコマンドを実行して、新規インデックスイメージをターゲットレジストリーにプッシュします。

```
$ podman push <target_registry>:<port>/<namespace>/redhat-operator-index:v4
```

ここで、**<namespace>** はレジストリー上の既存の namespace になります。

4.7.4. カタログソースと Pod セキュリティーアドミッション

Pod のセキュリティー標準を確保するために、**Pod** セキュリティーアドミッション が OpenShift Dedicated 4.11 で導入されました。SQLite ベースのカタログ形式と、OpenShift Dedicated 4.11 より前にリリースされたバージョンの **opm** CLI ツールを使用してビルドされたカタログソースは、制限付き Pod セキュリティーの適用下では実行できません。

OpenShift Dedicated 4 では、制限付き Pod セキュリティーが namespace にデフォルトで適用されず、カタログソースのデフォルトセキュリティーモードが **legacy** に設定されています。

すべての namespace に対するデフォルトの制限付き適用は、今後の OpenShift Dedicated リリースに組み込まれる予定です。制限付き適用が発生した場合、カタログソース Pod の Pod 仕様のセキュリティーコンテキストは、制限付き Pod のセキュリティー標準に一致する必要があります。カタログソースイメージで別の Pod セキュリティー標準が必要な場合は、namespace の Pod セキュリティーアドミッションラベルを明示的に設定する必要があります。

注記

SQLite ベースのカタログソース Pod を制限付きで実行する必要がない場合は、OpenShift Dedicated 4 でカタログソースを更新する必要はありません。

ただし、制限付きの Pod セキュリティー適用下でカタログソースが確実に実行されるように、今すぐ対策を講じることを推奨します。制限付き Pod セキュリティー適用下でカタログソースが確実に実行されるように対策を講じないと、今後の OpenShift Dedicated リリースでカタログソースが動作しなくなる可能性があります。

カタログの作成者は、次のいずれかのアクションを実行することで、制限付き Pod セキュリティー適用との互換性を有効にできます。

- カタログをファイルベースのカタログ形式に移行します。
- OpenShift Dedicated 4.11 以降でリリースされた **opm** CLI ツールのバージョンを使用してカタログイメージを更新します。

注記

SQLite データベースカタログ形式は非推奨ですが、Red Hat では引き続きサポートされています。将来のリリースでは、SQLite データベース形式はサポートされなくなり、カタログはファイルベースのカタログ形式に移行する必要があります。OpenShift Dedicated 4.11 以降、デフォルトの Red Hat 提供の Operator カタログは、ファイルベースのカタログ形式でリリースされます。ファイルベースのカタログは、制限付き Pod セキュリティー適用と互換性があります。

SQLite データベースカタログイメージを更新したり、カタログをファイルベースのカタログ形式に移行したりしたくない場合は、昇格されたアクセス許可で実行するようにカタログを設定できます。

関連情報

- Pod セキュリティーアドミッションの理解と管理

4.7.4.1. SQLite データベースカタログをファイルベースのカタログ形式に移行する

非推奨の SQLite データベース形式のカタログをファイルベースのカタログ形式に更新できます。

前提条件

- SQLite データベースカタログソースがある。
- **dedicated-admin** ロールを持つユーザーとしてクラスターにアクセスできる。

- OpenShift Dedicated 4 でリリースされた **opm** CLI ツールの最新バージョンが、ワークステーションにインストールされている。

手順

1. 次のコマンドを実行して、SQLite データベースカタログをファイルベースのカタログに移行します。

```
$ opm migrate <registry_image> <fbc_directory>
```

2. 次のコマンドを実行して、ファイルベースのカタログ用の Dockerfile を生成します。

```
$ opm generate dockerfile <fbc_directory> \
  --binary-image \
  registry.redhat.io/openshift4/ose-operator-registry:v4
```

次のステップ

- 生成された Dockerfile をビルドしてタグ付けし、レジストリーにプッシュできます。

関連情報

- クラスターへのカタログソースの追加

4.7.4.2. SQLite データベースカタログイメージの再ビルド

OpenShift Dedicated のお使いのバージョンでリリースされた **opm** CLI ツールの最新バージョンを使用して、SQLite データベースカタログイメージを再ビルドできます。

前提条件

- SQLite データベースカタログソースがある。
- **dedicated-admin** ロールを持つユーザーとしてクラスターにアクセスできる。
- OpenShift Dedicated 4 でリリースされた **opm** CLI ツールの最新バージョンが、ワークステーションにインストールされている。

手順

- 次のコマンドを実行して、最新バージョンの **opm** CLI ツールでカタログを再構築します。

```
$ opm index add --binary-image \
  registry.redhat.io/openshift4/ose-operator-registry:v4 \
  --from-index <your_registry_image> \
  --bundles "" -t \<your_registry_image>
```

4.7.4.3. 昇格された権限で実行するためのカタログの設定

SQLite データベースカタログイメージを更新したり、カタログをファイルベースのカタログ形式に移行したりしたくない場合は、次のアクションを実行して、デフォルトの Pod セキュリティー適用が制限付きに変更されたときにカタログソースが確実に実行されるようにすることができます。

- カタログソース定義でカタログセキュリティーモードをレガシーに手動で設定します。このアクションにより、デフォルトのカタログセキュリティーモードが制限付きに変更された場合でも、カタログが従来のアクセス許可で実行されることが保証されます。
- ベースラインまたは特権付き Pod のセキュリティー適用のために、カタログソースの namespace にラベルを付けます。

注記

SQLite データベースカタログ形式は非推奨ですが、Red Hat では引き続きサポートされています。将来のリリースでは、SQLite データベース形式はサポートされなくなり、カタログはファイルベースのカタログ形式に移行する必要があります。ファイルベースのカタログは、制限付き Pod セキュリティー適用と互換性があります。

前提条件

- SQLite データベースカタログソースがある。
- **dedicated-admin** ロールを持つユーザーとしてクラスターにアクセスできる。
- Pod Security Admission 標準が **baseline** または **privileged** に昇格された実行中の Pod をサポートするターゲット namespace がある。

手順

1. 次の例に示すように、**spec.grpcPodConfig.securityContextConfig** ラベルを **legacy** に設定して、**CatalogSource** 定義を編集します。

   CatalogSource 定義の例

   ```
   apiVersion: operators.coreos.com/v1alpha1
   kind: CatalogSource
   metadata:
     name: my-catsrc
     namespace: my-ns
   spec:
     sourceType: grpc
     grpcPodConfig:
       securityContextConfig: legacy
     image: my-image:latest
   ```

   ヒント

   OpenShift Dedicated 4 では、**spec.grpcPodConfig.securityContextConfig** フィールドはデフォルトで **legacy** に設定されています。OpenShift Dedicated の今後のリリースでは、デフォルト設定が **restricted** に変更される予定です。カタログを制限付き適用で実行できない場合は、このフィールドを手動で **legacy** に設定することを推奨します。

2. 次の例に示すように、**<namespace>.yaml** ファイルを編集して、上位の Pod Security Admission 標準をカタログソース namespace に追加します。

   <namespace>.yaml ファイルの例

   ```
   apiVersion: v1
   ```

```
kind: Namespace
metadata:
...
  labels:
    security.openshift.io/scc.podSecurityLabelSync: "false" (1)
    openshift.io/cluster-monitoring: "true"
    pod-security.kubernetes.io/enforce: baseline (2)
  name: "<namespace_name>"
```

(1) **security.openshift.io/scc.podSecurityLabelSync=false** ラベルを namespace に追加して、Pod のセキュリティーラベルの同期をオフにします。

(2) Pod セキュリティーアドミッションの **pod-security.kubernetes.io/enforce** ラベルを適用します。ラベルを **baseline** または **privileged** に設定します。namespace 内の他のワークロードが **privileged** プロファイルを必要としないかぎり、**baseline** Pod セキュリティープロファイルを使用します。

4.7.5. クラスターへのカタログソースの追加

OpenShift Dedicated クラスターにカタログソースを追加すると、ユーザーが Operator を検出してインストールできるようになります。**dedicated-admin** ロールを持つ管理者は、インデックスイメージを参照する **CatalogSource** オブジェクトを作成できます。OperatorHub はカタログソースを使用してユーザーインターフェイスを設定します。

ヒント

または、Web コンソールを使用してカタログソースを管理できます。**Home → Search** ページからプロジェクトを選択し、**Resources** ドロップダウンをクリックして **CatalogSource** を検索します。個々のソースを作成、更新、削除、無効化、および有効化できます。

前提条件

- インデックスイメージをビルドしてレジストリーにプッシュしている。
- **dedicated-admin** ロールを持つユーザーとしてクラスターにアクセスできる。

手順

1. インデックスイメージを参照する **CatalogSource** オブジェクトを作成します。
   a. 仕様を以下のように変更し、これを **catalogSource.yaml** ファイルとして保存します。

```
apiVersion: operators.coreos.com/v1alpha1
kind: CatalogSource
metadata:
  name: my-operator-catalog
  namespace: openshift-marketplace (1)
  annotations:
    olm.catalogImageTemplate: (2)
      "<registry>/<namespace>/<index_image_name>:v{kube_major_version}.
{kube_minor_version}.{kube_patch_version}"
spec:
  sourceType: grpc
```

```
  grpcPodConfig:
    securityContextConfig: <security_mode> 3
  image: <registry>/<namespace>/<index_image_name>:<tag> 4
  displayName: My Operator Catalog
  publisher: <publisher_name> 5
  updateStrategy:
    registryPoll: 6
      interval: 30m
```

1. カタログソースを全 namespace のユーザーがグローバルに利用できるようにする場合は、**openshift-marketplace** namespace を指定します。それ以外の場合は、そのカタログの別の namespace を対象とし、その namespace のみが利用できるように指定できます。

2. 任意: **olm.catalogImageTemplate** アノテーションをカタログイメージ名に設定し、イメージタグのテンプレートを作成する際に、1 つ以上の Kubernetes クラスターバージョン変数を使用します。

3. **legacy** または **restricted** の値を指定します。フィールドが設定されていない場合、デフォルト値は **legacy** です。今後の OpenShift Dedicated リリースでは、デフォルト値が **restricted** になる予定です。**restricted** 権限でカタログを実行できない場合は、このフィールドを手動で **legacy** に設定することを推奨します。

4. インデックスイメージを指定します。イメージ名の後にタグを指定すると (**:v4** など)、カタログソース Pod が **Always** イメージプルポリシーを使用します。つまり、Pod がコンテナーを起動する前に常にイメージをプルするようになります。**@sha256:<id>** などのダイジェストを指定した場合、イメージプルポリシーは **IfNotPresent** になります。これは、イメージがノード上にまだ存在しない場合にのみ、Pod がイメージをプルすることを意味します。

5. カタログを公開する名前または組織名を指定します。

6. カタログソースは新規バージョンの有無を自動的にチェックし、最新の状態を維持します。

b. このファイルを使用して **CatalogSource** オブジェクトを作成します。

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

2. 以下のリソースが正常に作成されていることを確認します。

a. Pod を確認します。

```
$ oc get pods -n openshift-marketplace
```

出力例

```
NAME                                  READY   STATUS    RESTARTS  AGE
my-operator-catalog-6njx6             1/1     Running   0         28s
marketplace-operator-d9f549946-96sgr  1/1     Running   0         26h
```

b. カタログソースを確認します。

```
$ oc get catalogsource -n openshift-marketplace
```

出力例

```
NAME                 DISPLAY              TYPE PUBLISHER  AGE
my-operator-catalog  My Operator Catalog  grpc           5s
```

c. パッケージマニフェストを確認します。

```
$ oc get packagemanifest -n openshift-marketplace
```

出力例

```
NAME                     CATALOG              AGE
jaeger-product           My Operator Catalog  93s
```

これで OpenShift Dedicated Web コンソールの **OperatorHub** ページから Operator をインストールできるようになりました。

関連情報

- Operator Lifecycle Manager の概念およびリソース → カタログソース

4.7.6. カスタムカタログの削除

dedicated-admin ロールを持つ管理者は、関連するカタログソースを削除することで、以前にクラスターに追加したカスタム Operator カタログを削除できます。

前提条件

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

手順

1. Web コンソールの **Administrator** パースペクティブで、**Home → Search** に移動します。
2. **Project:** リストからプロジェクトを選択します。
3. **Resources** リストから **CatalogSource** を選択します。
4. 削除するカタログの **Options** メニュー  を選択し、**Delete CatalogSource** をクリックします。

4.8. カタログソース POD のスケジューリング

ソースタイプ **grpc** の Operator Lifecycle Manager (OLM) カタログソースが **spec.image** を定義すると、Catalog Operator は、定義されたイメージコンテンツを提供する Pod を作成します。デフォルトでは、この Pod は、その仕様で以下を定義します。

- **kubernetes.io/os=linux** ノードセレクターのみ
- デフォルトの優先クラス名: **system-cluster-critical**。

- 容認なし

管理者は、**CatalogSource** オブジェクトのオプションの **spec.grpcPodConfig** セクションのフィールドを変更すると、これらの値をオーバーライドできます。

重要

Marketplace Operator の **openshift-marketplace** は、デフォルトの **OperatorHub** カスタムリソース (CR) を管理します。この CR は **CatalogSource** オブジェクトを管理します。**CatalogSource** オブジェクトの **spec.grpcPodConfig** セクションのフィールドを変更しようとすると、Marketplace Operator はこれらの変更を自動的に元に戻します。デフォルトでは、**CatalogSource** オブジェクトの **spec.grpcPodConfig** セクションのフィールドを変更すると、Marketplace Operator はこれらの変更を自動的に元に戻します。

CatalogSource オブジェクトに永続的な変更を適用するには、まずデフォルトの **CatalogSource** オブジェクトを無効にする必要があります。

関連情報

- OLM concepts and resources → Catalog source

4.8.1. ローカルレベルでのデフォルト CatalogSource オブジェクトの無効化

デフォルトの **CatalogSource** オブジェクトを無効にすることで、カタログソース Pod などの永続的な変更をローカルレベルで **CatalogSource** オブジェクトに適用できます。デフォルトの **CatalogSource** オブジェクトの設定が組織のニーズを満たさない場合は、デフォルト設定を検討してください。デフォルトでは、**CatalogSource** オブジェクトの **spec.grpcPodConfig** セクションのフィールドを変更すると、Marketplace Operator はこれらの変更を自動的に元に戻します。

Marketplace Operator の **openshift-marketplace** は、**OperatorHub** のデフォルトのカスタムリソース (CR) を管理します。**OperatorHub** は **CatalogSource** オブジェクトを管理します。

CatalogSource オブジェクトに永続的な変更を適用するには、まずデフォルトの **CatalogSource** オブジェクトを無効にする必要があります。

手順

- すべてのデフォルトの **CatalogSource** オブジェクトをローカルレベルで無効にするには、次のコマンドを入力します。

```
$ oc patch operatorhub cluster -p '{"spec": {"disableAllDefaultSources": true}}' --type=merge
```

注記

また、デフォルトの **OperatorHub** CR を設定して、すべての **CatalogSource** オブジェクトを無効にするか、または特定のオブジェクトを無効にすることもできます。

関連情報

- OperatorHub カスタムリソース

4.8.2. カタログソース Pod のノードセレクターのオーバーライド

前提条件

- **spec.image** を持つソースタイプ **grpc** の **CatalogSource** オブジェクトが定義されている。
- **dedicated-admin** ロールを持つユーザーとしてクラスターにアクセスできる。

手順

- **CatalogSource** オブジェクトを編集し、**spec.grpcPodConfig** セクションを追加または変更して、以下を含めます。

```
grpcPodConfig:
  nodeSelector:
    custom_label: <label>
```

<label> は、カタログソース Pod がスケジュールに使用するノードセレクターのラベルです。

関連情報

- ノードセレクターの使用による特定ノードへの Pod の配置

4.8.3. カタログソース Pod の優先度クラス名のオーバーライド

前提条件

- **spec.image** を持つソースタイプ **grpc** の **CatalogSource** オブジェクトが定義されている。
- **dedicated-admin** ロールを持つユーザーとしてクラスターにアクセスできる。

手順

- **CatalogSource** オブジェクトを編集し、**spec.grpcPodConfig** セクションを追加または変更して、以下を含めます。

```
grpcPodConfig:
  priorityClassName: <priority_class>
```

<priority_class> は次のいずれかです。

  - Kubernetes によって提供されるデフォルトの優先度クラスの 1 つ: **system-cluster-critical** または **system-node-critical**
  - デフォルトの優先度を割り当てる空のセット (**""**)
  - 既存およびカスタム定義の優先度クラス

注記

以前は、オーバーライドできる唯一の Pod スケジューリングパラメーターは **priorityClassName** でした。これは、**operatorframework.io/priorityclass** アノテーションを **CatalogSource** オブジェクトに追加することによって行われました。以下に例を示します。

```
apiVersion: operators.coreos.com/v1alpha1
kind: CatalogSource
metadata:
  name: example-catalog
  namespace: openshift-marketplace
  annotations:
    operatorframework.io/priorityclass: system-cluster-critical
```

CatalogSource オブジェクトがアノテーションと **spec.grpcPodConfig.priorityClassName** の両方を定義する場合、アノテーションは設定パラメーターよりも優先されます。

関連情報

- Pod の優先度クラス

4.8.4. カタログソース Pod の Toleration のオーバーライド

前提条件

- **spec.image** を持つソースタイプ **grpc** の **CatalogSource** オブジェクトが定義されている。
- **dedicated-admin** ロールを持つユーザーとしてクラスターにアクセスできる。

手順

- **CatalogSource** オブジェクトを編集し、**spec.grpcPodConfig** セクションを追加または変更して、以下を含めます。

```
grpcPodConfig:
  tolerations:
    - key: "<key_name>"
      operator: "<operator_type>"
      value: "<value>"
      effect: "<effect>"
```

関連情報

- テイントおよび容認 (Toleration) について

4.9. OPERATOR 関連の問題のトラブルシューティング

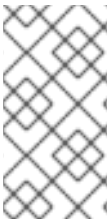
Operator に問題が発生した場合には、Operator Subscription のステータスを確認します。クラスター全体で Operator Pod の正常性を確認し、診断用に Operator ログを収集します。

4.9.1. Operator サブスクリプションの状態のタイプ

サブスクリプションは状態についての以下のタイプを報告します。

表4.2 サブスクリプションの状態のタイプ

状態	説明
CatalogSourcesUnhealthy	解決に使用される一部のまたはすべてのカタログソースは正常ではありません。
InstallPlanMissing	サブスクリプションのインストール計画がありません。
InstallPlanPending	サブスクリプションのインストール計画はインストールの保留中です。
InstallPlanFailed	サブスクリプションのインストール計画が失敗しました。
ResolutionFailed	サブスクリプションの依存関係の解決に失敗しました。



注記

デフォルトの OpenShift Dedicated クラスター Operator は、Cluster Version Operator (CVO) によって管理されます。この Operator には **Subscription** オブジェクトがありません。アプリケーション Operator は、Operator Lifecycle Manager (OLM) によって管理されます。この Operator には **Subscription** オブジェクトがあります。

関連情報

- カタログの正常性要件

4.9.2. CLI を使用した Operator サブスクリプションステータスの表示

CLI を使用して Operator サブスクリプションステータスを表示できます。

前提条件

- **dedicated-admin** ロールを持つユーザーとしてクラスターにアクセスできる。
- OpenShift CLI (**oc**) がインストールされている。

手順

1. Operator サブスクリプションをリスト表示します。

```
$ oc get subs -n <operator_namespace>
```

2. **oc describe** コマンドを使用して、**Subscription** リソースを検査します。

```
$ oc describe sub <subscription_name> -n <operator_namespace>
```

3. コマンド出力で、**Conditions** セクションで Operator サブスクリプションの状態タイプのステータスを確認します。以下の例では、利用可能なすべてのカタログソースが正常であるため、**CatalogSourcesUnhealthy** 状態タイプのステータスは **false** になります。

出力例

```
Name:         cluster-logging
Namespace:    openshift-logging
Labels:       operators.coreos.com/cluster-logging.openshift-logging=
Annotations:  <none>
API Version:  operators.coreos.com/v1alpha1
Kind:         Subscription
# ...
Conditions:
  Last Transition Time:  2019-07-29T13:42:57Z
  Message:               all available catalogsources are healthy
  Reason:                AllCatalogSourcesHealthy
  Status:                False
  Type:                  CatalogSourcesUnhealthy
# ...
```

注記

デフォルトの OpenShift Dedicated クラスター Operator は、Cluster Version Operator (CVO) によって管理されます。この Operator には **Subscription** オブジェクトがありません。アプリケーション Operator は、Operator Lifecycle Manager (OLM) によって管理されます。この Operator には **Subscription** オブジェクトがあります。

4.9.3. CLI を使用した Operator カタログソースのステータス表示

Operator カタログソースのステータスは、CLI を使用して確認できます。

前提条件

- **dedicated-admin** ロールを持つユーザーとしてクラスターにアクセスできる。
- OpenShift CLI (**oc**) がインストールされている。

手順

1. namespace のカタログソースをリスト表示します。例えば、クラスター全体のカタログソースに使用されている **openshift-marketplace**namespace を確認することができます。

```
$ oc get catalogsources -n openshift-marketplace
```

出力例

```
NAME                  DISPLAY               TYPE   PUBLISHER   AGE
certified-operators   Certified Operators   grpc   Red Hat     55m
community-operators   Community Operators   grpc   Red Hat     55m
example-catalog       Example Catalog       grpc   Example Org 2m25s
redhat-marketplace    Red Hat Marketplace   grpc   Red Hat     55m
redhat-operators      Red Hat Operators     grpc   Red Hat     55m
```

2. カタログソースの詳細やステータスを確認するには、**oc describe** コマンドを使用します。

```
$ oc describe catalogsource example-catalog -n openshift-marketplace
```

出力例

```
Name:         example-catalog
Namespace:    openshift-marketplace
Labels:       <none>
Annotations:  operatorframework.io/managed-by: marketplace-operator
              target.workload.openshift.io/management: {"effect": "PreferredDuringScheduling"}
API Version:  operators.coreos.com/v1alpha1
Kind:         CatalogSource
# ...
Status:
  Connection State:
    Address:              example-catalog.openshift-marketplace.svc:50051
    Last Connect:         2021-09-09T17:07:35Z
    Last Observed State:  TRANSIENT_FAILURE
  Registry Service:
    Created At:         2021-09-09T17:05:45Z
    Port:               50051
    Protocol:           grpc
    Service Name:       example-catalog
    Service Namespace:  openshift-marketplace
# ...
```

前述の出力例では、最後に観測された状態が **TRANSIENT_FAILURE** となっています。この状態は、カタログソースの接続確立に問題があることを示しています。

3. カタログソースが作成された namespace の Pod をリストアップします。

```
$ oc get pods -n openshift-marketplace
```

出力例

```
NAME                                  READY   STATUS             RESTARTS   AGE
certified-operators-cv9nn             1/1     Running            0          36m
community-operators-6v8lp             1/1     Running            0          36m
marketplace-operator-86bfc75f9b-jkgbc   1/1     Running            0          42m
example-catalog-bwt8z                 0/1     ImagePullBackOff   0          3m55s
redhat-marketplace-57p8c              1/1     Running            0          36m
redhat-operators-smxx8                1/1     Running            0          36m
```

namespace にカタログソースを作成すると、その namespace にカタログソース用の Pod が作成されます。前述の出力例では、**example-catalog-bwt8z** Pod のステータスが **ImagePullBackOff** になっています。このステータスは、カタログソースのインデックスイメージのプルに問題があることを示しています。

4. **oc describe** コマンドを使用して、より詳細な情報を得るために Pod を検査します。

```
$ oc describe pod example-catalog-bwt8z -n openshift-marketplace
```

出力例

```
Name:         example-catalog-bwt8z
Namespace:    openshift-marketplace
Priority:     0
```

```
Node:         ci-ln-jyryyg2-f76d1-ggdbq-worker-b-vsxjd/10.0.128.2
...
Events:
  Type     Reason          Age                From               Message
  ----     ------          ----               ----               -------
  Normal   Scheduled       48s                default-scheduler  Successfully assigned openshift-marketplace/example-catalog-bwt8z to ci-ln-jyryyf2-f76d1-fgdbq-worker-b-vsxjd
  Normal   AddedInterface  47s                multus             Add eth0 [10.131.0.40/23] from openshift-sdn
  Normal   BackOff         20s (x2 over 46s)  kubelet            Back-off pulling image "quay.io/example-org/example-catalog:v1"
  Warning  Failed          20s (x2 over 46s)  kubelet            Error: ImagePullBackOff
  Normal   Pulling         8s (x3 over 47s)   kubelet            Pulling image "quay.io/example-org/example-catalog:v1"
  Warning  Failed          8s (x3 over 47s)   kubelet            Failed to pull image "quay.io/example-org/example-catalog:v1": rpc error: code = Unknown desc = reading manifest v1 in quay.io/example-org/example-catalog: unauthorized: access to the requested resource is not authorized
  Warning  Failed          8s (x3 over 47s)   kubelet            Error: ErrImagePull
```

前述の出力例では、エラーメッセージは、カタログソースのインデックスイメージが承認問題のために正常にプルできないことを示しています。例えば、インデックスイメージがログイン認証情報を必要とするレジストリーに保存されている場合があります。

関連情報

- gRPC ドキュメント:接続性の状態

4.9.4. Operator Pod ステータスのクエリー

クラスター内の Operator Pod およびそれらのステータスをリスト表示できます。詳細な Operator Pod の要約を収集することもできます。

前提条件

- **dedicated-admin** ロールを持つユーザーとしてクラスターにアクセスできる。
- API サービスが機能している。
- OpenShift CLI (**oc**) がインストールされている。

手順

1. クラスターで実行されている Operator をリスト表示します。出力には、Operator バージョン、可用性、およびアップタイムの情報が含まれます。

```
$ oc get clusteroperators
```

2. Operator の namespace で実行されている Operator Pod をリスト表示し、Pod のステータス、再起動、および経過時間をリスト表示します。

```
$ oc get pod -n <operator_namespace>
```

3. 詳細な Operator Pod の要約を出力します。

```
$ oc describe pod <operator_pod_name> -n <operator_namespace>
```

4.9.5. Operator ログの収集

Operator の問題が発生した場合、Operator Pod ログから詳細な診断情報を収集できます。

前提条件

- **dedicated-admin** ロールを持つユーザーとしてクラスターにアクセスできる。
- API サービスが機能している。
- OpenShift CLI (**oc**) がインストールされている。
- コントロールプレーンまたはコントロールプレーンマシンの完全修飾ドメイン名がある。

手順

1. Operator の namespace で実行されている Operator Pod、Pod のステータス、再起動、および経過時間をリスト表示します。

```
$ oc get pods -n <operator_namespace>
```

2. Operator Pod のログを確認します。

```
$ oc logs pod/<pod_name> -n <operator_namespace>
```

Operator Pod に複数のコンテナーがある場合、前述のコマンドにより各コンテナーの名前が含まれるエラーが生成されます。個別のコンテナーからログをクエリーします。

```
$ oc logs pod/<operator_pod_name> -c <container_name> -n <operator_namespace>
```

3. API が機能しない場合には、代わりに SSH を使用して各コントロールプレーンノードで Operator Pod およびコンテナーログを確認します。**<master-node>.<cluster_name>.<base_domain>** を適切な値に置き換えます。

   a. 各コントロールプレーンノードの Pod をリスト表示します。

```
$ ssh core@<master-node>.<cluster_name>.<base_domain> sudo crictl pods
```

   b. Operator Pod で **Ready** ステータスが表示されない場合は、Pod のステータスを詳細に検査します。**<operator_pod_id>** を直前のコマンドの出力にリスト表示されている Operator Pod の ID に置き換えます。

```
$ ssh core@<master-node>.<cluster_name>.<base_domain> sudo crictl inspectp
<operator_pod_id>
```

   c. Operator Pod に関連するコンテナーをリスト表示します。

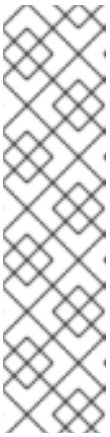
```
$ ssh core@<master-node>.<cluster_name>.<base_domain> sudo crictl ps --pod=
<operator_pod_id>
```

d. **Ready** ステータスが Operator コンテナーに表示されない場合は、コンテナーのステータスを詳細に検査します。**<container_id>** を前述のコマンドの出力に一覧表示されているコンテナー ID に置き換えます。

```
$ ssh core@<master-node>.<cluster_name>.<base_domain> sudo crictl inspect
<container_id>
```

e. **Ready** ステータスが表示されない Operator コンテナーのログを確認します。**<container_id>** を前述のコマンドの出力に一覧表示されているコンテナー ID に置き換えます。

```
$ ssh core@<master-node>.<cluster_name>.<base_domain> sudo crictl logs -f
<container_id>
```



注記

Red Hat Enterprise Linux CoreOS (RHCOS) を実行する OpenShift Dedicated 4 クラスターノードは変更できず、Operator を使用してクラスターの変更を適用します。SSH を使用したクラスターノードへのアクセスは推奨されません。SSH 経由で診断データの収集を試行する前に、**oc adm must gather** およびその他の **oc** コマンドを実行して収集されるデータが十分であるかどうかを確認してください。ただし、OpenShift Dedicated API が使用できない場合、または kubelet がターゲットノード上で適切に機能していない場合は、**oc** 操作が影響を受けます。この場合は、代わりに **ssh core@<node>.<cluster_name>.<base_domain>** を使用してノードにアクセスできます。

第5章 OPERATOR の開発

5.1. OPERATOR SDK について

Operator Framework は **Operator** と呼ばれる 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 Kubernetes の拡張性を利用して、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 が実行される場所であればどこでも実行することができます。

Operator により、Kubernetes の上部の複雑で、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、現時点での Operator の作成は、低レベルの API の使用、ボイラープレートの作成、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、困難になる場合があります。

Operator Framework のコンポーネントである Operator SDK は、Operator 開発者が Operator のビルド、テストおよびデプロイに使用できるコマンドラインインターフェイス (CLI) ツールを提供します。

Operator SDK を使用する理由

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。Operator 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ボイラープレートコードの量を減らします。

Operator SDK は、controller-runtime ライブラリーを使用して、以下の機能を提供することで Operator を容易に作成する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キャフォールディングツールおよびコード生成ツール
- Operator Lifecycle Manager (OLM) との統合による、クラスターでの Operator のパッケージング、インストール、および実行の単純化
- 共通する Operator ユースケースに対応する拡張機能
- 生成された Go ベースの Operator でメトリクスを自動的に設定し、Prometheus Operator がデプロイされているクラスターで使用

OpenShift Dedicated への dedicated-admin アクセス権を持つ Operator 作成者は、Operator SDK CLI を使用して、Go、Ansible、Java、または Helm ベースの独自の Operator を開発できます。Kubebuilder は Go ベースの Operator のスキャフォールディングソリューションとして Operator SDK に組み込まれます。つまり、既存の Kubebuilder プロジェクトは Operator SDK でそのまま使用でき、引き続き機能します。

注記

OpenShift Dedicated 4 は Operator SDK 1.31.0 をサポートします。

5.1.1. Operator について

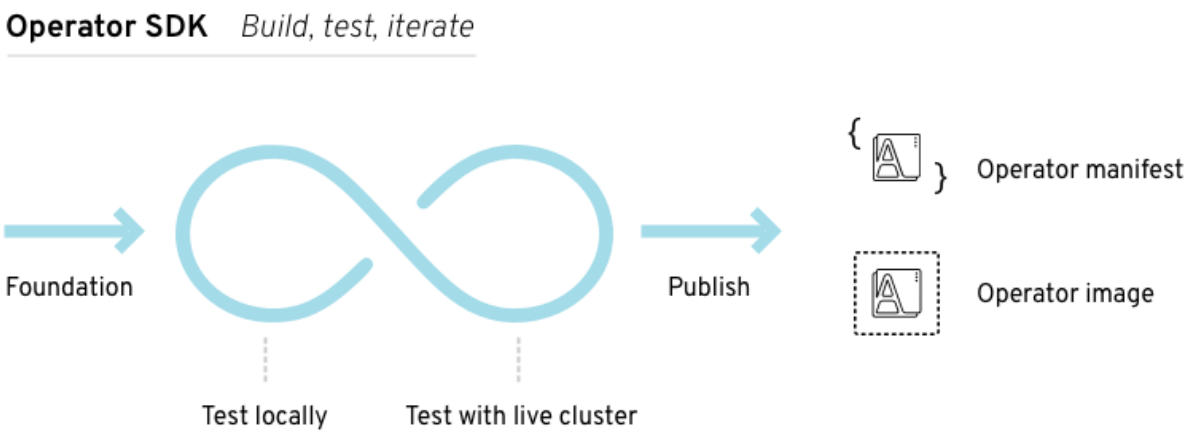
基本的な Operator の概念および用語の概要については、Operator について を参照してください。

5.1.2. 開発ワークフロー

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イス (CLI) を使用した 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5.1 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。

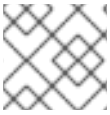
5.1.3. 関連情報

- 認定 Operator ビルドガイド

5.2. OPERATOR SDK CLI のインストール

Operator SDK は、Operator 開発者が Operator のビルド、テストおよびデプロイに使用できるコマンドラインインターフェイス (CLI) ツールを提供します。ワークステーションに Operator 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準備を整えることができます。

OpenShift Dedicated への dedicated-admin アクセス権を持つ Operator 作成者は、Operator SDK CLI を使用して、Go、Ansible、Java、または Helm ベースの独自の Operator を開発できます。Kubebuilder は Go ベースの Operator のスキャフォールディングソリューションとして Operator SDK に組み込まれます。つまり、既存の Kubebuilder プロジェクトは Operator SDK でそのまま使用でき、引き続き機能します。



注記

OpenShift Dedicated 4 は Operator SDK 1.31.0 をサポートします。

5.2.1. Linux での Operator SDK CLI のインストール

OpenShift SDK CLI ツールは Linux にインストールできます。

前提条件

- Go v1.19 以降
- **docker** v17.03+、**podman** v1.9.3+、または **buildah** v1.7+

手順

1. OpenShift ミラーサイト に移動します。
2. 最新の 4 ディレクトリーから、Linux 用の最新バージョンの tarball をダウンロードします。
3. アーカイブを展開します。

```
$ tar xvf operator-sdk-v1.31.0-ocp-linux-x86_64.tar.gz
```

4. ファイルを実行可能にします。

```
$ chmod +x operator-sdk
```

5. デプロイメントされた **operator-sdk** バイナリーを **PATH** にあるディレクトリーに移動します。

ヒント

PATH を確認するには、以下を実行します。

```
$ echo $PATH
```

```
$ sudo mv ./operator-sdk /usr/local/bin/operator-sdk
```

検証

- Operator SDK CLI のインストール後に、これが利用可能であることを確認します。

```
$ operator-sdk version
```

出力例

```
operator-sdk version: "v1.31.0-ocp", ...
```

5.2.2. macOS への Operator SDK CLI のインストール

macOS に OpenShift SDK CLI ツールをインストールできます。

前提条件

- Go v1.19 以降
- **docker** v17.03+、**podman** v1.9.3+、または **buildah** v1.7+

手順

1. **amd64** アーキテクチャーの場合は、**amd64** アーキテクチャーの OpenShift ミラーサイト に移動します。
2. 最新の 4 ディレクトリーから、macOS 用の最新バージョンの tarball をダウンロードします。
3. 以下のコマンドを実行して、**amd64** アーキテクチャー用の Operator SDK アーカイブを解凍します。

```
$ tar xvf operator-sdk-v1.31.0-ocp-darwin-x86_64.tar.gz
```

4. 次のコマンドを実行して、ファイルを実行可能にします。

```
$ chmod +x operator-sdk
```

5. 次のコマンドを実行して、抽出した **operator-sdk** バイナリーを **PATH** 上のディレクトリーに移動します。

ヒント

次のコマンドを実行して、**PATH** を確認します。

```
$ echo $PATH
```

```
$ sudo mv ./operator-sdk /usr/local/bin/operator-sdk
```

検証

- Operator SDK CLI をインストールしたら、次のコマンドを実行して、それが使用可能であることを確認します。

```
$ operator-sdk version
```

出力例

```
operator-sdk version: "v1.31.0-ocp", ...
```

5.3. GO ベースの OPERATOR

5.3.1. Go ベースの Operator の Operator SDK チュートリアル

Operator 開発者は、Operator SDK での Go プログラミング言語のサポートを利用して、Go ベースの Memcached Operator のサンプルをビルドして、分散キー/値のストアを作成し、そのライフサイクルを管理することができます。

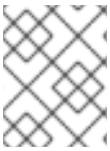
このプロセスは、Operator Framework の 2 つの重要な設定要素を使用して実行されます。

Operator SDK

operator-sdk CLI ツールおよび **controller-runtime** ライブラリー API

Operator Lifecycle Manager (OLM)

クラスター上の 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)



注記

このチュートリアルでは、OpenShift Container Platform ドキュメントの Go ベースの Operator の Operator SDK の使用を開始する よりも詳細に説明します。

5.3.1.1. 前提条件

- Operator SDK CLI がインストールされている。
- OpenShift CLI (**oc**) 4 以降がインストールされている。
- Go 1.19 以降
- **dedicated-admin** 権限を持つアカウントを使用して、**oc** で OpenShift Dedicated クラスターにログインしている。
- クラスターがイメージをプルできるように、イメージをプッシュするリポジトリーを public として設定するか、イメージプルシークレットを設定している。

関連情報

- Operator SDK CLI のインストール
- Getting started with the OpenShift CLI

5.3.1.2. プロジェクトの作成

Operator SDK CLI を使用して **memcached-operator** というプロジェクトを作成します。

手順

1. プロジェクトのディレクトリーを作成します。

```
$ mkdir -p $HOME/projects/memcached-operator
```

2. ディレクトリーに切り替えます。

```
$ cd $HOME/projects/memcached-operator
```

3. Go モジュールのサポートをアクティブにします。

```
$ export GO111MODULE=on
```

4. **operator-sdk init** コマンドを実行してプロジェクトを初期化します。

```
$ operator-sdk init \
    --domain=example.com \
    --repo=github.com/example-inc/memcached-operator
```

注記

operator-sdk init コマンドは、デフォルトで Go プラグインを使用します。

operator-sdk init コマンドは、Go モジュール と使用する **go.mod** ファイルを生成します。生成されるファイルには有効なモジュールパスが必要であるため、**$GOPATH/src/** 外のプロジェクトを作成する場合は、**--repo** フラグが必要です。

5.3.1.2.1. PROJECT ファイル

operator-sdk init コマンドで生成されるファイルの 1 つに、Kubebuilder の **PROJECT** ファイルがあります。プロジェクトルートから実行される後続の **operator-sdk** コマンドおよび **help** 出力は、このファイルを読み取り、プロジェクトタイプが Go であることを認識しています。以下に例を示します。

```
domain: example.com
layout:
- go.kubebuilder.io/v3
projectName: memcached-operator
repo: github.com/example-inc/memcached-operator
version: "3"
plugins:
  manifests.sdk.operatorframework.io/v2: {}
  scorecard.sdk.operatorframework.io/v2: {}
  sdk.x-openshift.io/v1: {}
```

5.3.1.2.2. Manager について

Operator の主なプログラムは、Manager を初期化して実行する **main.go** ファイルです。Manager はすべてのカスタムリソース (CR) API 定義の Scheme を自動的に登録し、コントローラーおよび Webhook を設定して実行します。

Manager は、すべてのコントローラーがリソースの監視をする namespace を制限できます。

```
mgr, err := ctrl.NewManager(cfg, manager.Options{Namespace: namespace})
```

デフォルトで、Manager は Operator が実行される namespace を監視します。すべての namespace を確認するには、**namespace** オプションを空のままにすることができます。

```
mgr, err := ctrl.NewManager(cfg, manager.Options{Namespace: ""})
```

MultiNamespacedCacheBuilder 関数を使用して、特定の namespace セットを監視することもできます。

```
var namespaces []string (1)
mgr, err := ctrl.NewManager(cfg, manager.Options{ (2)
  NewCache: cache.MultiNamespacedCacheBuilder(namespaces),
```

```
})
```

1. namespace のリスト
2. **Cmd** 構造を作成し、共有依存関係を提供してコンポーネントを起動します。

5.3.1.2.3. 複数グループ API について

API およびコントローラーを作成する前に、Operator に複数の API グループが必要かどうかを検討してください。このチュートリアルでは、単一グループ API のデフォルトケースについて説明しますが、複数グループ API をサポートするようにプロジェクトのレイアウトを変更するには、以下のコマンドを実行します。

```
$ operator-sdk edit --multigroup=true
```

このコマンドにより、**PROJECT** ファイルが更新されます。このファイルは、以下の例のようになります。

```
domain: example.com
layout: go.kubebuilder.io/v3
multigroup: true
...
```

複数グループプロジェクトの場合、API Go タイプのファイルが **apis/<group>/<version>/** ディレクトリーに作成され、コントローラーは **controllers/<group>/** ディレクトリーに作成されます。続いて、Dockerfile が適宜更新されます。

追加リソース

- 複数グループのプロジェクトへの移行に関する詳細は、Kubebuilder のドキュメント を参照してください。

5.3.1.3. API およびコントローラーの作成

Operator SDK CLI を使用してカスタムリソース定義 (CRD) API およびコントローラーを作成します。

手順

1. 以下のコマンドを実行して、グループ **cache**、バージョン、**v1**、および種類 **Memcached** を指定して API を作成します。

```
$ operator-sdk create api \
    --group=cache \
    --version=v1 \
    --kind=Memcached
```

2. プロンプトが表示されたら **y** を入力し、リソースとコントローラーの両方を作成します。

```
Create Resource [y/n]
y
Create Controller [y/n]
y
```

出力例

```
Writing scaffold for you to edit...
api/v1/memcached_types.go
controllers/memcached_controller.go
...
```

このプロセスでは、**api/v1/memcached_types.go** で **Memcached** リソース API が生成され、**controllers/memcached_controller.go** でコントローラーが生成されます。

5.3.1.3.1. API の定義

Memcached カスタムリソース (CR) の API を定義します。

手順

1. **api/v1/memcached_types.go** で Go タイプの定義を変更し、以下の **spec** および **status** を追加します。

```
// MemcachedSpec defines the desired state of Memcached
type MemcachedSpec struct {
 // +kubebuilder:validation:Minimum=0
 // Size is the size of the memcached deployment
 Size int32 `json:"size"`
}

// MemcachedStatus defines the observed state of Memcached
type MemcachedStatus struct {
 // Nodes are the names of the memcached pods
 Nodes []string `json:"nodes"`
}
```

2. リソースタイプ用に生成されたコードを更新します。

```
$ make generate
```

ヒント

***_types.go** ファイルの変更後は、**make generate** コマンドを実行し、該当するリソースタイプ用に生成されたコードを更新する必要があります。

上記の Makefile ターゲットは **controller-gen** ユーティリティーを呼び出して、**api/v1/zz_generated.deepcopy.go** ファイルを更新します。これにより、API Go タイプの定義は、すべての Kind タイプが実装する必要のある **runtime.Object** インターフェイスを実装します。

5.3.1.3.2. CRD マニフェストの生成

API が **spec** フィールドと **status** フィールドおよびカスタムリソース定義 (CRD) 検証マーカーで定義された後に、CRD マニフェストを生成できます。

手順

- 以下のコマンドを実行し、CRD マニフェストを生成して更新します。

```
$ make manifests
```

この Makefile ターゲットは **controller-gen** ユーティリティーを呼び出し、**config/crd/bases/cache.example.com_memcacheds.yaml** ファイルに CRD マニフェストを生成します。

5.3.1.3.2.1. OpenAPI 検証

OpenAPIv3 スキーマは、マニフェストの生成時に **spec.validation** ブロックの CRD マニフェストに追加されます。この検証ブロックにより、Kubernetes が作成または更新時に Memcached CR のプロパティーを検証できます。

API の検証を設定するには、マーカーまたはアノテーションを使用できます。これらのマーカーには、**+kubebuilder:validation** 接頭辞が常にあります。

関連情報

- API コードでのマーカーの使用に関する詳細は、以下の Kubebuilder ドキュメントを参照してください。
  - CRD generation
  - Markers
  - List of OpenAPIv3 validation markers
- CRD の OpenAPIv3 検証スキーマに関する詳細は、Kubernetes のドキュメント を参照してください。

5.3.1.4. コントローラーの実装

新規 API およびコントローラーの作成後に、コントローラーロジックを実装することができます。

手順

- この例では、生成されたコントローラーファイル **controllers/memcached_controller.go** を以下の実装例に置き換えます。

例5.1 **memcached_controller.go** の例

```
/*
Copyright 2020.

Licensed under the Apache License, Version 2.0 (the "License");
you may not use this file except in compliance with the License.
You may obtain a copy of the License at

    http://www.apache.org/licenses/LICENSE-2.0

Unless required by applicable law or agreed to in writing, software
distributed under the License is distributed on an "AS IS" BASIS,
WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
See the License for the specific language governing permissions and
limitations under the License.
```

```
*/

package controllers

import (
	appsv1 "k8s.io/api/apps/v1"
	corev1 "k8s.io/api/core/v1"
	"k8s.io/apimachinery/pkg/api/errors"
	metav1 "k8s.io/apimachinery/pkg/apis/meta/v1"
	"k8s.io/apimachinery/pkg/types"
	"reflect"

	"context"

	"github.com/go-logr/logr"
	"k8s.io/apimachinery/pkg/runtime"
	ctrl "sigs.k8s.io/controller-runtime"
	"sigs.k8s.io/controller-runtime/pkg/client"
	ctrllog "sigs.k8s.io/controller-runtime/pkg/log"

	cachev1 "github.com/example-inc/memcached-operator/api/v1"
)

// MemcachedReconciler reconciles a Memcached object
type MemcachedReconciler struct {
	client.Client
	Log    logr.Logger
	Scheme *runtime.Scheme
}

//
+kubebuilder:rbac:groups=cache.example.com,resources=memcacheds,verbs=get;list;watch
;create;update;patch;delete
//
+kubebuilder:rbac:groups=cache.example.com,resources=memcacheds/status,verbs=get;up
date;patch
//
+kubebuilder:rbac:groups=cache.example.com,resources=memcacheds/finalizers,verbs=upd
ate
//
+kubebuilder:rbac:groups=apps,resources=deployments,verbs=get;list;watch;create;update;
patch;delete
// +kubebuilder:rbac:groups=core,resources=pods,verbs=get;list;

// Reconcile is part of the main kubernetes reconciliation loop which aims to
// move the current state of the cluster closer to the desired state.
// TODO(user): Modify the Reconcile function to compare the state specified by
// the Memcached object against the actual cluster state, and then
// perform operations to make the cluster state reflect the state specified by
// the user.
//
// For more details, check Reconcile and its Result here:
// - https://pkg.go.dev/sigs.k8s.io/controller-runtime@v0.7.0/pkg/reconcile
func (r *MemcachedReconciler) Reconcile(ctx context.Context, req ctrl.Request)
(ctrl.Result, error) {
	//log := r.Log.WithValues("memcached", req.NamespacedName)
```

```
	log := ctrllog.FromContext(ctx)
	// Fetch the Memcached instance
	memcached := &cachev1.Memcached{}
	err := r.Get(ctx, req.NamespacedName, memcached)
	if err != nil {
		if errors.IsNotFound(err) {
			// Request object not found, could have been deleted after reconcile
request.
			// Owned objects are automatically garbage collected. For additional
cleanup logic use finalizers.
			// Return and don't requeue
			log.Info("Memcached resource not found. Ignoring since object must be
deleted")
			return ctrl.Result{}, nil
		}
		// Error reading the object - requeue the request.
		log.Error(err, "Failed to get Memcached")
		return ctrl.Result{}, err
	}

	// Check if the deployment already exists, if not create a new one
	found := &appsv1.Deployment{}
	err = r.Get(ctx, types.NamespacedName{Name: memcached.Name, Namespace:
memcached.Namespace}, found)
	if err != nil && errors.IsNotFound(err) {
		// Define a new deployment
		dep := r.deploymentForMemcached(memcached)
		log.Info("Creating a new Deployment", "Deployment.Namespace",
dep.Namespace, "Deployment.Name", dep.Name)
		err = r.Create(ctx, dep)
		if err != nil {
			log.Error(err, "Failed to create new Deployment",
"Deployment.Namespace", dep.Namespace, "Deployment.Name", dep.Name)
			return ctrl.Result{}, err
		}
		// Deployment created successfully - return and requeue
		return ctrl.Result{Requeue: true}, nil
	} else if err != nil {
		log.Error(err, "Failed to get Deployment")
		return ctrl.Result{}, err
	}

	// Ensure the deployment size is the same as the spec
	size := memcached.Spec.Size
	if *found.Spec.Replicas != size {
		found.Spec.Replicas = &size
		err = r.Update(ctx, found)
		if err != nil {
			log.Error(err, "Failed to update Deployment", "Deployment.Namespace",
found.Namespace, "Deployment.Name", found.Name)
			return ctrl.Result{}, err
		}
		// Spec updated - return and requeue
		return ctrl.Result{Requeue: true}, nil
	}
```

```
        // Update the Memcached status with the pod names
        // List the pods for this memcached's deployment
        podList := &corev1.PodList{}
        listOpts := []client.ListOption{
                client.InNamespace(memcached.Namespace),
                client.MatchingLabels(labelsForMemcached(memcached.Name)),
        }
        if err = r.List(ctx, podList, listOpts...); err != nil {
                log.Error(err, "Failed to list pods", "Memcached.Namespace",
memcached.Namespace, "Memcached.Name", memcached.Name)
                return ctrl.Result{}, err
        }
        podNames := getPodNames(podList.Items)

        // Update status.Nodes if needed
        if !reflect.DeepEqual(podNames, memcached.Status.Nodes) {
                memcached.Status.Nodes = podNames
                err := r.Status().Update(ctx, memcached)
                if err != nil {
                        log.Error(err, "Failed to update Memcached status")
                        return ctrl.Result{}, err
                }
        }

        return ctrl.Result{}, nil
}

// deploymentForMemcached returns a memcached Deployment object
func (r *MemcachedReconciler) deploymentForMemcached(m *cachev1.Memcached)
*appsv1.Deployment {
        ls := labelsForMemcached(m.Name)
        replicas := m.Spec.Size

        dep := &appsv1.Deployment{
                ObjectMeta: metav1.ObjectMeta{
                        Name:      m.Name,
                        Namespace: m.Namespace,
                },
                Spec: appsv1.DeploymentSpec{
                        Replicas: &replicas,
                        Selector: &metav1.LabelSelector{
                                MatchLabels: ls,
                        },
                        Template: corev1.PodTemplateSpec{
                                ObjectMeta: metav1.ObjectMeta{
                                        Labels: ls,
                                },
                                Spec: corev1.PodSpec{
                                        Containers: []corev1.Container{{
                                                Image:   "memcached:1.4.36-alpine",
                                                Name:    "memcached",
                                                Command: []string{"memcached", "-m=64", "-o", "modern",
"-v"},
                                                Ports: []corev1.ContainerPort{{
                                                        ContainerPort: 11211,
                                                        Name:          "memcached",
```

```
                                        }},
                                    }},
                                },
                            },
                        },
                    }
                    // Set Memcached instance as the owner and controller
                    ctrl.SetControllerReference(m, dep, r.Scheme)
                    return dep
}

// labelsForMemcached returns the labels for selecting the resources
// belonging to the given memcached CR name.
func labelsForMemcached(name string) map[string]string {
                    return map[string]string{"app": "memcached", "memcached_cr": name}
}

// getPodNames returns the pod names of the array of pods passed in
func getPodNames(pods []corev1.Pod) []string {
                    var podNames []string
                    for _, pod := range pods {
                                        podNames = append(podNames, pod.Name)
                    }
                    return podNames
}

// SetupWithManager sets up the controller with the Manager.
func (r *MemcachedReconciler) SetupWithManager(mgr ctrl.Manager) error {
                    return ctrl.NewControllerManagedBy(mgr).
                                        For(&cachev1.Memcached{}).
                                        Owns(&appsv1.Deployment{}).
                                        Complete(r)
}
```

コントローラーのサンプルは、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 デプロイメントを作成します (ない場合)。
- デプロイメントのサイズが、**Memcached** CR 仕様で指定されたも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 の名前に置き換えます。

次のサブセクションでは、実装例のコントローラーがリソースを監視する方法と reconcile ループがトリガーされる方法を説明しています。これらのサブセクションを省略し、直接 Operator の実行 に進むことができます。

5.3.1.4.1. コントローラーによって監視されるリソース

controllers/memcached_controller.go の **SetupWithManager()** 関数は、CR およびコントローラーによって所有され、管理される他のリソースを監視するようにコントローラーがビルドされる方法を指定します。

```
import (
 ...
 appsv1 "k8s.io/api/apps/v1"
 ...
)

func (r *MemcachedReconciler) SetupWithManager(mgr ctrl.Manager) error {
 return ctrl.NewControllerManagedBy(mgr).
  For(&cachev1.Memcached{}).
  Owns(&appsv1.Deployment{}).
  Complete(r)
}
```

NewControllerManagedBy() は、さまざまなコントローラー設定を可能にするコントローラービルダーを提供します。

For(&cachev1.Memcached{}) は、監視するプライマリーリソースとして **Memcached** タイプを指定します。**Memcached** タイプのそれぞれの Add、Update、または Delete イベントの場合、reconcile ループに **Memcached** オブジェクトの (namespace および name キーから成る) reconcile **Request** 引数が送られます。

Owns(&appsv1.Deployment{}) は、監視するセカンダリーリソースとして **Deployment** タイプを指定します。Add、Update、または Delete イベントの各 **Deployment** タイプの場合、イベントハンドラーは各イベントを、デプロイメントのオーナーの reconcile request にマップします。この場合、デプロイメントが作成された **Memcached** オブジェクトがオーナーです。

5.3.1.4.2. コントローラーの設定

多くの他の便利な設定を使用すると、コントローラーを初期化できます。以下に例を示します。

- **MaxConcurrentReconciles** オプションを使用して、コントローラーの同時調整の最大数を設定します。デフォルトは **1** です。

```
func (r *MemcachedReconciler) SetupWithManager(mgr ctrl.Manager) error {
    return ctrl.NewControllerManagedBy(mgr).
        For(&cachev1.Memcached{}).
        Owns(&appsv1.Deployment{}).
        WithOptions(controller.Options{
            MaxConcurrentReconciles: 2,
        }).
        Complete(r)
}
```

- 述語を使用した監視イベントをフィルタリングします。
- EventHandler のタイプを選択し、監視イベントが reconcile ループの reconcile request に変換する方法を変更します。プライマリーリソースおよびセカンダリーリソースよりも複雑な Operator 関係の場合は、**EnqueueRequestsFromMapFunc** ハンドラーを使用して、監視イベントを任意の reconcile request のセットに変換することができます。

これらの設定およびその他の設定に関する詳細は、アップストリームの Builder および Controller の GoDocs を参照してください。

5.3.1.4.3. reconcile ループ

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 および name キーである **Request** 引数が渡されます。

```
import (
 ctrl "sigs.k8s.io/controller-runtime"

 cachev1 "github.com/example-inc/memcached-operator/api/v1"
 ...
)

func (r *MemcachedReconciler) Reconcile(ctx context.Context, req ctrl.Request) (ctrl.Result, error) {
  // Lookup the Memcached instance for this reconcile request
  memcached := &cachev1.Memcached{}
  err := r.Get(ctx, req.NamespacedName, memcached)
  ...
}
```

返り値、結果、およびエラーに基づいて、Request は再度キューに入れられ、reconcile 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 ctrl.Result{}, nil
// Reconcile failed due to error - requeue
return ctrl.Result{}, err
// Requeue for any reason other than an error
return ctrl.Result{Requeue: true}, nil
```

Result.RequeueAfter を設定して、猶予期間後にも要求を再びキューに入れることができます。

```
import "time"

// Reconcile for any reason other than an error after 5 seconds
return ctrl.Result{RequeueAfter: time.Second*5}, nil
```

注記

RequeueAfter を定期的な CR の調整に設定している **Result** を返すことができます。

reconciler、クライアント、およびリソースイベントとの対話に関する詳細は、Controller Runtime Client API のドキュメントを参照してください。

5.3.1.4.4. パーミッションおよび RBAC マニフェスト

コントローラーには、管理しているリソースと対話するために特定の RBAC パーミッションが必要です。これらは、以下のような RBAC マーカーを使用して指定されます。

```
//
+kubebuilder:rbac:groups=cache.example.com,resources=memcacheds,verbs=get;list;watch;create;upd
ate;patch;delete
//
+kubebuilder:rbac:groups=cache.example.com,resources=memcacheds/status,verbs=get;update;patch
```

```
// +kubebuilder:rbac:groups=cache.example.com,resources=memcacheds/finalizers,verbs=update
//
+kubebuilder:rbac:groups=apps,resources=deployments,verbs=get;list;watch;create;update;patch;delete

// +kubebuilder:rbac:groups=core,resources=pods,verbs=get;list;

func (r *MemcachedReconciler) Reconcile(ctx context.Context, req ctrl.Request) (ctrl.Result, error) {
  ...
}
```

config/rbac/role.yaml の **ClusterRole** オブジェクトマニフェストは、**make manifests** コマンドが実行されるたびに **controller-gen** ユーティリティーを使用して、以前のマーカーから生成されます。

5.3.1.5. プロキシーサポートの有効化

Operator の作成者は、ネットワークプロキシーをサポートする Operator を開発できます。**dedicated-admin** ロールを持つ管理者は、Operator Lifecycle Manager (OLM) によって処理される環境変数のプロキシーサポートを設定します。Operator は以下の標準プロキシー変数の環境を検査し、値をオペランドに渡して、プロキシーされたクラスターをサポートする必要があります。

- **HTTP_PROXY**
- **HTTPS_PROXY**
- **NO_PROXY**

注記

このチュートリアルでは、**HTTP_PROXY** を環境変数の例として使用します。

前提条件

- クラスター全体の egress プロキシーが有効にされているクラスター。

手順

1. **controllers/memcached_controller.go** ファイルを編集し、以下のパラメーターを追加します。
   a. **operator-lib** ライブラリーから **proxy** パッケージをインポートします。

```
import (
  ...
  "github.com/operator-framework/operator-lib/proxy"
)
```

   b. **proxy.ReadProxyVarsFromEnv** helper 関数を調整ループに、結果をオペランド環境に追加します。

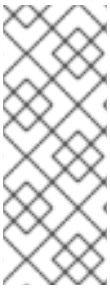
```
for i, container := range dep.Spec.Template.Spec.Containers {
  dep.Spec.Template.Spec.Containers[i].Env = append(container.Env,
proxy.ReadProxyVarsFromEnv()...)
}
...
```

2. 以下を **config/manager/manager.yaml** ファイルに追加して、Operator デプロイメントに環境変数を設定します。

```
containers:
 - args:
   - --leader-elect
   - --leader-election-id=ansible-proxy-demo
   image: controller:latest
   name: manager
   env:
     - name: "HTTP_PROXY"
       value: "http_proxy_test"
```

5.3.1.6. Operator の実行

Operator をビルドして実行するには、Operator SDK CLI を使用して Operator をバンドルし、Operator Lifecycle Manager (OLM) を使用してクラスターにデプロイします。



注記

Operator を OpenShift Dedicated ではなく OpenShift Container Platform クラスターにデプロイする場合は、次の 2 つのデプロイ方法をさらに使用できます。

- クラスター外で Go プログラムとしてローカルに実行します。
- クラスター上のデプロイメントとして実行します。

注記

Go ベースの Operator を、OLM を使用するバンドルとして実行する前に、サポート対象のイメージを使用するようにプロジェクトが更新されていることを確認してください。

関連情報

- クラスター外でローカルに実行する (OpenShift Container Platform ドキュメント)
- クラスター上でデプロイメントとして実行する (OpenShift Container Platform ドキュメント)

5.3.1.6.1. Operator のバンドルおよび Operator Lifecycle Manager を使用したデプロイ

5.3.1.6.1.1. Operator のバンドル

Operator Bundle Format は、Operator SDK および Operator Lifecycle Manager (OLM) のデフォルトパッケージ方法です。Operator SDK を使用して OLM に対して Operator を準備し、バンドルイメージをとして Operator プロジェクトをビルドしてプッシュできます。

前提条件

- 開発ワークステーションに Operator SDK CLI がインストールされている。
- OpenShift CLI (**oc**) v4 以降がインストールされている。
- Operator プロジェクトが Operator SDK を使用して初期化されている。

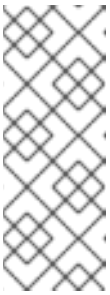
- Operator が Go ベースの場合、OpenShift Dedicated で Operator を実行するために、サポート対象のイメージを使用するようにプロジェクトが更新されている。

手順

1. 以下の **make** コマンドを Operator プロジェクトディレクトリーで実行し、Operator イメージをビルドし、プッシュします。以下の手順の **IMG** 引数を変更して、アクセス可能なリポジトリーを参照します。Quay.io などのリポジトリーサイトにコンテナーを保存するためのアカウントを取得できます。

   a. イメージをビルドします。

   ```
   $ make docker-build IMG=<registry>/<user>/<operator_image_name>:<tag>
   ```

   注記

   Operator の SDK によって生成される Dockerfile は、**go build** について **GOARCH=amd64** を明示的に参照します。これは、AMD64 アーキテクチャー以外の場合は **GOARCH=$TARGETARCH** に修正できます。Docker は、**-platform** で指定された値に環境変数を自動的に設定します。Buildah では、そのために **-build-arg** を使用する必要があります。詳細は、Multiple Architectures を参照してください。

   b. イメージをリポジトリーにプッシュします。

   ```
   $ make docker-push IMG=<registry>/<user>/<operator_image_name>:<tag>
   ```

2. Operator SDK **generate bundle** および **bundle validate** のサブコマンドを含む複数のコマンドを呼び出す **make bundle** コマンドを実行し、Operator バンドルマニフェストを作成します。

   ```
   $ make bundle IMG=<registry>/<user>/<operator_image_name>:<tag>
   ```

   Operator のバンドルマニフェストは、アプリケーションを表示し、作成し、管理する方法を説明します。**make bundle** コマンドは、以下のファイルおよびディレクトリーを Operator プロジェクトに作成します。

   - **ClusterServiceVersion** オブジェクトを含む **bundle/manifests** という名前のバンドルマニフェストディレクトリー
   - **bundle/metadata** という名前のバンドルメタデータディレクトリー
   - **config/crd** ディレクトリー内のすべてのカスタムリソース定義 (CRD)
   - Dockerfile **bundle.Dockerfile**

   続いて、これらのファイルは **operator-sdk bundle validate** を使用して自動的に検証され、ディスク上のバンドル表現が正しいことを確認します。

3. 以下のコマンドを実行し、バンドルイメージをビルドしてプッシュします。OLM は、1 つ以上のバンドルイメージを参照するインデックスイメージを使用して Operator バンドルを使用します。

   a. バンドルイメージをビルドします。イメージをプッシュしようとするレジストリー、ユーザー namespace、およびイメージタグの詳細で **BUNDLE_IMG** を設定します。

```
$ make bundle-build BUNDLE_IMG=<registry>/<user>/<bundle_image_name>:<tag>
```

b. バンドルイメージをプッシュします。

```
$ docker push <registry>/<user>/<bundle_image_name>:<tag>
```

5.3.1.6.1.2. Operator Lifecycle Manager を使用した Operator のデプロイ

Operator Lifecycle Manager (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、ライフサイクルを管理するのに役立ちます。OLM はデフォルトで OpenShift Dedicated にインストールされ、Kubernetes 拡張として実行されます。そのため、追加のツールなしで、すべての Operator ライフサイクル管理機能に Web コンソールと OpenShift CLI (**oc**) を使用できます。

Operator Bundle Format は、Operator SDK および OLM のデフォルトパッケージ方法です。Operator SDK を使用して OLM でバンドルイメージを迅速に実行し、適切に実行されるようにできます。

前提条件

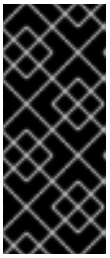
- 開発ワークステーションに Operator SDK CLI がインストールされている。
- Operator バンドルイメージがビルドされ、レジストリーにプッシュされている。
- OLM が Kubernetes ベースのクラスターにインストールされている (**apiextensions.k8s.io/v1** CRD を使用する場合は v1.16.0 以降、たとえば OpenShift Dedicated 4)
- **dedicated-admin** 権限を持つアカウントを使用して **oc** でクラスターにログインしている。
- Operator が Go ベースの場合、OpenShift Dedicated で Operator を実行するために、サポート対象のイメージを使用するようにプロジェクトが更新されている。

手順

- 以下のコマンドを入力してクラスターで Operator を実行します。

```
$ operator-sdk run bundle \ 1
  -n <namespace> \ 2
  <registry>/<user>/<bundle_image_name>:<tag> 3
```

1. **run bundle** コマンドは、有効なファイルベースのカタログを作成し、OLM を使用して Operator バンドルをクラスターにインストールします。
2. オプション: デフォルトで、このコマンドは **~/.kube/config** ファイルの現在アクティブなプロジェクトに Operator をインストールします。**-n** フラグを追加して、インストールに異なる namespace スコープを設定できます。
3. イメージを指定しない場合、コマンドは **quay.io/operator-framework/opm:latest** をデフォルトのインデックスイメージとして使用します。イメージを指定した場合は、コマンドはバンドルイメージ自体をインデックスイメージとして使用します。



重要

OpenShift Dedicated 4.11 以降、**run bundle** コマンドは Operator カタログのファイルベースのカタログ形式をデフォルトでサポートしています。Operator カタログに関して、非推奨の SQLite データベース形式は引き続きサポートされますが、今後のリリースで削除される予定です。Operator の作成者はワークフローをファイルベースのカタログ形式に移行することが推奨されます。

このコマンドにより、以下のアクションが行われます。

- バンドルイメージをインジェクトしてインデックスイメージを作成します。インデックスイメージは不透明で一時的なものですが、バンドルを実稼働環境でカタログに追加する方法を正確に反映します。
- 新規インデックスイメージを参照するカタログソースを作成します。これにより、OperatorHub が Operator を検出できるようになります。
- **OperatorGroup**、**Subscription**、**InstallPlan**、および RBAC を含むその他の必要なリソースすべてを作成して、Operator をクラスターにデプロイします。

5.3.1.7. カスタムリソースの作成

Operator のインストール後に、Operator によってクラスターに提供されるカスタムリソース (CR) を作成して、これをテストできます。

前提条件

- クラスターにインストールされている **Memcached** CR を提供する Memcached Operator の例

手順

1. Operator がインストールされている namespace へ変更します。たとえば、**make deploy** コマンドを使用して Operator をデプロイした場合は、以下のようになります。

```
$ oc project memcached-operator-system
```

2. **config/samples/cache_v1_memcached.yaml** で **Memcached** CR マニフェストのサンプルを編集し、以下の仕様が含まれるようにします。

```
apiVersion: cache.example.com/v1
kind: Memcached
metadata:
  name: memcached-sample
...
spec:
...
  size: 3
```

3. CR を作成します。

```
$ oc apply -f config/samples/cache_v1_memcached.yaml
```

4. **Memcached** Operator が、正しいサイズで CR サンプルのデプロイメントを作成することを確認します。

```
$ oc get deployments
```

出力例

```
NAME                                    READY   UP-TO-DATE   AVAILABLE   AGE
memcached-operator-controller-manager   1/1     1            1           8m
memcached-sample                        3/3     3            3           1m
```

5. 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 ステータスを確認します。

   a. Pod を確認します。

   ```
   $ oc get pods
   ```

   出力例

   ```
   NAME                                READY     STATUS    RESTARTS   AGE
   memcached-sample-6fd7c98d8-7dqdr    1/1       Running   0          1m
   memcached-sample-6fd7c98d8-g5k7v    1/1       Running   0          1m
   memcached-sample-6fd7c98d8-m7vn7    1/1       Running   0          1m
   ```

   b. CR ステータスを確認します。

   ```
   $ oc get memcached/memcached-sample -o yaml
   ```

   出力例

   ```
   apiVersion: cache.example.com/v1
   kind: Memcached
   metadata:
   ...
     name: memcached-sample
   ...
   spec:
     size: 3
   status:
     nodes:
     - memcached-sample-6fd7c98d8-7dqdr
     - memcached-sample-6fd7c98d8-g5k7v
     - memcached-sample-6fd7c98d8-m7vn7
   ```

6. デプロイメントサイズを更新します。

   a. **config/samples/cache_v1_memcached.yaml** ファイルを更新し、**Memcached** CR の **spec.size** フィールドを **3** から **5** に変更します。

   ```
   $ oc patch memcached memcached-sample \
       -p '{"spec":{"size": 5}}' \
       --type=merge
   ```

   b. Operator がデプロイメントサイズを変更することを確認します。

```
$ oc get deployments
```

出力例

```
NAME                                    READY   UP-TO-DATE   AVAILABLE   AGE
memcached-operator-controller-manager   1/1     1            1           10m
memcached-sample                        5/5     5            5           3m
```

7. 次のコマンドを実行して CR を削除します。

```
$ oc delete -f config/samples/cache_v1_memcached.yaml
```

8. このチュートリアルの一環として作成したリソースをクリーンアップします。

- Operator のテストに **make deploy** コマンドを使用した場合は、以下のコマンドを実行します。

```
$ make undeploy
```

- Operator のテストに **operator-sdk run bundle** コマンドを使用した場合は、以下のコマンドを実行します。

```
$ operator-sdk cleanup <project_name>
```

5.3.1.8. 関連情報

- Operator SDK によって作成されるディレクトリー構造の詳細は、Go ベースの Operator のプロジェクトレイアウト を参照してください。
- クラスター全体の egress プロキシーが設定されている場合、**dedicated-admin** ロールを持つ管理者は、Operator Lifecycle Manager (OLM) で実行されている特定の Operator の プロキシー設定をオーバーライドしたり、カスタム CA 証明書を挿入 したりできます。

5.3.2. Go ベースの Operator のプロジェクトレイアウト

operator-sdk CLI は、各 Operator プロジェクトに多数のパッケージおよびファイルを生成、または スキャフォールディング することができます。

5.3.2.1. Go ベースのプロジェクトレイアウト

operator-sdk init 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。

ファイルまたはディレクトリー	目的
main.go	Operator のメインプログラム。これは、**apis/** ディレクトリーのすべてのカスタムリソース定義 (CRD) を登録する新規のマネージャーをインスタンス化し、**controllers/** ディレクトリーのすべてのコントローラーを起動します。

ファイルまたはディレクトリー	目的
apis/	CRD の API を定義するディレクトリーツリー。**apis/<version>/<kind>_types.go** ファイルを編集して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、これらのリソースタイプを監視する必要があります。
controllers/	コントローラーの実装。**controller/<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 reconcile ロジックを定義します。
config/	クラスターにコントローラーをデプロイするために使用される Kubernetes マニフェスト (CRD、RBAC、および証明書を含む)。
Makefile	コントローラーのビルドおよびデプロイに使用するターゲット。
Dockerfile	コンテナーエンジンが Operator をビルドするために使用する手順。
manifests/	CRD の登録、RBAC のセットアップ、およびデプロイメントとして Operator のデプロイをする Kubernetes マニフェスト。

5.3.3. 新しい Operator SDK バージョンの Go ベースの Operator プロジェクトの更新

OpenShift Dedicated 4 は Operator SDK 1.31.0 をサポートします。ワークステーションに 1.28.0 CLI がすでにインストールされている場合は、最新バージョンをインストール して CLI を 1.31.0 に更新できます。

ただし、既存の Operator プロジェクトが Operator SDK 1.31.0 との互換性を維持するには、1.28.0 以降に導入された関連する重大な変更に対し、更新手順を実行する必要があります。アップグレードの手順は、以前は 1.28.0 で作成または維持されている Operator プロジェクトのいずれかで手動で実行する必要があります。

5.3.3.1. Operator SDK 1.31.0 の Go ベースの Operator プロジェクトの更新

次の手順では、1.31.0 との互換性を確保するため、既存の Go ベースの Operator プロジェクトを更新します。

前提条件

- Operator SDK 1.31.0 がインストールされている
- Operator SDK 1.28.0 で作成または保守されている Operator プロジェクト

手順

- 次の例に示すように、Operator プロジェクトの Makefile を編集して Operator SDK バージョンを 1.31.0 に更新します。

  例 makefile

```
# Set the Operator SDK version to use. By default, what is installed on the system is used.
# This is useful for CI or a project to utilize a specific version of the operator-sdk toolkit.
OPERATOR_SDK_VERSION ?= v1.31.0 (1)
```

(1) バージョンを **1.28.0** から **1.31.0** に変更します。

5.3.3.2. 関連情報

- パッケージマニフェストプロジェクトのバンドル形式への移行
- Operator SDK 1.16.0 のプロジェクトのアップグレード
- Operator SDK v1.10.1 のプロジェクトのアップグレード
- Operator SDK v1.8.0 のプロジェクトのアップグレード

5.4. ANSIBLE ベース OPERATOR

5.4.1. Ansible ベース Operator の Operator SDK チュートリアル

Operator 開発者は、Operator SDK での Ansible のサポートを利用して、Ansible ベースの Memcached Operator のサンプルをビルドして、分散キー/値のストアを作成し、そのライフサイクルを管理することができます。このチュートリアルでは、以下のプロセスについて説明します。

- Memcached デプロイメントを作成します。
- デプロイメントのサイズが、**Memcached** カスタムリソース (CR) 仕様で指定されたものと同じであることを確認します。
- ステータスライターを使用して、**Memcached** CR ステータスを **memcached** Pod の名前で更新します。

このプロセスは、Operator Framework の 2 つの重要な設定要素を使用して実行されます。

Operator SDK

operator-sdk CLI ツールおよび **controller-runtime** ライブラリー API

Operator Lifecycle Manager (OLM)

クラスター上の 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)

注記

このチュートリアルでは、OpenShift Container Platform ドキュメントの Ansible ベースの Operator の Operator SDK の使用を開始する よりも詳細に説明します。

5.4.1.1. 前提条件

- Operator SDK CLI がインストールされている。
- OpenShift CLI (**oc**) 4 以降がインストールされている。
- Ansible 2.15.0

- Ansible Runner 2.3.3 以降
- Ansible Runner HTTP Event Emitter プラグイン 1.0.0 以降
- Python 3.9 以降
- Python Kubernetes クライアント
- **dedicated-admin** 権限を持つアカウントを使用して、**oc** で OpenShift Dedicated クラスターにログインしている。
- クラスターがイメージをプルできるように、イメージをプッシュするリポジトリーを public として設定するか、イメージプルシークレットを設定している。

関連情報

- Operator SDK CLI のインストール
- Getting started with the OpenShift CLI

5.4.1.2. プロジェクトの作成

Operator SDK CLI を使用して **memcached-operator** というプロジェクトを作成します。

手順

1. プロジェクトのディレクトリーを作成します。

```
$ mkdir -p $HOME/projects/memcached-operator
```

2. ディレクトリーに切り替えます。

```
$ cd $HOME/projects/memcached-operator
```

3. **ansible** プラグインを指定して **operator-sdk init** コマンドを実行し、プロジェクトを初期化します。

```
$ operator-sdk init \
    --plugins=ansible \
    --domain=example.com
```

5.4.1.2.1. PROJECT ファイル

operator-sdk init コマンドで生成されるファイルの 1 つに、Kubebuilder の **PROJECT** ファイルがあります。プロジェクトルートから実行される後続の **operator-sdk** コマンドおよび **help** 出力は、このファイルを読み取り、プロジェクトタイプが Ansible であることを認識しています。以下に例を示します。

```
domain: example.com
layout:
- ansible.sdk.operatorframework.io/v1
plugins:
  manifests.sdk.operatorframework.io/v2: {}
  scorecard.sdk.operatorframework.io/v2: {}
```

```
  sdk.x-openshift.io/v1: {}
projectName: memcached-operator
version: "3"
```

5.4.1.3. API の作成

Operator SDK CLI を使用して Memcached API を作成します。

手順

- 以下のコマンドを実行して、グループ **cache**、バージョン、**v1**、および種類 **Memcached** を指定して API を作成します。

```
$ operator-sdk create api \
    --group cache \
    --version v1 \
    --kind Memcached \
    --generate-role 1
```

1 API の Ansible ロールを生成します。

API の作成後に、Operator プロジェクトは以下の構造で更新します。

Memcached CRD
サンプル **Memcached** リソースが含まれます。

Manager
以下を使用して、クラスターの状態を必要な状態に調整するプログラム。

- reconciler (Ansible ロールまたは Playbook のいずれか)
- **Memcached** リソースを **memcached** Ansible ロールに接続する **watches.yaml** ファイル

5.4.1.4. マネージャーの変更

Operator プロジェクトを更新して、Ansible ロールの形式で reconcile ロジックを提供します。これは、**Memcached** リソースが作成、更新、または削除されるたびに実行されます。

手順

1. **roles/memcached/tasks/main.yml** ファイルを以下の構造で更新します。

```
---
- name: start memcached
  k8s:
    definition:
      kind: Deployment
      apiVersion: apps/v1
      metadata:
        name: '{{ ansible_operator_meta.name }}-memcached'
        namespace: '{{ ansible_operator_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
```

この **memcached** ロールは、**memcached** デプロイメントが存在することを確実にし、デプロイメントサイズを設定します。

2. **roles/memcached/defaults/main.yml** ファイルを編集して、Ansible ロールで使用される変数のデフォルト値を設定します。

```
---
# defaults file for Memcached
size: 1
```

3. 以下の構造で、**config/samples/cache_v1_memcached.yaml** ファイルの **Memcached** サンプルリソースを更新します。

```
apiVersion: cache.example.com/v1
kind: Memcached
metadata:
  labels:
    app.kubernetes.io/name: memcached
    app.kubernetes.io/instance: memcached-sample
    app.kubernetes.io/part-of: memcached-operator
    app.kubernetes.io/managed-by: kustomize
    app.kubernetes.io/created-by: memcached-operator
  name: memcached-sample
spec:
  size: 3
```

カスタムリソース (CR) 仕様のキー/値のペアは、追加の変数として Ansible に渡されます。

注記

spec 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。

watches.yaml ファイルで **snakeCaseParameters** オプションを **false** に設定して、このケース変換を無効にすることができます。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していることを確認することが推奨されます。

5.4.1.5. プロキシーサポートの有効化

Operator の作成者は、ネットワークプロキシーをサポートする Operator を開発できます。**dedicated-admin** ロールを持つ管理者は、Operator Lifecycle Manager (OLM) によって処理される環境変数のプロキシーサポートを設定します。Operator は以下の標準プロキシー変数の環境を検査し、値をオペランドに渡して、プロキシーされたクラスターをサポートする必要があります。

- **HTTP_PROXY**
- **HTTPS_PROXY**
- **NO_PROXY**

注記

このチュートリアルでは、**HTTP_PROXY** を環境変数の例として使用します。

前提条件

- クラスター全体の egress プロキシーが有効にされているクラスター。

手順

1. 以下で **roles/memcached/tasks/main.yml** ファイルを更新して、環境変数をデプロイメントに追加します。

```
...
env:
  - name: HTTP_PROXY
    value: '{{ lookup("env", "HTTP_PROXY") | default("", True) }}'
  - name: http_proxy
    value: '{{ lookup("env", "HTTP_PROXY") | default("", True) }}'
...
```

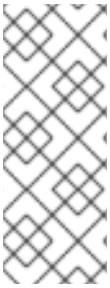
2. 以下を **config/manager/manager.yaml** ファイルに追加して、Operator デプロイメントに環境変数を設定します。

```
containers:
 - args:
   - --leader-elect
   - --leader-election-id=ansible-proxy-demo
   image: controller:latest
   name: manager
```

```
env:
  - name: "HTTP_PROXY"
    value: "http_proxy_test"
```

5.4.1.6. Operator の実行

Operator をビルドして実行するには、Operator SDK CLI を使用して Operator をバンドルし、Operator Lifecycle Manager (OLM) を使用してクラスターにデプロイします。



注記

Operator を OpenShift Dedicated ではなく OpenShift Container Platform クラスターにデプロイする場合は、次の 2 つのデプロイ方法をさらに使用できます。

- クラスター外で Go プログラムとしてローカルに実行します。
- クラスター上のデプロイメントとして実行します。

関連情報

- クラスター外でローカルに実行する (OpenShift Container Platform ドキュメント)
- クラスター上でデプロイメントとして実行する (OpenShift Container Platform ドキュメント)

5.4.1.6.1. Operator のバンドルおよび Operator Lifecycle Manager を使用したデプロイ

5.4.1.6.1.1. Operator のバンドル

Operator Bundle Format は、Operator SDK および Operator Lifecycle Manager (OLM) のデフォルトパッケージ方法です。Operator SDK を使用して OLM に対して Operator を準備し、バンドルイメージをとして Operator プロジェクトをビルドしてプッシュできます。

前提条件

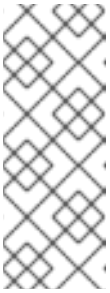
- 開発ワークステーションに Operator SDK CLI がインストールされている。
- OpenShift CLI (**oc**) v4 以降がインストールされている。
- Operator プロジェクトが Operator SDK を使用して初期化されている。

手順

1. 以下の **make** コマンドを Operator プロジェクトディレクトリーで実行し、Operator イメージをビルドし、プッシュします。以下の手順の **IMG** 引数を変更して、アクセス可能なリポジトリーを参照します。Quay.io などのリポジトリーサイトにコンテナーを保存するためのアカウントを取得できます。

   a. イメージをビルドします。

   ```
   $ make docker-build IMG=<registry>/<user>/<operator_image_name>:<tag>
   ```



注記

Operator の SDK によって生成される Dockerfile は、**go build** について **GOARCH=amd64** を明示的に参照します。これは、AMD64 アーキテクチャー以外の場合は **GOARCH=$TARGETARCH** に修正できます。Docker は、**-platform** で指定された値に環境変数を自動的に設定します。Buildah では、そのために **-build-arg** を使用する必要があります。詳細は、Multiple Architectures を参照してください。

b. イメージをリポジトリーにプッシュします。

```
$ make docker-push IMG=<registry>/<user>/<operator_image_name>:<tag>
```

2. Operator SDK **generate bundle** および **bundle validate** のサブコマンドを含む複数のコマンドを呼び出す **make bundle** コマンドを実行し、Operator バンドルマニフェストを作成します。

```
$ make bundle IMG=<registry>/<user>/<operator_image_name>:<tag>
```

Operator のバンドルマニフェストは、アプリケーションを表示し、作成し、管理する方法を説明します。**make bundle** コマンドは、以下のファイルおよびディレクトリーを Operator プロジェクトに作成します。

- **ClusterServiceVersion** オブジェクトを含む **bundle/manifests** という名前のバンドルマニフェストディレクトリー
- **bundle/metadata** という名前のバンドルメタデータディレクトリー
- **config/crd** ディレクトリー内のすべてのカスタムリソース定義 (CRD)
- Dockerfile **bundle.Dockerfile**

続いて、これらのファイルは **operator-sdk bundle validate** を使用して自動的に検証され、ディスク上のバンドル表現が正しいことを確認します。

3. 以下のコマンドを実行し、バンドルイメージをビルドしてプッシュします。OLM は、1 つ以上のバンドルイメージを参照するインデックスイメージを使用して Operator バンドルを使用します。

a. バンドルイメージをビルドします。イメージをプッシュしようとするレジストリー、ユーザー namespace、およびイメージタグの詳細で **BUNDLE_IMG** を設定します。

```
$ make bundle-build BUNDLE_IMG=<registry>/<user>/<bundle_image_name>:<tag>
```

b. バンドルイメージをプッシュします。

```
$ docker push <registry>/<user>/<bundle_image_name>:<tag>
```

5.4.1.6.1.2. Operator Lifecycle Manager を使用した Operator のデプロイ

Operator Lifecycle Manager (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、ライフサイクルを管理するのに役立ちます。OLM はデフォルトで OpenShift Dedicated にインストールされ、Kubernetes 拡張として実行されます。そのため、追加のツールなしで、すべての Operator ライフサイクル管理機能に Web コンソールと OpenShift CLI (**oc**) を使用できます。

Operator Bundle Format は、Operator SDK および OLM のデフォルトパッケージ方法です。Operator SDK を使用して OLM でバンドルイメージを迅速に実行し、適切に実行されるようにできます。

前提条件

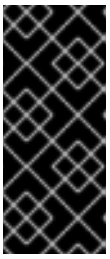
- 開発ワークステーションに Operator SDK CLI がインストールされている。
- Operator バンドルイメージがビルドされ、レジストリーにプッシュされている。
- OLM が Kubernetes ベースのクラスターにインストールされている (**apiextensions.k8s.io/v1** CRD を使用する場合は v1.16.0 以降、たとえば OpenShift Dedicated 4)
- **dedicated-admin** 権限を持つアカウントを使用して **oc** でクラスターにログインしている。

手順

- 以下のコマンドを入力してクラスターで Operator を実行します。

```
$ operator-sdk run bundle \ 1
    -n <namespace> \ 2
    <registry>/<user>/<bundle_image_name>:<tag> 3
```

1. **run bundle** コマンドは、有効なファイルベースのカタログを作成し、OLM を使用して Operator バンドルをクラスターにインストールします。
2. オプション: デフォルトで、このコマンドは **~/.kube/config** ファイルの現在アクティブなプロジェクトに Operator をインストールします。**-n** フラグを追加して、インストールに異なる namespace スコープを設定できます。
3. イメージを指定しない場合、コマンドは **quay.io/operator-framework/opm:latest** をデフォルトのインデックスイメージとして使用します。イメージを指定した場合は、コマンドはバンドルイメージ自体をインデックスイメージとして使用します。



重要

OpenShift Dedicated 4.11 以降、**run bundle** コマンドは Operator カタログのファイルベースのカタログ形式をデフォルトでサポートしています。Operator カタログに関して、非推奨の SQLite データベース形式は引き続きサポートされますが、今後のリリースで削除される予定です。Operator の作成者はワークフローをファイルベースのカタログ形式に移行することが推奨されます。

このコマンドにより、以下のアクションが行われます。

- バンドルイメージをインジェクトしてインデックスイメージを作成します。インデックスイメージは不透明で一時的なものですが、バンドルを実稼働環境でカタログに追加する方法を正確に反映します。
- 新規インデックスイメージを参照するカタログソースを作成します。これにより、OperatorHub が Operator を検出できるようになります。
- **OperatorGroup**、**Subscription**、**InstallPlan**、および RBAC を含むその他の必要なリソースすべてを作成して、Operator をクラスターにデプロイします。

5.4.1.7. カスタムリソースの作成

Operator のインストール後に、Operator によってクラスターに提供されるカスタムリソース (CR) を作成して、これをテストできます。

前提条件

- クラスターにインストールされている **Memcached** CR を提供する Memcached Operator の例

手順

1. Operator がインストールされている namespace へ変更します。たとえば、**make deploy** コマンドを使用して Operator をデプロイした場合は、以下のようになります。

```
$ oc project memcached-operator-system
```

2. **config/samples/cache_v1_memcached.yaml** で **Memcached** CR マニフェストのサンプルを編集し、以下の仕様が含まれるようにします。

```
apiVersion: cache.example.com/v1
kind: Memcached
metadata:
  name: memcached-sample
...
spec:
...
  size: 3
```

3. CR を作成します。

```
$ oc apply -f config/samples/cache_v1_memcached.yaml
```

4. **Memcached** Operator が、正しいサイズで CR サンプルのデプロイメントを作成することを確認します。

```
$ oc get deployments
```

出力例

```
NAME                                    READY   UP-TO-DATE   AVAILABLE   AGE
memcached-operator-controller-manager   1/1     1            1           8m
memcached-sample                        3/3     3            3           1m
```

5. 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 ステータスを確認します。

   a. Pod を確認します。

```
$ oc get pods
```

出力例

```
NAME                                    READY   STATUS    RESTARTS   AGE
```

```
memcached-sample-6fd7c98d8-7dqdr      1/1       Running   0          1m
memcached-sample-6fd7c98d8-g5k7v      1/1       Running   0          1m
memcached-sample-6fd7c98d8-m7vn7      1/1       Running   0          1m
```

b. CR ステータスを確認します。

```
$ oc get memcached/memcached-sample -o yaml
```

出力例

```
apiVersion: cache.example.com/v1
kind: Memcached
metadata:
...
  name: memcached-sample
...
spec:
  size: 3
status:
  nodes:
  - memcached-sample-6fd7c98d8-7dqdr
  - memcached-sample-6fd7c98d8-g5k7v
  - memcached-sample-6fd7c98d8-m7vn7
```

6. デプロイメントサイズを更新します。

a. **config/samples/cache_v1_memcached.yaml** ファイルを更新し、**Memcached** CR の **spec.size** フィールドを **3** から **5** に変更します。

```
$ oc patch memcached memcached-sample \
    -p '{"spec":{"size": 5}}' \
    --type=merge
```

b. Operator がデプロイメントサイズを変更することを確認します。

```
$ oc get deployments
```

出力例

```
NAME                                  READY   UP-TO-DATE   AVAILABLE   AGE
memcached-operator-controller-manager   1/1     1            1           10m
memcached-sample                      5/5     5            5           3m
```

7. 次のコマンドを実行して CR を削除します。

```
$ oc delete -f config/samples/cache_v1_memcached.yaml
```

8. このチュートリアルの一環として作成したリソースをクリーンアップします。

- Operator のテストに **make deploy** コマンドを使用した場合は、以下のコマンドを実行します。

```
$ make undeploy
```

- Operator のテストに **operator-sdk run bundle** コマンドを使用した場合は、以下のコマンドを実行します。

```
$ operator-sdk cleanup <project_name>
```

5.4.1.8. 関連情報

- Operator SDK によって作成されるディレクトリー構造の詳細は、Ansible ベース Operator のプロジェクトレイアウト を参照してください。
- クラスター全体の egress プロキシーが設定されている場合、**dedicated-admin** ロールを持つ管理者は、Operator Lifecycle Manager (OLM) で実行されている特定の Operator の プロキシー設定をオーバーライドしたり、カスタム CA 証明書を挿入 したりできます。

5.4.2. Ansible ベース Operator のプロジェクトレイアウト

operator-sdk CLI は、各 Operator プロジェクトに多数のパッケージおよびファイルを生成、または スキャフォールディング することができます。

5.4.2.1. Ansible ベースのプロジェクトレイアウト

operator-sdk init --plugins ansible コマンドを使用して生成される Ansible ベースの 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。

ファイルまたはディレクトリー	目的
Dockerfile	Operator のコンテナーイメージをビルドするための Dockerfile。
Makefile	Operator バイナリーをラップするコンテナーイメージのビルド、公開、デプロイに使用するターゲット、およびカスタムリソース定義 (CRD) のインストールおよびアンインストールに使用するターゲット。
PROJECT	Operator のメタデータ情報が含まれる YAML ファイル。
config/crd	ベース CRD ファイルおよび **kustomization.yaml** ファイルの設定。
config/default	デプロイメント用のすべての Operator マニフェストを収集します。**make deploy** コマンドを使用します。
config/manager	コントローラーマネージャーデプロイメント。
config/prometheus	Operator をモニタリングするための **ServiceMonitor** リソース。
config/rbac	リーダー選択および認証プロキシーのロールとロールバインディング。
config/samples	CRD 用に作成されたサンプルリソース。
config/testing	テスト用の設定例。

ファイルまたはディレクトリー	目的
playbooks/	実行する Playbook のサブディレクトリー。
roles/	実行するロールツリーのサブディレクトリー。
watches.yaml	監視するリソースの group/version/kind (GVK) および Ansible 呼び出しメソッド。新しいエントリーは、**create api** コマンドを使用して追加します。
requirements.yml	ビルド時にインストールする Ansible コレクションおよびロールの依存関係が含まれる YAML ファイル。
molecule/	ロールおよび Operator のエンドツーエンドのテストを行う Molecule シナリオ。

5.4.3. 新しい Operator SDK バージョンのプロジェクトのアップグレード

OpenShift Dedicated 4 は Operator SDK 1.31.0 をサポートします。ワークステーションに 1.28.0 CLI がすでにインストールされている場合は、最新バージョンをインストール して CLI を 1.31.0 に更新できます。

ただし、既存の Operator プロジェクトが Operator SDK 1.31.0 との互換性を維持するには、1.28.0 以降に導入された関連する重大な変更に対し、更新手順を実行する必要があります。アップグレードの手順は、以前は 1.28.0 で作成または維持されている Operator プロジェクトのいずれかで手動で実行する必要があります。

5.4.3.1. Operator SDK 1.31.0 の Ansible ベースの Operator プロジェクトの更新

次の手順では、1.31.0 との互換性を確保するため、既存の Ansible ベースの Operator プロジェクトを更新します。

前提条件

- Operator SDK 1.31.0 がインストールされている
- Operator SDK 1.28.0 で作成または保守されている Operator プロジェクト

手順

1. Operator の Dockerfile に次の変更を加えます。
   a. 次の例に示すように、**ansible-operator-2.11-preview** ベースイメージを **ansible-operator** ベースイメージに置き換え、バージョンを 1.31.0 に更新します。

      Dockerfile の例

      ```
      FROM quay.io/operator-framework/ansible-operator:v1.31.0
      ```

   b. Ansible Operator バージョン 1.30.0 の Ansible 2.15.0 への更新により、次のプリインストールされた Python モジュールが削除されました。

      - **ipaddress**

- **openshift**
- **jmespath**
- **cryptography**
- **oauthlib**

Operator がこれらの削除された Python モジュールのいずれかに依存している場合は、Dockerfile を更新して、**pip install** コマンドを使用して必要なモジュールをインストールします。

2. 次の例に示すように、Operator プロジェクトの Makefile を編集して Operator SDK バージョンを 1.31.0 に更新します。

例 makefile

```
# Set the Operator SDK version to use. By default, what is installed on the system is used.
# This is useful for CI or a project to utilize a specific version of the operator-sdk toolkit.
OPERATOR_SDK_VERSION ?= v1.31.0 (1)
```

(1) バージョンを **1.28.0** から **1.31.0** に変更します。

3. 次の例に示すように、**requirements.yaml** ファイルと **requirements.go** ファイルを更新して、**community.kubernetes** コレクションを削除し、**operator_sdk.util** コレクションをバージョン **0.5.0** に更新します。

requirements.yaml ファイルの例

```
 collections:
-  - name: community.kubernetes (1)
-    version: "2.0.1"
   - name: operator_sdk.util
-    version: "0.4.0"
+    version: "0.5.0" (2)
   - name: kubernetes.core
     version: "2.4.0"
   - name: cloud.common
```

(1) **community.kubernetes** コレクションを削除します

(2) **operator_sdk.util** コレクションをバージョン **0.5.0** に更新します。

4. 次の例に示すように、**molecule/kind/molecule.yml** および **molecule/default/molecule.yml** ファイルから **lint** フィールドのすべてのインスタンスを削除します。

```
 ---
 dependency:
   name: galaxy
 driver:
   name: delegated
-  lint: |
-    set -e
```

```
-     yamllint -d "{extends: relaxed, rules: {line-length: {max: 120}}}" .
  platforms:
    - name: cluster
      groups:
  - k8s
  provisioner:
    name: ansible
-     lint: |
-       set -e
      ansible-lint
    inventory:
      group_vars:
  all:
    namespace: ${TEST_OPERATOR_NAMESPACE:-osdk-test}
      host_vars:
  localhost:
    ansible_python_interpreter: '{{ ansible_playbook_python }}'
    config_dir: ${MOLECULE_PROJECT_DIRECTORY}/config
    samples_dir: ${MOLECULE_PROJECT_DIRECTORY}/config/samples
    operator_image: ${OPERATOR_IMAGE:-""}
    operator_pull_policy: ${OPERATOR_PULL_POLICY:-"Always"}
    kustomize: ${KUSTOMIZE_PATH:-kustomize}
    env:
      K8S_AUTH_KUBECONFIG: ${KUBECONFIG:-"~/.kube/config"}
  verifier:
    name: ansible
-     lint: |
-       set -e
-       ansible-lint
```

5.4.3.2. 関連情報

- Operator SDK v1.25.4 のプロジェクトのアップグレード
- Operator SDK v1.22.0 のプロジェクトのアップグレード
- Upgrading projects for Operator SDK v1.16.0
- Operator SDK v1.10.1 のプロジェクトのアップグレード
- Operator SDK v1.8.0 のプロジェクトのアップグレード
- パッケージマニフェストプロジェクトのバンドル形式への移行

5.4.4. Operator SDK における Ansible サポート

5.4.4.1. 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5.1 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。
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リスト。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、**operator_sdk.util.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リストは Operator の動作を変更します。

表5.2 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** のデフォルト接尾辞を適用し、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: "test1.example.com/v1alpha1"
kind: "Test1"
metadata:
  name: "example"
annotations:
  ansible.operator-sdk/reconcile-period: "30s"
```

5.4.4.2. watches.yaml ファイル

group/version/kind(GVK) は Kubernetes API の一意の識別子です。 **watches.yaml** ファイルには、その GVK 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 へのマッピングのリストが含まれます。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5.3 **watches.yaml** 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。
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 の使用は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。

watches.yaml ファイルの例

```
- version: v1alpha1 (1)
  group: test1.example.com
  kind: Test1
  role: /opt/ansible/roles/Test1

- version: v1alpha1 (2)
  group: test2.example.com
  kind: Test2
  playbook: /opt/ansible/playbook.yml

- version: v1alpha1 (3)
  group: test3.example.com
  kind: Test3
  playbook: /opt/ansible/test3.yml
  reconcilePeriod: 0
  manageStatus: false
```

(1) **Test1** の **test1** ロールへの単純なマッピングの例。

(2) **Test2** の Playbook への単純なマッピングの例。

(3) **Test3** 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。

5.4.4.2.1. 高度なオプション

高度な機能は、それらを GVK ごとに **watches.yaml** 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、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5.4 高度な **watches.yaml** ファイルのオプション

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。	**ansible.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 watches.yml 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
```

5.4.4.3. 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: {{ ansible_operator_meta.name }}, {{ ansible_operator_meta.namespace }}"
```

5.4.4.4. 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。

関連情報

- **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。

5.4.5. Kubernetes Collection for Ansible

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、Kubernetes Collection for Ansible を使用できます。この Ansible モジュールのコレクションにより、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、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、Kubernetes コレクション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コレクションをインストールし、これをテストしてから、Operator 内での使用を開始します。

5.4.5.1. Kubernetes Collection for Ansible のインストール

Kubernetes Collection for Ansible をローカルワークステーションにインストールできます。

手順

1. Ansible 2.15 以降をインストールします。

```
$ sudo dnf install ansible
```

2. Python Kubernetes クライアント パッケージをインストールします。

```
$ pip install kubernetes
```

3. 以下の方法のいずれかを使用して、Kubernetes コレクションをインストールします。
   - コレクションは、Ansible Galaxy から直接インストールできます。

```
$ ansible-galaxy collection install community.kubernetes
```

   - Operator がすでに初期化されている場合は、プロジェクトのトップレベルに **requirements.yml** ファイルがあるかもしれません。このファイルは、Operator が機能するためにインストールする必要のある Ansible 依存関係を指定します。デフォルトで、このファイルは **community.kubernetes** コレクションと **operator_sdk.util** コレクションをインストールします。これは、Operator 固有の機能のモジュールおよびプラグインを提供します。
     requirements.yml ファイルから依存モジュールをインストールするには、以下を実行します。

```
$ ansible-galaxy collection install -r requirements.yml
```

5.4.5.2. Kubernetes コレクションのローカルでのテスト

Operator 開発者は、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ことができます。

前提条件

- Ansible ベースの Operator プロジェクトを初期化し、Operator SDK を使用して、生成された Ansible ロールを持つ API を作成します。
- Kubernetes Collection for Ansible をインストールします。

手順

1. Ansible ベースの Operator プロジェクトディレクトリーで、必要な Ansible ロジックを使用して **roles/<kind>/tasks/main.yml** ファイルを変更します。**roles/<kind>/** ディレクトリーは、API の作成時に **--generate-role** フラグを使用する場合に作成されます。**<kind>** を置き換え可能なものは、API に指定した kind と一致します。
以下の例では、**state** という名前の変数の値に基づいた設定マップを作成し、削除します。

```
---
- name: set ConfigMap example-config to {{ state }}
  community.kubernetes.k8s:
    api_version: v1
    kind: ConfigMap
    name: example-config
    namespace: <operator_namespace> (1)
    state: "{{ state }}"
  ignore_errors: true (2)
```

(1) config map を作成する namespace を指定します。

(2) **ignore_errors: true** を設定することにより、存在しない設定マップを削除しても失敗しません。

2. デフォルトで **state** を **present** に設定するように、**roles/<kind>/defaults/main.yml** ファイルを変更します。

```
---
state: present
```

3. プロジェクトディレクトリーのトップレベルに **playbook.yml** ファイルを作成して Ansible playbook を作成し、**<kind>** ロールを追加します。

```
---
- hosts: localhost
  roles:
    - <kind>
```

4. Playbook を実行します。

```
$ ansible-playbook playbook.yml
```

出力例

```
[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 [memcached : set ConfigMap example-config to present]
*********************************************************************************
changed: [localhost]
```

```
PLAY RECAP *********************************************************************************
localhost                  : ok=2    changed=1    unreachable=0    failed=0    skipped=0
rescued=0    ignored=0
```

5. 設定マップが作成されたことを確認します。

```
$ oc get configmaps
```

出力例

```
NAME             DATA   AGE
example-config    0      2m1s
```

6. **state** を **absent** に設定して Playbook を再実行します。

```
$ ansible-playbook playbook.yml --extra-vars state=absent
```

出力例

```
[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 [memcached : set ConfigMap example-config to absent]
**********************************************************************************
changed: [localhost]

PLAY RECAP *********************************************************************************
localhost                  : ok=2    changed=1    unreachable=0    failed=0    skipped=0
rescued=0    ignored=0
```

7. 設定マップが削除されたことを確認します。

```
$ oc get configmaps
```

5.4.5.3. 次のステップ

- カスタムリソース (CR) の変更時に、Operator 内でカスタム Ansible ロジックをトリガーする方法については、Operator 内での Ansible の使用 参照してください。

5.4.6. Operator 内での Ansible の使用

Kubernetes Collection for Ansible 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 **watches.yaml** ファイルで実行されます。

5.4.6.1. 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5.5 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。
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リスト。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、**operator_sdk.util.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リストは Operator の動作を変更します。

表5.6 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** のデフォルト接尾辞を適用し、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: "test1.example.com/v1alpha1"
kind: "Test1"
metadata:
  name: "example"
annotations:
  ansible.operator-sdk/reconcile-period: "30s"
```

5.4.6.2. Ansible ベース Operator のローカルでのテスト

Operator プロジェクトのトップレベルディレクトリーから **make run** コマンドを使用して、ローカル

で実行中の Ansible ベースの Operator 内でロジックをテストできます。**make run** Makefile ターゲットは、**ansible-operator** バイナリーをローカルで実行します。これは **watches.yaml** ファイルを読み取り、~/**.kube**/**config** ファイルを使用して **k8s** 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。

注記

環境変数 **ANSIBLE_ROLES_PATH** を設定するか、**ansible-roles-path** フラグを使用して、ロールパスをカスタマイズすることができます。ロールが **ANSIBLE_ROLES_PATH** の値にない場合、Operator は **{{current directory}}/roles** で検索します。

前提条件

- Ansible Runner v2.3.3 以降
- Ansible Runner HTTP Event Emitter プラグイン v1.0.0+
- Kubernetes コレクションをローカルでテストするための前述の手順を実施済みである。

手順

1. カスタムリソース定義 (CRD) およびカスタムリソース (CR) の適切なロールベースアクセス制御 (RBAC) 定義をインストールします。

```
$ make install
```

出力例

```
/usr/bin/kustomize build config/crd | kubectl apply -f -
customresourcedefinition.apiextensions.k8s.io/memcacheds.cache.example.com created
```

2. **make run** コマンドを実行します。

```
$ make run
```

出力例

```
/home/user/memcached-operator/bin/ansible-operator run
{"level":"info","ts":1612739145.2871568,"logger":"cmd","msg":"Version","Go
Version":"go1.15.5","GOOS":"linux","GOARCH":"amd64","ansible-
operator":"v1.10.1","commit":"1abf57985b43bf6a59dcd18147b3c574fa57d3f6"}
...
{"level":"info","ts":1612739148.347306,"logger":"controller-runtime.metrics","msg":"metrics
server is starting to listen","addr":":8080"}
{"level":"info","ts":1612739148.3488882,"logger":"watches","msg":"Environment variable not
set; using default
value","envVar":"ANSIBLE_VERBOSITY_MEMCACHED_CACHE_EXAMPLE_COM","default":
2}
{"level":"info","ts":1612739148.3490262,"logger":"cmd","msg":"Environment variable not set;
using default
value","Namespace":"","envVar":"ANSIBLE_DEBUG_LOGS","ANSIBLE_DEBUG_LOGS":fals
e}
{"level":"info","ts":1612739148.3490646,"logger":"ansible-controller","msg":"Watching
```

```
resource","Options.Group":"cache.example.com","Options.Version":"v1","Options.Kind":"Memc
ached"}
{"level":"info","ts":1612739148.350217,"logger":"proxy","msg":"Starting to
serve","Address":"127.0.0.1:8888"}
{"level":"info","ts":1612739148.3506632,"logger":"controller-runtime.manager","msg":"starting
metrics server","path":"/metrics"}
{"level":"info","ts":1612739148.350784,"logger":"controller-
runtime.manager.controller.memcached-controller","msg":"Starting
EventSource","source":"kind source: cache.example.com/v1, Kind=Memcached"}
{"level":"info","ts":1612739148.5511978,"logger":"controller-
runtime.manager.controller.memcached-controller","msg":"Starting Controller"}
{"level":"info","ts":1612739148.5512562,"logger":"controller-
runtime.manager.controller.memcached-controller","msg":"Starting workers","worker
count":8}
```

Operator が CR のイベントを監視していることから、CR の作成により、Ansible ロールの実行がトリガーされます。

注記

config/samples/<gvk>.yaml CR マニフェストの例を見てみましょう。

```
apiVersion: <group>.example.com/v1alpha1
kind: <kind>
metadata:
  name: "<kind>-sample"
```

spec フィールドが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。CR から Ansible へ追加の変数を渡すことについては、別のセクションで説明します。Operator に妥当なデフォルトを設定することは重要です。

3. デフォルト変数 **state** を **present** に設定し、CR インスタンスを作成します。

```
$ oc apply -f config/samples/<gvk>.yaml
```

4. **example-config** 設定マップが作成されたことを確認します。

```
$ oc get configmaps
```

出力例

```
NAME                 STATUS    AGE
example-config         Active    3s
```

5. **state** フィールドを **absent** に設定するように、**config/samples/<gvk>.yaml** ファイルを変更します。以下に例を示します。

```
apiVersion: cache.example.com/v1
kind: Memcached
metadata:
  name: memcached-sample
spec:
  state: absent
```

6. 変更を適用します。

```
$ oc apply -f config/samples/<gvk>.yaml
```

7. 設定マップが削除されていることを確認します。

```
$ oc get configmap
```

5.4.6.3. クラスター上での Ansible ベース Operator のテスト

Operator 内でカスタム Ansible ロジックをローカルにテストした後、OpenShift Dedicated クラスターの Pod 内で Operator をテストできます。実稼働環境での使用を目的としている場合、このテストが推奨されます。

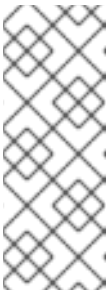
Operator プロジェクトは、クラスター上でデプロイメントとして実行できます。

手順

1. 以下の **make** コマンドを実行して Operator イメージをビルドし、プッシュします。以下の手順の **IMG** 引数を変更して、アクセス可能なリポジトリーを参照します。Quay.io などのリポジトリーサイトにコンテナーを保存するためのアカウントを取得できます。

   a. イメージをビルドします。

```
$ make docker-build IMG=<registry>/<user>/<image_name>:<tag>
```



注記

Operator の SDK によって生成される Dockerfile は、**go build** について **GOARCH=amd64** を明示的に参照します。これは、AMD64 アーキテクチャー以外の場合は **GOARCH=$TARGETARCH** に修正できます。Docker は、**-platform** で指定された値に環境変数を自動的に設定します。Buildah では、そのために **-build-arg** を使用する必要があります。詳細は、Multiple Architectures を参照してください。

   b. イメージをリポジトリーにプッシュします。

```
$ make docker-push IMG=<registry>/<user>/<image_name>:<tag>
```

注記

両方のコマンドのイメージの名前とタグ (例: **IMG=<registry>/<user>/<image_name>:<tag>**) を Makefile に設定することもできます。**IMG ?= controller:latest** の値を変更して、デフォルトのイメージ名を設定します。

2. 以下のコマンドを実行して Operator をデプロイします。

```
$ make deploy IMG=<registry>/<user>/<image_name>:<tag>
```

デフォルトで、このコマンドは **<project_name>-system** の形式で Operator プロジェクトの名前で namespace を作成し、デプロイメントに使用します。このコマンドは、**config/rbac** から RBAC マニフェストもインストールします。

3. 以下のコマンドを実行して、Operator が実行されていることを確認します。

```
$ oc get deployment -n <project_name>-system
```

出力例

```
NAME                                    READY   UP-TO-DATE   AVAILABLE   AGE
<project_name>-controller-manager       1/1     1            1           8m
```

5.4.6.4. Ansible ログ

Ansible ベースの Operator は、Ansible の実行に関するログを提供します。これは、Ansible タスクのデバッグに役立ちます。ログには、Operator の内部および Kubernetes との対話に関する詳細情報を含めることもできます。

5.4.6.4.1. Ansible ログの表示

前提条件

- Ansible ベースの Operator が、デプロイメントとしてクラスター上で実行されている。

手順

- Ansible ベースの Operator からログを表示するには、以下のコマンドを実行します。

```
$ oc logs deployment/<project_name>-controller-manager \
    -c manager \ 1
    -n <namespace> 2
```

1. **manager** コンテナーのログを表示します。
2. **make deploy** コマンドを使用して Operator をデプロイメントとして実行している場合は、**<project_name>-system** namespace を使用します。

出力例

```
{"level":"info","ts":1612732105.0579333,"logger":"cmd","msg":"Version","Go
Version":"go1.15.5","GOOS":"linux","GOARCH":"amd64","ansible-
operator":"v1.10.1","commit":"1abf57985b43bf6a59dcd18147b3c574fa57d3f6"}
{"level":"info","ts":1612732105.0587437,"logger":"cmd","msg":"WATCH_NAMESPACE
environment variable not set. Watching all namespaces.","Namespace":""}
I0207 21:08:26.110949       7 request.go:645] Throttling request took 1.035521578s, request:
GET:https://172.30.0.1:443/apis/flowcontrol.apiserver.k8s.io/v1alpha1?timeout=32s
{"level":"info","ts":1612732107.768025,"logger":"controller-runtime.metrics","msg":"metrics
server is starting to listen","addr":"127.0.0.1:8080"}
{"level":"info","ts":1612732107.768796,"logger":"watches","msg":"Environment variable not
set; using default
value","envVar":"ANSIBLE_VERBOSITY_MEMCACHED_CACHE_EXAMPLE_COM","default":
```

```
2}
{"level":"info","ts":1612732107.7688773,"logger":"cmd","msg":"Environment variable not set;
using default
value","Namespace":"","envVar":"ANSIBLE_DEBUG_LOGS","ANSIBLE_DEBUG_LOGS":fals
e}
{"level":"info","ts":1612732107.7688901,"logger":"ansible-controller","msg":"Watching
resource","Options.Group":"cache.example.com","Options.Version":"v1","Options.Kind":"Memc
ached"}
{"level":"info","ts":1612732107.770032,"logger":"proxy","msg":"Starting to
serve","Address":"127.0.0.1:8888"}
I0207 21:08:27.770185       7 leaderelection.go:243] attempting to acquire leader lease
memcached-operator-system/memcached-operator...
{"level":"info","ts":1612732107.770202,"logger":"controller-runtime.manager","msg":"starting
metrics server","path":"/metrics"}
I0207 21:08:27.784854       7 leaderelection.go:253] successfully acquired lease
memcached-operator-system/memcached-operator
{"level":"info","ts":1612732107.7850506,"logger":"controller-
runtime.manager.controller.memcached-controller","msg":"Starting
EventSource","source":"kind source: cache.example.com/v1, Kind=Memcached"}
{"level":"info","ts":1612732107.8853772,"logger":"controller-
runtime.manager.controller.memcached-controller","msg":"Starting Controller"}
{"level":"info","ts":1612732107.8854098,"logger":"controller-
runtime.manager.controller.memcached-controller","msg":"Starting workers","worker
count":4}
```

5.4.6.4.2. ログでの Ansible のすべての結果の有効化

環境変数 **ANSIBLE_DEBUG_LOGS** を **True** に設定すると、Ansible のすべての結果をログで確認できるようになります。これはデバッグの際に役立ちます。

手順

- **config/manager/manager.yaml** ファイルおよび **config/default/manager_auth_proxy_patch.yaml** ファイルを編集し、以下の設定を追加します。

```
containers:
- name: manager
  env:
  - name: ANSIBLE_DEBUG_LOGS
    value: "True"
```

5.4.6.4.3. ログでの詳細デバッグの有効化

Ansible ベースの Operator の開発中は、ログでの追加のデバッグの有効化が役立つ場合があります。

手順

- **ansible.sdk.operatorframework.io/verbosity** アノテーションをカスタムリソースに追加して、必要な詳細レベルを有効にします。以下に例を示します。

```
apiVersion: "cache.example.com/v1alpha1"
kind: "Memcached"
metadata:
```

```
    name: "example-memcached"
    annotations:
      "ansible.sdk.operatorframework.io/verbosity": "4"
  spec:
    size: 4
```

5.4.7. カスタムリソースのステータス管理

5.4.7.1. Ansible ベースの Operator でのカスタムリソースのステータスについて

Ansible ベースの Operator は、以前の Ansible 実行に関する一般的な情報を使用して、カスタムリソース (CR) ステータス サブリソース を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
```

さらに Ansible ベースの Operator は、Operator の作成者が **operator_sdk.util** コレクション に含まれる **k8s_status** Ansible モジュールでカスタムのステータス値を指定できるようにします。これにより、作成者は必要に応じ、任意のキー/値のペアを使用して 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は、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。

5.4.7.2. カスタムリソースステータスの手動による追跡

operator_sdk.util コレクションを使用して Ansible ベースの Operator を変更し、アプリケーションからカスタムリソース (CR) ステータスを手動で追跡できます。

前提条件

- Operator SDK を使用して Ansible ベースの Operator プロジェクトが作成済みである。

手順

1. **manageStatus** フィールドを **false** に設定して **watches.yaml** ファイルを更新します。

```
- version: v1
  group: api.example.com
  kind: <kind>
  role: <role>
  manageStatus: false
```

2. **operator_sdk.util.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 **test** および値 **data** を使用して更新するには、**operator_sdk.util** を以下のように使用することができます。

```
- operator_sdk.util.k8s_status:
    api_version: app.example.com/v1
    kind: <kind>
    name: "{{ ansible_operator_meta.name }}"
    namespace: "{{ ansible_operator_meta.namespace }}"
    status:
      test: data
```

3. スキャフォールディングされた Ansible ベースの Operator に含まれるロールの **meta/main.yml** ファイルで、コレクションを宣言することができます。

```
collections:
  - operator_sdk.util
```

4. ロールのメタでコレクションを宣言すると、**k8s_status** モジュールを直接起動することができます。

```
k8s_status:
  ...
  status:
    key1: value1
```

5.5. HELM ベースの OPERATOR

5.5.1. Helm ベースの Operator の Operator SDK チュートリアル

Operator 開発者は、Operator SDK での Helm のサポートを利用して、Helm ベースの Nginx Operator のサンプルをビルドし、そのライフサイクルを管理することができます。このチュートリアルでは、以下のプロセスについて説明します。

- Nginx デプロイメントの作成
- デプロイメントのサイズが、**Nginx** カスタムリソース (CR) 仕様で指定されたものと同じであることを確認します。
- ステータスライターを使用して、**Nginx** CR ステータスを **nginx** Pod の名前で更新します。

このプロセスは、Operator Framework の 2 つの重要な設定要素を使用して実行されます。

Operator SDK

operator-sdk CLI ツールおよび **controller-runtime** ライブラリー API

Operator Lifecycle Manager (OLM)

クラスター上の 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)

注記

このチュートリアルでは、OpenShift Container Platform ドキュメントの Helm ベースの Operator の Operator SDK の使用を開始する よりも詳細に説明します。

5.5.1.1. 前提条件

- Operator SDK CLI がインストールされている。
- OpenShift CLI (**oc**) 4 以降がインストールされている。
- **dedicated-admin** 権限を持つアカウントを使用して、**oc** で OpenShift Dedicated クラスターにログインしている。
- クラスターがイメージをプルできるように、イメージをプッシュするリポジトリーを public として設定するか、イメージプルシークレットを設定している。

関連情報

- Operator SDK CLI のインストール
- Getting started with the OpenShift CLI

5.5.1.2. プロジェクトの作成

Operator SDK CLI を使用して **nginx-operator** というプロジェクトを作成します。

手順

1. プロジェクトのディレクトリーを作成します。

```
$ mkdir -p $HOME/projects/nginx-operator
```

2. ディレクトリーに切り替えます。

```
$ cd $HOME/projects/nginx-operator
```

3. **helm** プラグインを指定して **operator-sdk init** コマンドを実行し、プロジェクトを初期化します。

```
$ operator-sdk init \
    --plugins=helm \
    --domain=example.com \
    --group=demo \
    --version=v1 \
    --kind=Nginx
```

注記

デフォルトで、**helm** プラグインは、ボイラープレート Helm チャートを使用してプロジェクトを初期化します。**--helm-chart** フラグなどの追加のフラグを使用すると、既存の Helm チャートを使用してプロジェクトを初期化できます。

init コマンドは、API バージョン **example.com/v1** および Kind **Nginx** でのリソースの監視に特化した **nginx-operator** プロジェクトを作成します。

4. Helm ベースのプロジェクトの場合、**init** コマンドは、チャートのデフォルトマニフェストによってデプロイされるリソースに基づいて **config/rbac/role.yaml** ファイルに RBAC ルールを生成します。このファイルで生成されるルールが Operator のパーミッション要件を満たしていることを確認します。

5.5.1.2.1. 既存の Helm チャート

ボイラープレート Helm チャートでプロジェクトを作成する代わりに、以下のフラグを使用してローカルファイルシステムまたはリモートチャートリポジトリーから既存のチャートを使用することもできます。

- **--helm-chart**
- **--helm-chart-repo**
- **--helm-chart-version**

--helm-chart フラグを指定すると、**--group**、**--version**、および **--kind** フラグは任意となります。未設定のままにすると、以下のデフォルト値が使用されます。

フラグ	値
--domain	**my.domain**
--group	**charts**
--version	**v1**
--kind	指定されたチャートからの推定値。

--helm-chart フラグがローカルチャートアーカイブ (例: **example-chart-1.2.0.tgz**) またはディレクトリーを指定する場合、チャートは検証され、プロジェクトにデプロイメントされるかコピーされます。そうでない場合は、Operator SDK はリモートリポジトリーからチャートの取得を試みます。

--helm-chart-repo フラグでカスタムリポジトリーの URL が指定されない場合には、以下のチャート参照形式がサポートされます。

フォーマット	説明

フォーマット	説明
<repo_name>/<chart_name>	**$HELM_HOME/repositories/repositories.yaml** ファイルで指定されるように、**<repo_name>** という名前の Helm チャートリポジトリーから、**<chart_name>** という名前の Helm チャートを取得します。**helm repo add** コマンドを使用して、このファイルを設定します。
<url>	指定された URL で Helm チャートアーカイブを取得します。

カスタムリポジトリーの URL が **--helm-chart-repo** によって指定される場合、以下のチャート参照形式がサポートされます。

フォーマット	説明
<chart_name>	**--helm-chart-repo** URL の値で指定された Helm チャートリポジトリーで、**<chart_name>** という名前の Helm チャートを取得します。

--helm-chart-version フラグが設定されていない場合は、Operator SDK は Helm チャートの利用可能な最新バージョンを取得します。フラグが設定されている場合は、指定したバージョンを取得します。**--helm-chart** フラグで指定したチャートが特定のバージョンを参照する場合 (例: ローカルパスまたは URL の場合)、オプションの **--helm-chart-version** フラグは使用されません。

詳細と例を確認するには、以下のコマンドを実行します。

```
$ operator-sdk init --plugins helm --help
```

5.5.1.2.2. PROJECT ファイル

operator-sdk init コマンドで生成されるファイルの 1 つに、Kubebuilder の **PROJECT** ファイルがあります。プロジェクトルートから実行される後続の **operator-sdk** コマンドおよび **help** 出力は、このファイルを読み取り、プロジェクトタイプが Helm であることを認識しています。以下に例を示します。

```
domain: example.com
layout:
- helm.sdk.operatorframework.io/v1
plugins:
  manifests.sdk.operatorframework.io/v2: {}
  scorecard.sdk.operatorframework.io/v2: {}
  sdk.x-openshift.io/v1: {}
projectName: nginx-operator
resources:
- api:
    crdVersion: v1
    namespaced: true
  domain: example.com
  group: demo
  kind: Nginx
  version: v1
version: "3"
```

5.5.1.3. Operator ロジックについて

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)。
- デプロイメント、サービス、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** プロジェクトは、**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。

```
# Use the 'create api' subcommand to add watches to this file.
- group: demo
  version: v1
  kind: Nginx
  chart: helm-charts/nginx
# +kubebuilder:scaffold:watch
```

5.5.1.3.1. Helm チャートのサンプル

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、デプロイメント、サービス、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 開発者用のドキュメント を参照してください。

5.5.1.3.2. カスタムリソース仕様の変更

Helm は 値 (value) という概念を使用して、**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。

カスタムリソース (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することができます。例としてレプリカ数を使用することができます。

手順

1. **helm-charts/nginx/values.yaml** ファイルには、デフォルトで **replicaCount** という名前の値が **1** に設定されています。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。
   config/samples/demo_v1_nginx.yaml ファイルを編集し、**replicaCount: 2** を設定します。

```
apiVersion: demo.example.com/v1
kind: Nginx
metadata:
  name: nginx-sample
...
```

```
spec:
...
  replicaCount: 2
```

2. 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** を使用するには、**config/samples/demo_v1_nginx.yaml** ファイルを編集し、**spec.port: 8080** を設定します。これにより、サービスポートの上書きが追加されます。

```
apiVersion: demo.example.com/v1
kind: Nginx
metadata:
  name: nginx-sample
spec:
  replicaCount: 2
  service:
    port: 8080
```

Helm Operator は、**helm install -f ./overrides.yaml** コマンドのように、仕様全体を values ファイルの内容のように適用します。

5.5.1.4. プロキシーサポートの有効化

Operator の作成者は、ネットワークプロキシーをサポートする Operator を開発できます。**dedicated-admin** ロールを持つ管理者は、Operator Lifecycle Manager (OLM) によって処理される環境変数のプロキシーサポートを設定します。Operator は以下の標準プロキシー変数の環境を検査し、値をオペランドに渡して、プロキシーされたクラスターをサポートする必要があります。

- **HTTP_PROXY**
- **HTTPS_PROXY**
- **NO_PROXY**

注記

このチュートリアルでは、**HTTP_PROXY** を環境変数の例として使用します。

前提条件

- クラスター全体の egress プロキシーが有効にされているクラスター。

手順

1. **watches.yaml** ファイルを編集し、**overrideValues** フィールドを追加して、環境変数に基づいてオーバーライドを含めます。

```
...
- group: demo.example.com
  version: v1alpha1
  kind: Nginx
  chart: helm-charts/nginx
  overrideValues:
    proxy.http: $HTTP_PROXY
...
```

2. **helm-charts/nginx/values.yaml** ファイルに **proxy.http** 値を追加します。

```
...
proxy:
  http: ""
  https: ""
  no_proxy: ""
```

3. チャートテンプレートで変数の使用がサポートされているようにするには、**helm-charts/nginx/templates/deployment.yaml** ファイルのチャートテンプレートを編集して以下を追加します。

```
containers:
  - name: {{ .Chart.Name }}
    securityContext:
      - toYaml {{ .Values.securityContext | nindent 12 }}
    image: "{{ .Values.image.repository }}:{{ .Values.image.tag | default .Chart.AppVersion }}"
    imagePullPolicy: {{ .Values.image.pullPolicy }}
    env:
      - name: http_proxy
        value: "{{ .Values.proxy.http }}"
```

4. 以下を **config/manager/manager.yaml** ファイルに追加して、Operator デプロイメントに環境変数を設定します。

```
containers:
 - args:
   - --leader-elect
   - --leader-election-id=ansible-proxy-demo
   image: controller:latest
   name: manager
   env:
     - name: "HTTP_PROXY"
       value: "http_proxy_test"
```

5.5.1.5. Operator の実行

Operator をビルドして実行するには、Operator SDK CLI を使用して Operator をバンドルし、Operator Lifecycle Manager (OLM) を使用してクラスターにデプロイします。

注記

Operator を OpenShift Dedicated ではなく OpenShift Container Platform クラスターにデプロイする場合は、次の 2 つのデプロイ方法をさらに使用できます。

- クラスター外で Go プログラムとしてローカルに実行します。
- クラスター上のデプロイメントとして実行します。

関連情報

- クラスター外でローカルに実行する (OpenShift Container Platform ドキュメント)
- クラスター上でデプロイメントとして実行する (OpenShift Container Platform ドキュメント)

5.5.1.5.1. Operator のバンドルおよび Operator Lifecycle Manager を使用したデプロイ

5.5.1.5.1.1. Operator のバンドル

Operator Bundle Format は、Operator SDK および Operator Lifecycle Manager (OLM) のデフォルトパッケージ方法です。Operator SDK を使用して OLM に対して Operator を準備し、バンドルイメージをとして Operator プロジェクトをビルドしてプッシュできます。

前提条件

- 開発ワークステーションに Operator SDK CLI がインストールされている。
- OpenShift CLI (**oc**) v4 以降がインストールされている。
- Operator プロジェクトが Operator SDK を使用して初期化されている。

手順

1. 以下の **make** コマンドを Operator プロジェクトディレクトリーで実行し、Operator イメージをビルドし、プッシュします。以下の手順の **IMG** 引数を変更して、アクセス可能なリポジトリーを参照します。Quay.io などのリポジトリーサイトにコンテナーを保存するためのアカウントを取得できます。

   a. イメージをビルドします。

   ```
   $ make docker-build IMG=<registry>/<user>/<operator_image_name>:<tag>
   ```

   注記

   Operator の SDK によって生成される Dockerfile は、**go build** について **GOARCH=amd64** を明示的に参照します。これは、AMD64 アーキテクチャー以外の場合は **GOARCH=$TARGETARCH** に修正できます。Docker は、**-platform** で指定された値に環境変数を自動的に設定します。Buildah では、そのために **-build-arg** を使用する必要があります。詳細は、Multiple Architectures を参照してください。

   b. イメージをリポジトリーにプッシュします。

   ```
   $ make docker-push IMG=<registry>/<user>/<operator_image_name>:<tag>
   ```

2. Operator SDK **generate bundle** および **bundle validate** のサブコマンドを含む複数のコマンドを呼び出す **make bundle** コマンドを実行し、Operator バンドルマニフェストを作成します。

   ```
   $ make bundle IMG=<registry>/<user>/<operator_image_name>:<tag>
   ```

   Operator のバンドルマニフェストは、アプリケーションを表示し、作成し、管理する方法を説明します。**make bundle** コマンドは、以下のファイルおよびディレクトリーを Operator プロジェクトに作成します。

   - **ClusterServiceVersion** オブジェクトを含む **bundle/manifests** という名前のバンドルマニフェストディレクトリー
   - **bundle/metadata** という名前のバンドルメタデータディレクトリー

- **config/crd** ディレクトリー内のすべてのカスタムリソース定義 (CRD)
- Dockerfile **bundle.Dockerfile**

続いて、これらのファイルは **operator-sdk bundle validate** を使用して自動的に検証され、ディスク上のバンドル表現が正しいことを確認します。

3. 以下のコマンドを実行し、バンドルイメージをビルドしてプッシュします。OLM は、1 つ以上のバンドルイメージを参照するインデックスイメージを使用して Operator バンドルを使用します。

   a. バンドルイメージをビルドします。イメージをプッシュしようとするレジストリー、ユーザー namespace、およびイメージタグの詳細で **BUNDLE_IMG** を設定します。

   ```
   $ make bundle-build BUNDLE_IMG=<registry>/<user>/<bundle_image_name>:<tag>
   ```

   b. バンドルイメージをプッシュします。

   ```
   $ docker push <registry>/<user>/<bundle_image_name>:<tag>
   ```

5.5.1.5.1.2. Operator Lifecycle Manager を使用した Operator のデプロイ

Operator Lifecycle Manager (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、ライフサイクルを管理するのに役立ちます。OLM はデフォルトで OpenShift Dedicated にインストールされ、Kubernetes 拡張として実行されます。そのため、追加のツールなしで、すべての Operator ライフサイクル管理機能に Web コンソールと OpenShift CLI (**oc**) を使用できます。

Operator Bundle Format は、Operator SDK および OLM のデフォルトパッケージ方法です。Operator SDK を使用して OLM でバンドルイメージを迅速に実行し、適切に実行されるようにできます。

前提条件

- 開発ワークステーションに Operator SDK CLI がインストールされている。
- Operator バンドルイメージがビルドされ、レジストリーにプッシュされている。
- OLM が Kubernetes ベースのクラスターにインストールされている (**apiextensions.k8s.io/v1** CRD を使用する場合は v1.16.0 以降、たとえば OpenShift Dedicated 4)
- **dedicated-admin** 権限を持つアカウントを使用して **oc** でクラスターにログインしている。

手順

- 以下のコマンドを入力してクラスターで Operator を実行します。

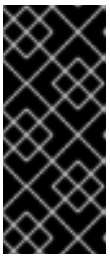
```
$ operator-sdk run bundle \ 1
  -n <namespace> \ 2
  <registry>/<user>/<bundle_image_name>:<tag> 3
```

1. **run bundle** コマンドは、有効なファイルベースのカタログを作成し、OLM を使用して Operator バンドルをクラスターにインストールします。

2.

オプション: デフォルトで、このコマンドは ~/**.kube/config** ファイルの現在アクティブなプロジェクトに Operator をインストールします。**-n** フラグを追加して、インストールに

3 イメージを指定しない場合、コマンドは **quay.io/operator-framework/opm:latest** をデフォルトのインデックスイメージとして使用します。イメージを指定した場合は、コマンドはバンドルイメージ自体をインデックスイメージとして使用します。



重要

OpenShift Dedicated 4.11 以降、**run bundle** コマンドは Operator カタログのファイルベースのカタログ形式をデフォルトでサポートしています。Operator カタログに関して、非推奨の SQLite データベース形式は引き続きサポートされますが、今後のリリースで削除される予定です。Operator の作成者はワークフローをファイルベースのカタログ形式に移行することが推奨されます。

このコマンドにより、以下のアクションが行われます。

- バンドルイメージをインジェクトしてインデックスイメージを作成します。インデックスイメージは不透明で一時的なものですが、バンドルを実稼働環境でカタログに追加する方法を正確に反映します。
- 新規インデックスイメージを参照するカタログソースを作成します。これにより、OperatorHub が Operator を検出できるようになります。
- **OperatorGroup**、**Subscription**、**InstallPlan**、および RBAC を含むその他の必要なリソースすべてを作成して、Operator をクラスターにデプロイします。

5.5.1.6. カスタムリソースの作成

Operator のインストール後に、Operator によってクラスターに提供されるカスタムリソース (CR) を作成して、これをテストできます。

前提条件

- クラスターにインストールされている **Nginx** CR を提供する Nginx Operator の例

手順

1. Operator がインストールされている namespace へ変更します。たとえば、**make deploy** コマンドを使用して Operator をデプロイした場合は、以下のようになります。

```
$ oc project nginx-operator-system
```

2. **config/samples/demo_v1_nginx.yaml** で **Nginx** CR マニフェストのサンプルを編集し、以下の仕様が含まれるようにします。

```
apiVersion: demo.example.com/v1
kind: Nginx
metadata:
  name: nginx-sample
...
spec:
...
  replicaCount: 3
```

3. Nginx サービスアカウントを OpenShift Dedicated で実行するには、特権アクセスが必要です。以下の SCC(Security Context Constraints) を **nginx-sample** Pod のサービスアカウントに追加します。

```
$ oc adm policy add-scc-to-user \
    anyuid system:serviceaccount:nginx-operator-system:nginx-sample
```

4. CR を作成します。

```
$ oc apply -f config/samples/demo_v1_nginx.yaml
```

5. **Nginx** Operator が、正しいサイズで CR サンプルのデプロイメントを作成することを確認します。

```
$ oc get deployments
```

出力例

```
NAME                                  READY   UP-TO-DATE   AVAILABLE   AGE
nginx-operator-controller-manager     1/1     1            1           8m
nginx-sample                          3/3     3            3           1m
```

6. ステータスが Nginx 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 ステータスを確認します。

   a. Pod を確認します。

```
$ oc get pods
```

出力例

```
NAME                                  READY    STATUS    RESTARTS   AGE
nginx-sample-6fd7c98d8-7dqdr          1/1      Running   0          1m
nginx-sample-6fd7c98d8-g5k7v          1/1      Running   0          1m
nginx-sample-6fd7c98d8-m7vn7          1/1      Running   0          1m
```

   b. CR ステータスを確認します。

```
$ oc get nginx/nginx-sample -o yaml
```

出力例

```
apiVersion: demo.example.com/v1
kind: Nginx
metadata:
...
  name: nginx-sample
...
spec:
  replicaCount: 3
status:
```

```
nodes:
- nginx-sample-6fd7c98d8-7dqdr
- nginx-sample-6fd7c98d8-g5k7v
- nginx-sample-6fd7c98d8-m7vn7
```

7. デプロイメントサイズを更新します。

   a. **config/samples/demo_v1_nginx.yaml** ファイルを更新して、**Nginx** CR の **spec.size** フィールドを **3** から **5** に変更します。

```
$ oc patch nginx nginx-sample \
    -p '{"spec":{"replicaCount": 5}}' \
    --type=merge
```

   b. Operator がデプロイメントサイズを変更することを確認します。

```
$ oc get deployments
```

   出力例

```
NAME                                READY   UP-TO-DATE   AVAILABLE   AGE
nginx-operator-controller-manager   1/1     1            1           10m
nginx-sample                        5/5     5            5           3m
```

8. 次のコマンドを実行して CR を削除します。

```
$ oc delete -f config/samples/demo_v1_nginx.yaml
```

9. このチュートリアルの一環として作成したリソースをクリーンアップします。

   - Operator のテストに **make deploy** コマンドを使用した場合は、以下のコマンドを実行します。

```
$ make undeploy
```

   - Operator のテストに **operator-sdk run bundle** コマンドを使用した場合は、以下のコマンドを実行します。

```
$ operator-sdk cleanup <project_name>
```

5.5.1.7. 関連情報

- Operator SDK によって作成されるディレクトリー構造の詳細は、Helm ベースの Operator のプロジェクトレイアウト を参照してください。
- クラスター全体の egress プロキシーが設定されている場合、**dedicated-admin** ロールを持つ管理者は、Operator Lifecycle Manager (OLM) で実行されている特定の Operator の プロキシー設定をオーバーライドしたり、カスタム CA 証明書を挿入 したりできます。

5.5.2. Helm ベースの Operator のプロジェクトレイアウト

operator-sdk CLI は、各 Operator プロジェクトに多数のパッケージおよびファイルを生成、または スキャフォールディング することができます。

5.5.2.1. Helm ベースのプロジェクトレイアウト

operator-sdk init --plugins helm コマンドを使用して生成される Helm ベースの 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
config/	Kubernetes クラスターへの Operator のデプロイに使用する Kustomize マニフェスト。
helm-charts/	**operator-sdk create api** コマンドで初期化された Helm チャート。
Dockerfile	**make docker-build** コマンドで Operator イメージをビルドする際に使用します。
watches.yaml	group/version/kind (GVK) および Helm チャートの場所。
Makefile	プロジェクトの管理に使用するターゲット。
PROJECT	Operator のメタデータ情報が含まれる YAML ファイル。

5.5.3. 新しい Operator SDK バージョンの Helm ベースのプロジェクトの更新

OpenShift Dedicated 4 は Operator SDK 1.31.0 をサポートします。ワークステーションに 1.28.0 CLI がすでにインストールされている場合は、最新バージョンをインストール して CLI を 1.31.0 に更新できます。

ただし、既存の Operator プロジェクトが Operator SDK 1.31.0 との互換性を維持するには、1.28.0 以降に導入された関連する重大な変更に対し、更新手順を実行する必要があります。アップグレードの手順は、以前は 1.28.0 で作成または維持されている Operator プロジェクトのいずれかで手動で実行する必要があります。

5.5.3.1. Operator SDK 1.31.0 の Helm ベースの Operator プロジェクトの更新

次の手順では、1.31.0 との互換性を確保するため、既存の Helm ベースの Operator プロジェクトを更新します。

前提条件

- Operator SDK 1.31.0 がインストールされている
- Operator SDK 1.28.0 で作成または保守されている Operator プロジェクト

手順

1. 次の例に示すように、Operator の Dockerfile を編集して Helm Operator バージョンを 1.31.0 に更新します。

   Dockerfile の例

   ```
   FROM quay.io/operator-framework/helm-operator:v1.31.0 1
   ```

1 Helm Operator のバージョンを **1.28.0** から **1.31.0** に更新します。

2. 次の例に示すように、Operator プロジェクトの Makefile を編集して Operator SDK を 1.31.0 に更新します。

例 makefile

```
# Set the Operator SDK version to use. By default, what is installed on the system is used.
# This is useful for CI or a project to utilize a specific version of the operator-sdk toolkit.
OPERATOR_SDK_VERSION ?= v1.31.0 1
```

1 バージョンを **1.28.0** から **1.31.0** に変更します。

3. デプロイにカスタムサービスアカウントを使用する場合は、次の例に示すように、シークレットリソースに対する監視操作を要求する次のロールを定義します。

config/rbac/role.yaml ファイルの例

```
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRoleBinding
metadata:
  name: <operator_name>-admin
subjects:
- kind: ServiceAccount
  name: <operator_name>
  namespace: <operator_namespace>
roleRef:
  kind: ClusterRole
  name: cluster-admin
  apiGroup: ""
rules: 1
  - apiGroups:
      - ""
    resources:
      - secrets
    verbs:
      - watch
```

1 **rules** スタンザを追加して、シークレットリソースの監視オペレーションを作成します。

5.5.3.2. 関連情報

- パッケージマニフェストプロジェクトのバンドル形式への移行
- Operator SDK 1.16.0 のプロジェクトのアップグレード
- Operator SDK v1.10.1 のプロジェクトのアップグレード
- Operator SDK v1.8.0 のプロジェクトのアップグレード

5.5.4. Operator SDK での Helm サポート

5.5.4.1. Helm チャート

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 **spec** 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リストです。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リスト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) で CR を登録することのみが必要になります。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。

5.6. クラスターサービスバージョン (CSV) の定義

クラスターサービスバージョン (CSV) は、**ClusterServiceVersion** オブジェクトで定義され、Operator Lifecycle Manager (OLM) によるクラスターでの Operator の実行をサポートする 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イスにロゴ、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に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、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 CSV を生成するための CSV ジェネレーターが含まれます。

CSV で生成されるコマンドにより、Operator の作成者が OLM について詳しく知らなくても、Operator は OLM と対話したり、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 仕様が変更される可能性が高いため、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。

5.6.1. CSV 生成の仕組み

クラスターサービスバージョン (CSV) を含む Operator バンドルマニフェストは、Operator Lifecycle Manager (OLM) でアプリケーションを表示、作成、および管理する方法を説明します。**generate bundle** サブコマンドによって呼び出される Operator SDK の CSV ジェネレーターは、Operator をカタログに公開し、これを OLM でデプロイする最初の手順になります。サブコマンドには、CSV マニフェストを作成するための特定の入力マニフェストが必要です。すべての入力は、コマンドが CSV ベースと共に呼び出される際に読み取られ、べき等性で CSV を生成したり、再生成したりします。

通常は、**generate kustomize manifests** サブコマンドが最初に実行され、**generate bundle** サブコマンドで使用される入力された Kustomize ベースを生成します。ただし、Operator SDK は **make bundle** コマンドを提供します。これは、以下のサブコマンドを順番に実行するなどの複数のタスクを自動化します。

1. **generate kustomize manifests**
2. **generate bundle**
3. **bundle validate**

関連情報

- バンドルと CSV の生成を含む詳細な手順は、Operator のバンドル を参照してください。

5.6.1.1. 生成されるファイルおよびリソース

make bundle コマンドは、以下のファイルおよびディレクトリーを Operator プロジェクトに作成します。

- **ClusterServiceVersion** (CSV) オブジェクトを含む **bundle/manifests** という名前のバンドルマニフェストディレクトリー
- **bundle/metadata** という名前のバンドルメタデータディレクトリー
- **config/crd** ディレクトリー内のすべてのカスタムリソース定義 (CRD)
- Dockerfile **bundle.Dockerfile**

通常、以下のリソースは CSV に含まれます。

Role
namespace 内で Operator パーミッションを定義します。

ClusterRole
クラスター全体の Operator パーミッションを定義します。

デプロイメント
Operator のオペランドが Pod で実行される方法を定義します。

CustomResourceDefinition (CRD)

Operator が調整するカスタムリソースを定義します。

カスタムリソースの例

特定の CRD の仕様に従ったリソースの例。

5.6.1.2. バージョンの管理

generate bundle サブコマンドの **--version** フラグは、バンドルの初回作成時および既存バンドルのアップグレード時に、バンドルのセマンティックバージョンを提供します。

Makefile に **VERSION** 変数を設定することで、**--version** フラグは、**generate bundle** サブコマンドが **make bundle** コマンドによって実行される際に、値を使用して自動的に呼び出されます。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。

5.6.2. 手動で定義される CSV フィールド

多くの CSV フィールドは、生成された、Operator SDK に特化していない汎用マニフェストを使用して設定することはできません。これらのフィールドは、ほとんどの場合、Operator および各種のカスタムリソース定義 (CRD) に関する人間が作成するメタデータです。

Operator 作成者はそれらのクラスターサービスバージョン (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 生成時に警告を送信します。

以下の表は、手動で定義された CSV フィールドのうち、必須フィールドとオプションフィールドについて詳細に示しています。

表5.7 必須の **CSV** フィールド

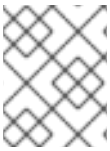
フィールド	説明
metadata.name	CSV の固有名。Operator バージョンは、**app-operator.v0.1.1** などのように一意性を確保するために名前に含める必要があります。
metadata.capabilities	Operator の成熟度モデルに応じた機能レベル。オプションには、**Basic Install**、**Seamless Upgrades**、**Full Lifecycle**、 **Deep Insights**、および **Auto Pilot** が含まれます。
spec.displayName	Operator を識別するためのパブリック名。
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)。

フィールド	説明
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**。
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**: description of the CRD. • **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** オブジェクト)。 • **specDescriptors**: Operator の入力および出力についての UI ヒント。

表5.8 オプションの CSV フィールド

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	このバージョンでソフトウェアが達成した成熟度。オプションに、**planning**、**pre-alpha**、**alpha**、**beta**、 **stable**、**mature**、 **inactive**、および **deprecated** が含まれます。

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。

関連情報

- Operator 成熟度モデル

5.6.3. Operator メタデータアノテーション

Operator 開発者は、クラスターサービスバージョン (CSV) のメタデータに特定のアノテーションを設定して、OperatorHub や Red Hat Ecosystem Catalog などのユーザーインターフェイス (UI) の機能を

有効にしたり、機能を強調したりできます。Operator のメタデータのアノテーションは、CSV YAML ファイルの **metadata.annotations** フィールドを設定して手動で定義します。

5.6.3.1. インフラストラクチャー機能のアノテーション

features.operators.openshift.io グループのアノテーションは、Operator がサポートするインフラストラクチャー機能の詳細を表すものです。この機能は、**"true"** または **"false"** 値を設定することで指定されます。ユーザーは、Web コンソールや Red Hat Ecosystem Catalog の OperatorHub から Operator を探すときに、これらの機能を使用して表示やフィルタリングを行うことができます。これらのアノテーションは、OpenShift Dedicated 4.10 以降でサポートされています。

重要

features.operators.openshift.io のインフラストラクチャー機能のアノテーションにより、OpenShift Dedicated の以前のバージョンで使用されていた **operators.openshift.io/infrastructure-features** アノテーションが非推奨となりました。詳細は、「非推奨のインフラストラクチャー機能のアノテーション」を参照してください。

表5.9 インフラストラクチャー機能のアノテーション

アノテーション	説明	有効な値 [1]
features.operators.openshift.io/disconnected	Operator が **spec.relatedImages** CSV フィールドを利用し、ダイジェストにより関連イメージを参照することでインターネット接続なしで実行できるかどうかを指定します。	**"true"** または **"false"**
features.operators.openshift.io/fips-compliant	Operator が基盤となるプラットフォームの FIPS-140 設定を受け入れ、FIPS モードで起動したノードで動作するかどうかを指定します。このモードでは、Operator とそれが管理するすべてのワークロード (オペランド) は、FIPS-140 検証用に提出された Red Hat Enterprise Linux (RHEL) 暗号化ライブラリーのみを呼び出します。	**"true"** または **"false"**
features.operators.openshift.io/proxy-aware	標準の **HTTP_PROXY** および **HTTPS_PROXY** プロキシー環境変数を受け入れて、プロキシーの背後にあるクラスターでの実行を Operator がサポートするかどうかを指定します。該当する場合、Operator はこの情報を管理対象のワークロード (オペランド) に渡します。	**"true"** または **"false"**
features.operators.openshift.io/tls-profiles	Operator が使用する TLS 暗号スイート、および該当する場合は管理対象のワークロード (オペランド) を変更するための既知の設定項目を Operator が実装するかどうかを指定します。	**"true"** または **"false"**

アノテーション	説明	有効な値 [1]
features.operators.openshift.io/token-auth-aws	Cloud Credential Operator (CCO) を使用して、AWS Secure Token Service (STS) を介した AWS API によるトークン化認証の設定を Operator がサポートするかどうかを指定します。	**"true"** または **"false"**
features.operators.openshift.io/token-auth-azure	Cloud Credential Operator (CCO) を使用して、Azure マネージド ID を介した Azure API によるトークン化認証の設定を Operator がサポートするかどうかを指定します。	**"true"** または **"false"**
features.operators.openshift.io/token-auth-gcp	Cloud Credential Operator (CCO) を使用して、GCP Workload Identity Foundation (WIF) を介した Google Cloud API によるトークン化認証の設定を Operator がサポートするかどうかを指定します。	**"true"** または **"false"**
features.operators.openshift.io/cnf	Cloud-Native Network Function (CNF) Kubernetes プラグインを Operator が提供するかどうかを指定します。	**"true"** または **"false"**
features.operators.openshift.io/cni	Container Network Interface (CNI) Kubernetes プラグインを Operator が提供するかどうかを指定します。	**"true"** または **"false"**
features.operators.openshift.io/csi	Container Storage Interface (CSI) Kubernetes プラグインを Operator が提供するかどうかを指定します。	**"true"** または **"false"**

1. Kubernetes アノテーションは文字列でなければならないため、有効な値は意図的に二重引用符で囲んで表示されています。

インフラストラクチャー機能のアノテーションを含む CSV の例

```
apiVersion: operators.coreos.com/v1alpha1
kind: ClusterServiceVersion
metadata:
  annotations:
    features.operators.openshift.io/disconnected: "true"
    features.operators.openshift.io/fips-compliant: "false"
    features.operators.openshift.io/proxy-aware: "false"
    features.operators.openshift.io/tls-profiles: "false"
    features.operators.openshift.io/token-auth-aws: "false"
    features.operators.openshift.io/token-auth-azure: "false"
    features.operators.openshift.io/token-auth-gcp: "false"
```

関連情報

- ネットワークが制限された環境についての Operator の有効化(非接続モード)

5.6.3.2. 非推奨のインフラストラクチャー機能のアノテーション

OpenShift Dedicated 4.14 以降、**operators.openshift.io/infrastructure-features** のアノテーションのグループが、**features.operators.openshift.io** namespace のアノテーションのグループによって非推奨となりました。新しいアノテーションを使用することを推奨しますが、現在は両方のグループを並行して使用できます。

これらのアノテーションは、Operator がサポートするインフラストラクチャー機能の詳細を表すものです。ユーザーは、Web コンソールや Red Hat Ecosystem Catalog の OperatorHub から Operator を探すときに、これらの機能を使用して表示やフィルタリングを行うことができます。

表5.10 非推奨の**operators.openshift.io/infrastructor-features** アノテーション

有効なアノテーション値	説明
disconnected	Operator はすべての依存関係を含む非接続カタログにミラーリングされるため、インターネットへのアクセスは必要ありません。ミラーリングに必要なすべての関連イメージが Operator によって一覧表示されます。
cnf	Operator は Cloud-native Network Functions (CNF) Kubernetes プラグインを提供します。
cni	Operator は Container Network Interface (CNI) Kubernetes プラグインを提供します。
csi	Operator は Container Storage Interface (CSI) Kubernetes プラグインを提供します。
fips	Operator は基盤となるプラットフォームの FIPS モードを受け入れ、FIPS モードで起動したノードで動作します。  **重要** FIPS モードで起動した Red Hat Enterprise Linux (RHEL) または Red Hat Enterprise Linux CoreOS (RHCOS) を実行する場合、OpenShift Dedicated のコアコンポーネントは、x86_64、ppc64le、および s390x アーキテクチャー上でのみ、FIPS 140-2/140-3 検証用に NIST に提出された RHEL 暗号化ライブラリーを使用します。
proxy-aware	Operator はプロキシーの背後のクラスターでの実行をサポートします。Operator は、クラスターがプロキシーを使用するように設定される際に Operator Lifecycle Manager (OLM) が Operator に自動的に提供する標準のプロキシー環境変数の **HTTP_PROXY** および **HTTPS_PROXY** を受け入れます。必要な環境変数は、管理されているワークロード用にオペランドに渡されます。

disconnected および proxy-aware サポートを含む CSV の例

```
apiVersion: operators.coreos.com/v1alpha1
kind: ClusterServiceVersion
metadata:
  annotations:
    operators.openshift.io/infrastructure-features: '["disconnected", "proxy-aware"]'
```

5.6.3.3. その他の任意のアノテーション

次の Operator のアノテーションは任意です。

表5.11 その他の任意のアノテーション

アノテーション	説明
alm-examples	カスタムリソース定義 (CRD) テンプレートに最低限の設定セットを指定します。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。
operatorframework.io/initialization-resource	Operator のインストール中に、クラスターサービスバージョン (CSV) に **operatorframework.io/initialization-resource** アノテーションを追加することで、必要なカスタムリソースを 1 つ指定します。ユーザーは、CSV で提供されるテンプレートを使用してカスタムリソースを作成するように求められます。完全な YAML 定義が含まれるテンプレートを含める必要があります。
operatorframework.io/suggested-namespace	Operator をデプロイする必要のある推奨 namespace を設定します。
operatorframework.io/suggested-namespace-template	指定された **Namespace** のデフォルトのノードセレクターを使用して、namespace オブジェクトのマニフェストを設定します。
operators.openshift.io/valid-subscription	Operator を使用するために必要とされる特定のサブスクリプションをリスト表示するための自由形式の配列です。例: **'["3Scale Commercial License", "Red Hat Managed Integration"]'**
operators.operatorframework.io/internal-objects	ユーザーの操作を目的としていない UI の CRD を非表示にします。

OpenShift Dedicated ライセンス要件を含む CSV の例

```
apiVersion: operators.coreos.com/v1alpha1
kind: ClusterServiceVersion
metadata:
  annotations:
    operators.openshift.io/valid-subscription: '["OpenShift Container Platform"]'
```

3scale ライセンス要件を含む CSV の例

```
apiVersion: operators.coreos.com/v1alpha1
kind: ClusterServiceVersion
metadata:
  annotations:
    operators.openshift.io/valid-subscription: '["3Scale Commercial License", "Red Hat Managed Integration"]'
```

関連情報

- CRD テンプレート
- 必要なカスタムリソースの初期化
- 推奨される namespace の設定
- デフォルトのノードセレクターを使用して推奨される namespace を設定する
- 内部オブジェクトの非表示

5.6.4. ネットワークが制限された環境についての Operator の有効化

Operator の作成者は、Operator がネットワークが制限された環境、または非接続の環境で適切に実行されるよう追加要件を満たすことを確認する必要があります。

非接続モードをサポートするための **Operator** の要件

- ハードコードされたイメージ参照は、環境変数に置き換えます。
- Operator のクラスターサービスバージョン (CSV) で以下を行います。
  - Operator がそれらの機能を実行するために必要となる可能性のある 関連イメージ または他のコンテナーをリスト表示します。
  - 指定されたすべてのイメージを、タグではなくダイジェスト (SHA) で参照します。
- Operator のすべての依存関係は、非接続モードでの実行もサポートする必要があります。
- Operator にはクラスター外のリソースは必要ありません。

前提条件

- CSV を含む Operator プロジェクト次の手順では、Go ベース、Ansible ベース、および Helm ベースのプロジェクトの例として Memcached Operator を使用します。

手順

1. **config/manager/manager.yaml**ファイルで、Operator が使用する追加のイメージ参照の環境変数を設定します。

   例5.2 **config/manager/manager.yaml** ファイル例

   ```
   ...
   spec:
   ```

```
...
 spec:
   ...
   containers:
   - command:
     - /manager
     ...
     env:
     - name: <related_image_environment_variable> (1)
       value: "<related_image_reference_with_tag>" (2)
```

1. **RELATED_IMAGE_MEMCACHED**などの環境変数を定義します。
2. **docker.io/memcached:1.4.36-alpine**などの関連するイメージ参照とタグを設定します。

2. ハードコードされたイメージ参照は、Operator プロジェクトタイプに関連するファイルの環境変数に置き換えます。

- Go ベースの Operator プロジェクトの場合には、次の例に示すように、環境変数を **controllers/memcached_controller.go** ファイルに追加します。

例5.3 **controllers/memcached_controller.go** ファイルの例

```
  // deploymentForMemcached returns a memcached Deployment object

...

 Spec: corev1.PodSpec{
        Containers: []corev1.Container{{
-   Image:   "memcached:1.4.36-alpine", (1)
+   Image:   os.Getenv("<related_image_environment_variable>"), (2)
    Name:    "memcached",
    Command: []string{"memcached", "-m=64", "-o", "modern", "-v"},
    Ports: []corev1.ContainerPort{{

...
```

1. イメージ参照とタグを削除します。
2. **os.Getenv**関数を使用して、 **<related_image_environment_variable>**を呼び出します。

注記

変数が設定されていない場合、 **os.Getenv**関数は空の文字列を返します。ファイルを変更する前に、**<related_image_environment_variable>**を設定してください。

- Ansible ベースの Operator プロジェクトの場合には、次の例に示すように、環境変数を**roles/memcached/tasks/main.yml**ファイルに追加します。

例5.4 **roles/memcached/tasks/main.yml** ファイルの例

```
spec:
  containers:
  - name: memcached
    command:
    - memcached
    - -m=64
    - -o
    - modern
    - -v
-   image: "docker.io/memcached:1.4.36-alpine"
+   image: "{{ lookup('env', '<related_image_environment_variable>') }}"
    ports:
      - containerPort: 11211

...
```

1. イメージ参照とタグを削除します。
2. **lookup** 関数を使用して、 **<related_image_environment_variable>**を呼び出します。

- Helm ベースの Operator プロジェクトの場合、以下の例のように **overrideValues** フィールドを **watches.yaml** ファイルに追加します。

例5.5 **watches.yaml** ファイルの例

```
...
- group: demo.example.com
  version: v1alpha1
  kind: Memcached
  chart: helm-charts/memcached
  overrideValues: 1
    relatedImage: ${<related_image_environment_variable>} 2
```

1. **overrideValues** フィールドを追加します。
2. **<related_image_environment_variable>** を使用して **overrideValues** フィールドを定義します (例: **RELATED_IMAGE_MEMCACHED**)。

a. 以下の例のように、**overrideValues** フィールドの値を **helm-charts/memchached/values.yaml** ファイルに追加します。

helm-charts/memchached/values.yaml ファイルの例

```
...
relatedImage: ""
```

b. 以下の例のように、**helm-charts/memcached/templates/deployment.yaml** ファイルのチャートテンプレートを編集します。

例5.6 **helm-charts/memcached/templates/deployment.yaml** ファイルの例

```
containers:
  - name: {{ .Chart.Name }}
    securityContext:
      - toYaml {{ .Values.securityContext | nindent 12 }}
    image: "{{ .Values.image.pullPolicy }}
    env: (1)
      - name: related_image (2)
        value: "{{ .Values.relatedImage }}" (3)
```

(1) **env** フィールドを追加します。

(2) 環境変数に名前を付けます。

(3) 環境変数の値を定義します。

3. 次の変更を加えて、**BUNDLE_GEN_FLAGS**変数定義を**Makefile**に追加します。

Makefile の例

```
  BUNDLE_GEN_FLAGS ?= -q --overwrite --version $(VERSION)
$(BUNDLE_METADATA_OPTS)

  # USE_IMAGE_DIGESTS defines if images are resolved via tags or digests
  # You can enable this value if you would like to use SHA Based Digests
  # To enable set flag to true
  USE_IMAGE_DIGESTS ?= false
  ifeq ($(USE_IMAGE_DIGESTS), true)
      BUNDLE_GEN_FLAGS += --use-image-digests
  endif

...

-  $(KUSTOMIZE) build config/manifests | operator-sdk generate bundle -q --overwrite --
version $(VERSION) $(BUNDLE_METADATA_OPTS) (1)
+  $(KUSTOMIZE) build config/manifests | operator-sdk generate bundle
$(BUNDLE_GEN_FLAGS) (2)

...
```

(1) **Makefile**でこの行を削除します。

(2) 上記の行は、この行に置き換えます。

4. タグではなくダイジェスト (SHA) を使用するように Operator イメージを更新するには、**make bundle**コマンドを実行し、**USE_IMAGE_DIGESTS**を**true**に設定します。

```
$ make bundle USE_IMAGE_DIGESTS=true
```

5. **disconnected** アノテーションを追加します。これは、Operator が非接続環境で機能することを示します。

```
metadata:
  annotations:
    operators.openshift.io/infrastructure-features: '["disconnected"]'
```

Operator は、このインフラストラクチャー機能によって OperatorHub でフィルターされます。

5.6.5. 複数のアーキテクチャーおよびオペレーティングシステム用の Operator の有効化

Operator Lifecycle Manager (OLM) は、すべての Operator が Linux ホスト上で実行されることを前提としています。ただし、Operator 作成者は、ワーカーノードが OpenShift Dedicated クラスターで利用可能な場合に、Operator が他のアーキテクチャーでのワークロードの管理をサポートするかどうかを指定できます。

Operator が AMD64 および Linux 以外のバリアントをサポートする場合、サポートされるバリアントをリスト表示するために Operator を提供するクラスターサービスバージョン (CSV) にラベルを追加できます。サポートされているアーキテクチャーとオペレーティングシステムを示すラベルは、以下で定義されます。

```
labels:
    operatorframework.io/arch.<arch>: supported (1)
    operatorframework.io/os.<os>: supported (2)
```

(1) **<arch>** をサポートされる文字列に設定します。

(2) **<os>** をサポートされる文字列に設定します。

注記

デフォルトチャネルのチャネルヘッドにあるラベルのみが、パッケージマニフェストをラベルでフィルターする場合に考慮されます。たとえば、デフォルト以外のチャネルで Operator の追加アーキテクチャーを提供することは可能ですが、そのアーキテクチャーは **PackageManifest** API でのフィルターには使用できません。

CSV に **os** ラベルが含まれていない場合、これはデフォルトで以下の Linux サポートラベルが設定されているかのように処理されます。

```
labels:
    operatorframework.io/os.linux: supported
```

CSV に **arch** ラベルが含まれていない場合、これはデフォルトで以下の AMD64 サポートラベルが設定されているかのように処理されます。

```
labels:
    operatorframework.io/arch.amd64: supported
```

Operator が複数のノードアーキテクチャーまたはオペレーティングシステムをサポートする場合、複数のラベルを追加することもできます。

前提条件

- CSV を含む Operator プロジェクト
- 複数のアーキテクチャーおよびオペレーティングシステムのリスト表示をサポートするには、CSV で参照される Operator イメージはマニフェストリストイメージである必要があります。
- Operator がネットワークが制限された環境または非接続環境で適切に機能できるようにするには、参照されるイメージは、タグではなくダイジェスト (SHA) を使用して指定される必要もあります。

手順

- Operator がサポートするサポートされるアーキテクチャーおよびオペレーティングシステムのそれぞれについて CSV の **metadata.labels** にラベルを追加します。

```
labels:
  operatorframework.io/arch.s390x: supported
  operatorframework.io/os.zos: supported
  operatorframework.io/os.linux: supported 1
  operatorframework.io/arch.amd64: supported 2
```

1 2 新規のアーキテクチャーまたはオペレーティングシステムを追加したら、デフォルトの **os.linux** および **arch.amd64** バリアントも明示的に組み込む必要があります。

関連情報

- マニフェストのリストについての詳細は、Image Manifest V 2, Schema 2 仕様を参照してください。

5.6.5.1. Operator のアーキテクチャーおよびオペレーティングシステムのサポート

以下の文字列は、複数のアーキテクチャーおよびオペレーティングシステムをサポートする Operator のラベル付けまたはフィルター時に OpenShift Dedicated の Operator Lifecycle Manager (OLM) でサポートされます。

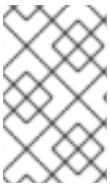
表5.12 OpenShift Dedicated でサポートされるアーキテクチャー

アーキテクチャー	文字列
AMD64	**amd64**
IBM Power®	**ppc64le**
IBM Z®	**s390x**

表5.13 OpenShift Dedicated でサポートされるオペレーティングシステム

オペレーティングシステム	文字列
Linux	**linux**

オペレーティングシステム	文字列
z/OS	**zos**



注記

サポートされるアーキテクチャーとオペレーティングシステムのセットは、OpenShift Dedicated のバージョンや他の Kubernetes ベースのディストリビューションによって異なる場合があります。

5.6.6. 推奨される namespace の設定

Operator が正しく機能するには、一部の Operator を特定の namespace にデプロイするか、特定の namespace で補助リソースと共にデプロイする必要があります。サブスクリプションから解決されている場合、Operator Lifecycle Manager (OLM) は Operator の namespace を使用したリソースをそのサブスクリプションの namespace にデフォルト設定します。

Operator の作成者は、必要なターゲット namespace をクラスターサービスバージョン (CSV) の一部として表現し、それらの Operator にインストールされるリソースの最終的な namespace の制御を維持できます。OperatorHub を使用してクラスターに Operator を追加する場合、インストールプロセス中に Web コンソールがインストーラーに応じて推奨される namespace を自動設定します。

手順

- CSV で、**operatorframework.io/suggested-namespace** アノテーションを提案される namespace に設定します。

```
metadata:
  annotations:
    operatorframework.io/suggested-namespace: <namespace> 1
```

1 提案された namespace を設定します。

5.6.7. デフォルトのノードセレクターを使用して推奨される namespace を設定する

一部の Operator は、コントロールプレーンノードでのみ実行することを想定しています。これは、Operator 自体が **Pod** 仕様で **nodeSelector** を設定することによって実行できます。

クラスター全体のデフォルト **nodeSelector** が重複して競合する可能性を回避するために、Operator が実行される namespace にデフォルトのノードセレクターを設定できます。デフォルトのノードセレクターはクラスターのデフォルトよりも優先されるため、クラスターのデフォルトは Operator の namespace 内の Pod には適用されません。

OperatorHub を使用して Operator をクラスターに追加するする場合、インストールプロセス中に Web コンソールがインストーラーに応じて推奨される namespace を自動設定します。推奨される namespace は、クラスターサービスバージョン (CSV) に含まれる YAML の namespace マニフェストを使用して作成されます。

手順

- CSV で、**namespace** オブジェクトのマニフェストを使用して **operatorframework.io/suggested-namespace-template** を設定します。次のサンプルは、

namespace のデフォルトのノードセレクターが指定された **Namespace** の例のマニフェストです。

```
metadata:
  annotations:
    operatorframework.io/suggested-namespace-template: 1
    {
      "apiVersion": "v1",
      "kind": "Namespace",
      "metadata": {
        "name": "vertical-pod-autoscaler-suggested-template",
        "annotations": {
          "openshift.io/node-selector": ""
        }
      }
    }
```

1 提案された namespace を設定します。

注記

suggested-namespace および **suggested-namespace-template** アノテーションの両方が CSV にある場合、**suggested-namespace-template** が優先されます。

5.6.8. Operator 条件の有効化

Operator Lifecycle Manager (OLM) は、Operator を管理する一方で OLM の動作に影響を与える複雑な状態を通信するためのチャネルを Operator に提供します。デフォルトで、OLM は Operator のインストール時に **OperatorCondition** カスタムリソース定義 (CRD) を作成します。**OperatorCondition** カスタムリソース (CR) に設定される条件に基づいて、OLM の動作は随時変わります。

Operator 条件をサポートするには、Operator は OLM によって作成された **OperatorCondition** CR を読み取ることができ、次のタスクを完了することができる必要があります。

- 特定の条件を取得します。
- 特定の条件のステータスを設定します。

これは、**operator-lib** ライブラリーを使用して実行できます。Operator の作成者は、ライブラリーがクラスター内の Operator が所有する **OperatorCondition** CR にアクセスできるように Operator に **controller-runtime** クライアント を指定できます。

ライブラリーは汎用的な **Conditions** インターフェイスを提供します。これには、**OperatorCondition** CR で **conditionType** の **Get** および **Set** を実行するための以下のメソッドがあります。

Get

特定の条件を取得するために、ライブラリーは **controller-runtime** の **client.Get** 機能を使用します。これには、**conditionAccessor** にあるタイプが **types.NamespacedName** の **ObjectKey** が必要です。

Set

特定の条件のステータスを更新するために、ライブラリーは **controller-runtime** の **client.Update** 機能を使用します。**conditionType** が CRD にない場合、エラーが生じます。

Operator は CR の **status** サブリソースのみを変更することができます。Operator は **status.conditions** 配列を削除したり、条件を追加できるようにこれを更新したりすることができます。条件にあるフィールドの形式および説明の詳細は、アップストリームの Condition GoDocs を参照してください。

注記

Operator SDK 1.31.0 は **operator-lib** v0.11.0 をサポートします。

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

Operator プロジェクトで Operator 条件を有効にするには、以下を実行します。

1. Operator プロジェクトの **go.mod** ファイルで、**operator-framework/operator-lib** を必要なライブラリーとして追加します。

```
module github.com/example-inc/memcached-operator

go 1.19

require (
  k8s.io/apimachinery v0.26.0
  k8s.io/client-go v0.26.0
  sigs.k8s.io/controller-runtime v0.14.1
  operator-framework/operator-lib v0.11.0
)
```

2. Operator ロジックに独自のコンストラクターを作成すると、次の結果が得られます。

   - **controller-runtime** クライアントを許可します。
   - **conditionType** を受け入れます。
   - 条件を更新または追加する **Condition** インターフェイスを返します。

   現時点で OLM は **Upgradeable** 状態をサポートするため、**Upgradeable** 条件にアクセスするためのメソッドを持つインターフェイスを作成できます。以下に例を示します。

```
import (
  ...
  apiv1 "github.com/operator-framework/api/pkg/operators/v1"
)

func NewUpgradeable(cl client.Client) (Condition, error) {
  return NewCondition(cl, "apiv1.OperatorUpgradeable")
}

cond, err := NewUpgradeable(cl);
```

   この例では、**NewUpgradeable** コンストラクターが、タイプ **Condition** の変数 **cond** を使用するためにさらに使用されます。**cond** 変数には、OLM の **Upgradeable** 条件を処理するために使用できる **Get** および **Set** メソッドが含まれます。

関連情報

- Operator 条件

5.6.9. Webhook の定義

Webhook により、リソースがオブジェクトストアに保存され、Operator コントローラーによって処理される前に、Operator の作成者はリソースのインターセプト、変更、許可、および拒否を実行することができます。Operator Lifecycle Manager (OLM) は、Operator と共に提供される際にこれらの Webhook のライフサイクルを管理できます。

Operator のクラスターサービスバージョン (CSV) リソースには、以下のタイプの Webhook を定義するために **webhookdefinitions** セクションを含めることができます。

- 受付 Webhook (検証および変更用)
- 変換 Webhook

手順

- **webhookdefinitions** セクションを Operator の CSV の **spec** セクションに追加し、**type** として **ValidatingAdmissionWebhook**、 **MutatingAdmissionWebhook**、または **ConversionWebhook** を使用して Webhook 定義を追加します。以下の例には、3 つのタイプの Webhook がすべて含まれます。

  Webhook が含まれる CSV

```
apiVersion: operators.coreos.com/v1alpha1
kind: ClusterServiceVersion
metadata:
  name: webhook-operator.v0.0.1
spec:
  customresourcedefinitions:
    owned:
    - kind: WebhookTest
      name: webhooktests.webhook.operators.coreos.io 1
      version: v1
  install:
    spec:
      deployments:
      - name: webhook-operator-webhook
        ...
        ...
        ...
    strategy: deployment
  installModes:
  - supported: false
    type: OwnNamespace
  - supported: false
    type: SingleNamespace
  - supported: false
    type: MultiNamespace
  - supported: true
    type: AllNamespaces
  webhookdefinitions:
```

```
    - type: ValidatingAdmissionWebhook 2
      admissionReviewVersions:
      - v1beta1
      - v1
      containerPort: 443
      targetPort: 4343
      deploymentName: webhook-operator-webhook
      failurePolicy: Fail
      generateName: vwebhooktest.kb.io
      rules:
      - apiGroups:
        - webhook.operators.coreos.io
        apiVersions:
        - v1
        operations:
        - CREATE
        - UPDATE
        resources:
        - webhooktests
      sideEffects: None
      webhookPath: /validate-webhook-operators-coreos-io-v1-webhooktest
    - type: MutatingAdmissionWebhook 3
      admissionReviewVersions:
      - v1beta1
      - v1
      containerPort: 443
      targetPort: 4343
      deploymentName: webhook-operator-webhook
      failurePolicy: Fail
      generateName: mwebhooktest.kb.io
      rules:
      - apiGroups:
        - webhook.operators.coreos.io
        apiVersions:
        - v1
        operations:
        - CREATE
        - UPDATE
        resources:
        - webhooktests
      sideEffects: None
      webhookPath: /mutate-webhook-operators-coreos-io-v1-webhooktest
    - type: ConversionWebhook 4
      admissionReviewVersions:
      - v1beta1
      - v1
      containerPort: 443
      targetPort: 4343
      deploymentName: webhook-operator-webhook
      generateName: cwebhooktest.kb.io
      sideEffects: None
      webhookPath: /convert
      conversionCRDs:
      - webhooktests.webhook.operators.coreos.io 5
...
```

1. 変換 Webhook がターゲットとする CRD がここに存在している必要があります。
2. 検証用の受付 Webhook。
3. 変更用の受付 Webhook。
4. 変換 Webhook。
5. 各 CRD の **spec.PreserveUnknownFields** プロパティーは **false** または **nil** に設定される必要があります。

関連情報

- Kubernetes ドキュメント:
  - 検証用の受付 Webhook
  - 変更用の受付 Webhook
  - 変換 Webhook

5.6.9.1. OLM についての Webhook の考慮事項

Operator Lifecycle Manager (OLM) を使用して Webhook で Operator をデプロイする場合、以下を定義する必要があります。

- **type** フィールドは **ValidatingAdmissionWebhook**、**MutatingAdmissionWebhook**、または **ConversionWebhook** のいずれかに設定する必要があります。そうでないと、CSV は失敗フェーズに置かれます。
- CSV には、**webhookdefinition** の **deploymentName** フィールドに指定される値に等しい名前のデプロイメントが含まれる必要があります。

Webhook が作成されると、OLM は、Operator がデプロイされる Operator グループに一致する namespace でのみ Webhook が機能するようにします。

認証局についての制約
OLM は、各デプロイメントに単一の認証局 (CA) を提供するように設定されます。CA を生成してデプロイメントにマウントするロジックは、元々 API サービスのライフサイクルロジックで使用されていました。結果は、以下のようになります。

- TLS 証明書ファイルは、**/apiserver.local.config/certificates/apiserver.crt** にあるデプロイメントにマウントされます。
- TLS キーファイルは、**/apiserver.local.config/certificates/apiserver.key** にあるデプロイメントにマウントされます。

受付 **Webhook** ルールについての制約
Operator がクラスターをリカバリー不可能な状態に設定しないようにするため、OLM は受付 Webhook に定義されたルールが以下の要求のいずれかをインターセプトする場合に、失敗フェーズに CSV を配置します。

- すべてのグループをターゲットとする要求
- **operators.coreos.com** グループをターゲットとする要求

- **ValidatingWebhookConfigurations** または **MutatingWebhookConfigurations** リソースをターゲットとする要求

変換 **Webhook** の制約
OLM は、変換 Webhook 定義が以下の制約に準拠しない場合に、失敗フェーズに CSV を配置します。

- 変換 Webhook と特長とする CSV は、**AllNamespaces** インストールモードのみをサポートできます。
- 変換 Webhook がターゲットとする CRD では、**spec.preserveUnknownFields** フィールドを **false** または **nil** に設定する必要があります。
- CSV で定義される変換 Webhook は所有 CRD をターゲットにする必要があります。
- 特定の CRD には、クラスター全体で 1 つの変換 Webhook のみを使用できます。

5.6.10. カスタムリソース定義 (CRD) について

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。

5.6.10.1. 所有 CRD (Owned CRD)

Operator が所有するカスタムリソース定義 (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レプリカセットの表現など)。それぞれは CSV ファイルにリスト表示される必要があります。

表5.14 所有 CRD フィールド

フィールド	説明	必須/オプション
Name	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 **resources** セクションにリスト表示されます。 この場合、オーケストレーションするすべてのリストではなく、重要なオブジェクトのみをリスト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設定マップをリスト表示しないでください。	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設定マップ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**: **Spec**、**Status**、または **Action** の人間が判読できる名前。 ● **Description**: **Spec**、**Status**、または **Action**、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 Dedicated の正規の React UI X-Descriptor リスト 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設定マップ でユーザー入力を必要とし、サービス、ステートフルセット、Pod および設定マップ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.
```

5.6.10.2. 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5.15 必須 CRD フィールド

フィールド	説明	必須/オプション
Name	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.
```

5.6.10.3. CRD のアップグレード

OLM は、単一のクラスターサービスバージョン (CSV) によって所有されている場合にはカスタムリソース定義 (CRD) をすぐにアップグレードします。CRD が複数の CSV によって所有されている場合、CRD は、以下の後方互換性の条件のすべてを満たす場合にアップグレードされます。

- 現行 CRD の既存の有効にされたバージョンすべてが新規 CRD に存在する。
- 検証が新規 CRD の検証スキーマに対して行われる場合、CRD の提供バージョンに関連付けられる既存インスタンスまたはカスタムリソースすべてが有効である。

5.6.10.3.1. 新規 CRD バージョンの追加

手順

CRD の新規バージョンを Operator に追加するには、以下を実行します。

1. CSV の **versions** セクションに CRD リソースの新規エントリーを追加します。

たとえば、現在の CRD にバージョン **v1alpha1** があり、新規バージョン **v1beta1** を追加し、これを新規のストレージバージョンとしてマークをする場合に、**v1beta1** の新規エントリーを追加します。

```
versions:
  - name: v1alpha1
    served: true
    storage: false
  - name: v1beta1 (1)
    served: true
    storage: true
```

(1) 新規エントリー。

2. CSV が新規バージョンを使用する場合、CSV の **owned** セクションの CRD の参照バージョンが更新されていることを確認します。

```
customresourcedefinitions:
  owned:
  - name: cluster.example.com
    version: v1beta1 (1)
    kind: cluster
    displayName: Cluster
```

(1) **version** を更新します。

3. 更新された CRD および CSV をバンドルにプッシュします。

5.6.10.3.2. CRD バージョンの非推奨または削除

Operator Lifecycle Manager (OLM) では、カスタムリソース定義 (CRD) の提供バージョンをすぐに削除できません。その代わりに、CRD の非推奨バージョンを CRD の **served** フィールドを **false** に設定して無効にする必要があります。その後に、無効にされたバージョンではないバージョンを後続の CRD アップグレードで削除できます。

手順

特定バージョンの CRD を非推奨にし、削除するには、以下を実行します。

1. 非推奨バージョンを non-serving (無効にされたバージョン) とマークして、このバージョンが使用されなくなり、後続のアップグレードで削除される可能性があることを示します。以下に例を示します。

```
versions:
  - name: v1alpha1
    served: false (1)
    storage: true
```

(1) **false** に設定します。

2. 非推奨となるバージョンが現在 **storage** バージョンの場合、**storage** バージョンを有効にされたバージョンに切り替えます。以下に例を示します。

```
versions:
  - name: v1alpha1
    served: false
    storage: false 1
  - name: v1beta1
    served: true
    storage: true 2
```

1 2 **storage** フィールドを適宜更新します。

注記

CRD から **storage** バージョンであるか、このバージョンであった特定のバージョンを削除するために、そのバージョンが CRD のステータスの **storedVersion** から削除される必要があります。OLM は、保存されたバージョンが新しい CRD に存在しないことを検知した場合に、この実行を試行します。

3. 上記の変更内容で CRD をアップグレードします。
4. 後続のアップグレードサイクルでは、無効にされたバージョンを CRD から完全に削除できます。以下に例を示します。

```
versions:
  - name: v1beta1
    served: true
    storage: true
```

5. 該当バージョンが CRD から削除される場合、CSV の **owned** セクションにある CRD の参照バージョンも更新されていることを確認します。

5.6.10.4. 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、各カスタムリソース定義 (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 のリストで構成されます (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":"<operator_namespace>"},"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>"}}}]
```

5.6.10.5. 内部オブジェクトの非表示

Operator がタスクを実行するためにカスタムリソース定義 (CRD) を内部で使用する方法は一般的な方法です。これらのオブジェクトはユーザーが操作することが意図されていません。オブジェクトの操作により Operator のユーザーにとって混乱を生じさせる可能性があります。たとえば、データベース Operator には、ユーザーが **replication: true** で Database オブジェクトを作成する際に常に作成される **Replication** CRD が含まれる場合があります。

Operator の作成者は、**operators.operatorframework.io/internal-objects** アノテーションを Operator のクラスターサービスバージョン (CSV) に追加して、ユーザー操作を目的としていないユーザーインターフェイスの CRD を非表示にすることができます。

手順

1. CRD のいずれかに internal のマークを付ける前に、アプリケーションの管理に必要となる可能性のあるデバッグ情報または設定が CR のステータスまたは **spec** ブロックに反映されていることを確認してください (使用する Opearator に該当する場合)。

2. **operators.operatorframework.io/internal-objects** アノテーションを Operator の CSV に追加し、ユーザーインターフェイスで非表示にする内部オブジェクトを指定します。

   内部オブジェクトのトアノテーション

```
apiVersion: operators.coreos.com/v1alpha1
kind: ClusterServiceVersion
metadata:
  name: my-operator-v1.2.3
  annotations:
    operators.operatorframework.io/internal-objects:
'["my.internal.crd1.io","my.internal.crd2.io"]' (1)
...
```

(1) 内部 CRD を文字列の配列として設定します。

5.6.10.6. 必要なカスタムリソースの初期化

Operator では、ユーザーが Operator が完全に機能する前にカスタムリソースをインスタンス化する必要がある場合があります。ただし、ユーザーが必要な内容やリソースの定義方法を判断することが困難な場合があります。

Operator 開発者は、Operator のインストール中に **operatorframework.io/initialization-resource** をクラスターサービスバージョン (CSV) に追加することで、必要なカスタムリソースを 1 つ指定できます。次に、CSV で提供されるテンプレートを使用してカスタムリソースを作成するように求められます。アノテーションには、インストール時にリソースを初期化するために必要な完全な YAML 定義が含まれるテンプレートが含まれている必要があります。

このアノテーションが定義されている場合、ユーザーが OpenShift Dedicated から Operator をインストールすると、CSV で提供されるテンプレートを使用してリソースを作成するように求められます。

手順

- **operatorframework.io/initialization-resource** アノテーションを Operator の CSV に追加し、必要なカスタムリソースを指定します。たとえば、以下のアノテーションでは **StorageCluster** リソースの作成が必要であり、これは完全な YAML 定義を提供します。

初期化リソースアノテーション

```
apiVersion: operators.coreos.com/v1alpha1
kind: ClusterServiceVersion
metadata:
  name: my-operator-v1.2.3
  annotations:
    operatorframework.io/initialization-resource: |-
        {
            "apiVersion": "ocs.openshift.io/v1",
            "kind": "StorageCluster",
            "metadata": {
                "name": "example-storagecluster"
            },
            "spec": {
                "manageNodes": false,
                "monPVCTemplate": {
                    "spec": {
                        "accessModes": [
                            "ReadWriteOnce"
                        ],
                        "resources": {
                            "requests": {
                                "storage": "10Gi"
                            }
                        },
                        "storageClassName": "gp2"
                    }
                },
                "storageDeviceSets": [
                    {
                        "count": 3,
                        "dataPVCTemplate": {
                            "spec": {
                                "accessModes": [
                                    "ReadWriteOnce"
                                ],
                                "resources": {
                                    "requests": {
                                        "storage": "1Ti"
                                    }
                                },
                                "storageClassName": "gp2",
                                "volumeMode": "Block"
                            }
                        },
                        "name": "example-deviceset",
                        "placement": {},
                        "portable": true,
                        "resources": {}
```

```
            }
          ]
        }
      }
...
```

5.6.11. API サービスについて

CRD の場合のように、Operator が使用できる API サービスの 2 つのタイプ (所有 **(owned)** および 必須 **(required)**) があります。

5.6.11.1. 所有 API サービス

CSV が API サービスを所有する場合、CSV は API サービスおよびこれが提供する group/version/kind (GVK) をサポートする拡張 **api-server** のデプロイメントを記述します。

API サービスはこれが提供する 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リスト表示できます。

表5.16 所有 **API** サービスフィールド

フィールド	説明	必須/オプション
Group	API サービスが提供するグループ (**database.example.com** など)。	必須
Version	API サービスのバージョン (**v1alpha1** など)。	必須
Kind	API サービスが提供することが予想される種類。	必須
Name	指定された API サービスの複数形の名前	必須
DeploymentName	API サービスに対応する CSV で定義されるデプロイメントの名前 (所有 API サービスに必要)。CSV の保留フェーズに、OLM Operator は CSV の **InstallStrategy** で一致する名前を持つ **Deployment** 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	API サービス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 API サービスを使用する方法についての短い説明、または API サービスが提供する機能の説明。	必須

フィールド	説明	必須/オプション
Resources	API サービスは 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リスト表示されます。 この場合、オーケストレーションするすべてのリストではなく、重要なオブジェクトのみをリスト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設定マップをリスト表示しないでください。	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

5.6.11.1.1. API サービスリソースの作成

Operator Lifecycle Manager (OLM) はそれぞれ固有の所有 API サービスについてサービスおよび API サービスリソースを作成するか、これらを置き換えます。

- サービス Pod セレクターは API サービスの記述の **DeploymentName** フィールドに一致する 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 API サービスリソースに組み込まれます。

5.6.11.1.2. API サービス提供証明書

OLM は、所有 API サービス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 **Service** リソースのホスト名が含まれる一般名 (CN) が含まれ、これは対応する API サービス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、 API サービスの記述の **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ボリュームマウント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。生成されるボリュームマウントのパスは **/apiserver.local.config/certificates** にデフォルト設定され、同じパスの既存のボリュームマウントが置き換えられます。

5.6.11.2. 必要な API サービス

OLM は、必要なすべての CSV に利用可能な API サービスがあり、すべての予想される GVK 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 API サービスによって提供される特定の種類に依存できます。

表5.17 必須 API サービスフィールド

フィールド	説明	必須/オプション
Group	API サービスが提供するグループ (**database.example.com** など)。	必須
Version	API サービスのバージョン (**v1alpha1** など)。	必須
Kind	API サービスが提供することが予想される種類。	必須
DisplayName	API サービス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 API サービスを使用する方法についての短い説明、または API サービスが提供する機能の説明。	必須

5.7. バンドルイメージの使用

Operator Lifecycle Manager (OLM) で使用するためのバンドル形式で Operator をパッケージ化してデプロイし、アップグレードするには、Operator SDK を使用できます。

5.7.1. Operator のバンドル

Operator Bundle Format は、Operator SDK および Operator Lifecycle Manager (OLM) のデフォルトパッケージ方法です。Operator SDK を使用して OLM に対して Operator を準備し、バンドルイメージをとして Operator プロジェクトをビルドしてプッシュできます。

前提条件

- 開発ワークステーションに Operator SDK CLI がインストールされている。
- OpenShift CLI (**oc**) v4 以降がインストールされている。
- Operator プロジェクトが Operator SDK を使用して初期化されている。
- Operator が Go ベースの場合、OpenShift Dedicated で Operator を実行するために、サポート対象のイメージを使用するようにプロジェクトが更新されている。

手順

1. 以下の **make** コマンドを Operator プロジェクトディレクトリーで実行し、Operator イメージをビルドし、プッシュします。以下の手順の **IMG** 引数を変更して、アクセス可能なリポジトリーを参照します。Quay.io などのリポジトリーサイトにコンテナーを保存するためのアカウントを取得できます。

   a. イメージをビルドします。

   ```
   $ make docker-build IMG=<registry>/<user>/<operator_image_name>:<tag>
   ```

注記

Operator の SDK によって生成される Dockerfile は、**go build** について **GOARCH=amd64** を明示的に参照します。これは、AMD64 アーキテクチャー以外の場合は **GOARCH=$TARGETARCH** に修正できます。Docker は、**-platform** で指定された値に環境変数を自動的に設定します。Buildah では、そのために **-build-arg** を使用する必要があります。詳細は、Multiple Architectures を参照してください。

b. イメージをリポジトリーにプッシュします。

```
$ make docker-push IMG=<registry>/<user>/<operator_image_name>:<tag>
```

2. Operator SDK **generate bundle** および **bundle validate** のサブコマンドを含む複数のコマンドを呼び出す **make bundle** コマンドを実行し、Operator バンドルマニフェストを作成します。

```
$ make bundle IMG=<registry>/<user>/<operator_image_name>:<tag>
```

Operator のバンドルマニフェストは、アプリケーションを表示し、作成し、管理する方法を説明します。**make bundle** コマンドは、以下のファイルおよびディレクトリーを Operator プロジェクトに作成します。

- **ClusterServiceVersion** オブジェクトを含む **bundle/manifests** という名前のバンドルマニフェストディレクトリー
- **bundle/metadata** という名前のバンドルメタデータディレクトリー
- **config/crd** ディレクトリー内のすべてのカスタムリソース定義 (CRD)
- Dockerfile **bundle.Dockerfile**

続いて、これらのファイルは **operator-sdk bundle validate** を使用して自動的に検証され、ディスク上のバンドル表現が正しいことを確認します。

3. 以下のコマンドを実行し、バンドルイメージをビルドしてプッシュします。OLM は、1つ以上のバンドルイメージを参照するインデックスイメージを使用して Operator バンドルを使用します。

a. バンドルイメージをビルドします。イメージをプッシュしようとするレジストリー、ユーザー namespace、およびイメージタグの詳細で **BUNDLE_IMG** を設定します。

```
$ make bundle-build BUNDLE_IMG=<registry>/<user>/<bundle_image_name>:<tag>
```

b. バンドルイメージをプッシュします。

```
$ docker push <registry>/<user>/<bundle_image_name>:<tag>
```

5.7.2. Operator Lifecycle Manager を使用した Operator のデプロイ

Operator Lifecycle Manager (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、ライフサイクルを管理するのに役立ちます。OLM はデフォルトで OpenShift Dedicated にインストールされ、Kubernetes 拡張として実行されます。そのため、追加のツールなしで、すべての Operator ライフサイクル管理機能に Web コンソールと OpenShift CLI (**oc**) を使用できます。

Operator Bundle Format は、Operator SDK および OLM のデフォルトパッケージ方法です。Operator SDK を使用して OLM でバンドルイメージを迅速に実行し、適切に実行されるようにできます。

前提条件

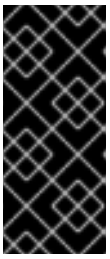
- 開発ワークステーションに Operator SDK CLI がインストールされている。
- Operator バンドルイメージがビルドされ、レジストリーにプッシュされている。
- OLM が Kubernetes ベースのクラスターにインストールされている (**apiextensions.k8s.io/v1** CRD を使用する場合は v1.16.0 以降、たとえば OpenShift Dedicated 4)
- **dedicated-admin** 権限を持つアカウントを使用して **oc** でクラスターにログインしている。
- Operator が Go ベースの場合、OpenShift Dedicated で Operator を実行するために、サポート対象のイメージを使用するようにプロジェクトが更新されている。

手順

- 以下のコマンドを入力してクラスターで Operator を実行します。

```
$ operator-sdk run bundle \ 1
  -n <namespace> \ 2
  <registry>/<user>/<bundle_image_name>:<tag> 3
```

1. **run bundle** コマンドは、有効なファイルベースのカタログを作成し、OLM を使用して Operator バンドルをクラスターにインストールします。
2. オプション: デフォルトで、このコマンドは **~/.kube/config** ファイルの現在アクティブなプロジェクトに Operator をインストールします。**-n** フラグを追加して、インストールに異なる namespace スコープを設定できます。
3. イメージを指定しない場合、コマンドは **quay.io/operator-framework/opm:latest** をデフォルトのインデックスイメージとして使用します。イメージを指定した場合は、コマンドはバンドルイメージ自体をインデックスイメージとして使用します。



重要

OpenShift Dedicated 4.11 以降、**run bundle** コマンドは Operator カタログのファイルベースのカタログ形式をデフォルトでサポートしています。Operator カタログに関して、非推奨の SQLite データベース形式は引き続きサポートされますが、今後のリリースで削除される予定です。Operator の作成者はワークフローをファイルベースのカタログ形式に移行することが推奨されます。

このコマンドにより、以下のアクションが行われます。

- バンドルイメージをインジェクトしてインデックスイメージを作成します。インデックスイメージは不透明で一時的なものですが、バンドルを実稼働環境でカタログに追加する方法を正確に反映します。
- 新規インデックスイメージを参照するカタログソースを作成します。これにより、OperatorHub が Operator を検出できるようになります。

- **OperatorGroup**、**Subscription**、**InstallPlan**、および RBAC を含むその他の必要なリソースすべてを作成して、Operator をクラスターにデプロイします。

関連情報

- Operator Framework パッケージ形式の ファイルベースのカタログ
- カスタムカタログの管理の ファイルベースのカタログ
- Bundle Format

5.7.3. バンドルされた Operator を含むカタログの公開

Operator をインストールおよび管理するには、Operator Lifecycle Manager (OLM) では、Operator バンドルがクラスターのカタログで参照されるインデックスイメージにリスト表示される必要があります。Operator の作成者は、Operator SDK を使用して Operator のバンドルおよびそれらのすべての依存関係を含むインデックスを作成できます。これは、リモートクラスターでのテストおよびコンテナーレジストリーへの公開に役立ちます。

注記

Operator SDK は **opm** CLI を使用してインデックスイメージの作成を容易にします。**opm** コマンドの経験は必要ありません。高度なユースケースでは、Operator SDK を使用せずに、**opm** コマンドを直接使用できます。

前提条件

- 開発ワークステーションに Operator SDK CLI がインストールされている。
- Operator バンドルイメージがビルドされ、レジストリーにプッシュされている。
- OLM が Kubernetes ベースのクラスターにインストールされている (**apiextensions.k8s.io/v1** CRD を使用する場合は v1.16.0 以降、たとえば OpenShift Dedicated 4)
- **dedicated-admin** 権限を持つアカウントを使用して **oc** でクラスターにログインしている。

手順

1. 以下の **make** コマンドを Operator プロジェクトディレクトリーで実行し、Operator バンドルを含むインデックスイメージをビルドします。

```
$ make catalog-build CATALOG_IMG=<registry>/<user>/<index_image_name>:<tag>
```

   ここでは、**CATALOG_IMG** 引数は、アクセス権限のあるリポジトリーを参照します。Quay.io などのリポジトリーサイトにコンテナーを保存するためのアカウントを取得できます。

2. ビルドしたインデックスイメージをリポジトリーにプッシュします。

```
$ make catalog-push CATALOG_IMG=<registry>/<user>/<index_image_name>:<tag>
```

ヒント

複数のアクションを順番にまとめて実行する場合には、Operator SDK の **make** コマンドを併用できます。たとえば、Operator プロジェクトのバンドルイメージをビルドしていない場合は、以下の構文でバンドルイメージとインデックスイメージの両方をビルドしてプッシュできます。

```
$ make bundle-build bundle-push catalog-build catalog-push \
    BUNDLE_IMG=<bundle_image_pull_spec> \
    CATALOG_IMG=<index_image_pull_spec>
```

または、**Makefile** の **IMAGE_TAG_BASE** フィールドを既存のリポジトリーに設定できます。

```
IMAGE_TAG_BASE=quay.io/example/my-operator
```

次に、以下の構文を使用して、バンドルイメージ用の **quay.io/example/my-operator-bundle:v0.0.1** および **quay.io/example/my-operator-catalog:v0.0.1** など、自動生成される名前でイメージをビルドおよびプッシュできます。

```
$ make bundle-build bundle-push catalog-build catalog-push
```

3. 生成したインデックスイメージを参照する **CatalogSource** オブジェクトを定義して、**oc apply** コマンドまたは Web コンソールを使用してオブジェクトを作成します。

CatalogSource YAML の例

```
apiVersion: operators.coreos.com/v1alpha1
kind: CatalogSource
metadata:
  name: cs-memcached
  namespace: <operator_namespace>
spec:
  displayName: My Test
  publisher: Company
  sourceType: grpc
  grpcPodConfig:
    securityContextConfig: <security_mode> 1
  image: quay.io/example/memcached-catalog:v0.0.1 2
  updateStrategy:
    registryPoll:
      interval: 10m
```

1. **legacy** または **restricted** の値を指定します。フィールドが設定されていない場合、デフォルト値は **legacy** です。今後の OpenShift Dedicated リリースでは、デフォルト値が **restricted** になる予定です。**restricted** 権限でカタログを実行できない場合は、このフィールドを手動で **legacy** に設定することを推奨します。

2. **CATALOG_IMG** 引数を使用して、**image** を以前に使用したイメージプル仕様に設定します。

4. カタログソースを確認します。

```
$ oc get catalogsource
```

出力例

```
NAME          DISPLAY     TYPE   PUBLISHER   AGE
cs-memcached   My Test     grpc   Company     4h31m
```

検証

1. カタログを使用して Operator をインストールします。
   a. **oc apply** コマンドまたは Web コンソールを使用して、**OperatorGroup** オブジェクトを定義して作成します。

      OperatorGroup YAML の例

      ```
      apiVersion: operators.coreos.com/v1
      kind: OperatorGroup
      metadata:
        name: my-test
        namespace: <operator_namespace>
      spec:
        targetNamespaces:
        - <operator_namespace>
      ```

   b. **oc apply** コマンドまたは Web コンソールを使用して、**Subscription** オブジェクトを定義して作成します。

      サブスクリプション **YAML の例**

      ```
      apiVersion: operators.coreos.com/v1alpha1
      kind: Subscription
      metadata:
        name: catalogtest
        namespace: <catalog_namespace>
      spec:
        channel: "alpha"
        installPlanApproval: Manual
        name: catalog
        source: cs-memcached
        sourceNamespace: <operator_namespace>
        startingCSV: memcached-operator.v0.0.1
      ```

2. インストールされた Operator が実行されていることを確認します。
   a. Operator グループを確認します。

      ```
      $ oc get og
      ```

      出力例

      ```
      NAME          AGE
      my-test       4h40m
      ```

b. クラスターサービスバージョン (CSV) を確認します。

```
$ oc get csv
```

出力例

```
NAME                    DISPLAY   VERSION   REPLACES   PHASE
memcached-operator.v0.0.1   Test      0.0.1             Succeeded
```

c. Operator の Pod を確認します。

```
$ oc get pods
```

出力例

```
NAME                                                READY   STATUS      RESTARTS   AGE
9098d908802769fbde8bd45255e69710a9f8420a8f3d814abe88b68f8ervdj6   0/1
Completed   0          4h33m
catalog-controller-manager-7fd5b7b987-69s4n                  2/2     Running     0
4h32m
cs-memcached-7622r                                  1/1     Running     0          4h33m
```

関連情報

- 高度なユースケースの **opm** CLI の直接使用に関する詳細は、カスタムカタログの管理 を参照してください。

5.7.4. Operator Lifecycle Manager での Operator アップグレードのテスト

インデックスイメージおよびカタログソースを手動で管理しなくても、Operator SDK で Operator Lifecycle Manager (OLM) 統合を使用して Operator のアップグレードを迅速にテストできます。

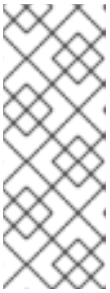
run bundle-upgrade サブコマンドは、より新しいバージョンのバンドルイメージを指定することにより、インストールされた Operator をトリガーしてそのバージョンにアップグレードするプロセスを自動化します。

前提条件

- **run bundle** サブコマンドを使用するか、従来の OLM インストールを使用して、Operator を OLM でと合わせてインストールしておく
- インストールされた Operator のより新しいバージョンを表すバンドルイメージ

手順

1. Operator が OLM でまだインストールしていない場合は、**run bundle** サブコマンドまたは従来の OLM インストールを使用して、以前のバージョンの Operator をインストールします。



注記

以前のバージョンのバンドルが従来 OLM を使用してインストールされている場合には、アップグレード予定の新しいバンドルは、カタログソースで参照されるインデックスイメージ内に含めることはできません。含めてしまっている場合には、**run bundle-upgrade** サブコマンドを実行すると、新しいバンドルがパッケージおよびクラスターサービスバージョン (CSV) を提供するインデックスですでに参照されているので、レジストリー Pod が失敗します。

たとえば、前述のバンドルイメージを指定して、Memcached Operator 用に以下の **run bundle** サブコマンドを使用できます。

```
$ operator-sdk run bundle <registry>/<user>/memcached-operator:v0.0.1
```

出力例

```
INFO[0006] Creating a File-Based Catalog of the bundle "quay.io/demo/memcached-operator:v0.0.1"
INFO[0008] Generated a valid File-Based Catalog
INFO[0012] Created registry pod: quay-io-demo-memcached-operator-v1-0-1
INFO[0012] Created CatalogSource: memcached-operator-catalog
INFO[0012] OperatorGroup "operator-sdk-og" created
INFO[0012] Created Subscription: memcached-operator-v0-0-1-sub
INFO[0015] Approved InstallPlan install-h9666 for the Subscription: memcached-operator-v0-0-1-sub
INFO[0015] Waiting for ClusterServiceVersion "my-project/memcached-operator.v0.0.1" to reach 'Succeeded' phase
INFO[0015] Waiting for ClusterServiceVersion ""my-project/memcached-operator.v0.0.1" to appear
INFO[0026] Found ClusterServiceVersion "my-project/memcached-operator.v0.0.1" phase: Pending
INFO[0028] Found ClusterServiceVersion "my-project/memcached-operator.v0.0.1" phase: Installing
INFO[0059] Found ClusterServiceVersion "my-project/memcached-operator.v0.0.1" phase: Succeeded
INFO[0059] OLM has successfully installed "memcached-operator.v0.0.1"
```

2. Operator のより新しいバージョンのバンドルイメージを指定して、インストールされた Operator をアップグレードします。

```
$ operator-sdk run bundle-upgrade <registry>/<user>/memcached-operator:v0.0.2
```

出力例

```
INFO[0002] Found existing subscription with name memcached-operator-v0-0-1-sub and namespace my-project
INFO[0002] Found existing catalog source with name memcached-operator-catalog and namespace my-project
INFO[0008] Generated a valid Upgraded File-Based Catalog
INFO[0009] Created registry pod: quay-io-demo-memcached-operator-v0-0-2
INFO[0009] Updated catalog source memcached-operator-catalog with address and annotations
INFO[0010] Deleted previous registry pod with name "quay-io-demo-memcached-operator-v0-0-1"
```

```
INFO[0041] Approved InstallPlan install-gvcjh for the Subscription: memcached-operator-v0-0-1-sub
INFO[0042] Waiting for ClusterServiceVersion "my-project/memcached-operator.v0.0.2" to reach 'Succeeded' phase
INFO[0019] Found ClusterServiceVersion "my-project/memcached-operator.v0.0.2" phase: Pending
INFO[0042] Found ClusterServiceVersion "my-project/memcached-operator.v0.0.2" phase: InstallReady
INFO[0043] Found ClusterServiceVersion "my-project/memcached-operator.v0.0.2" phase: Installing
INFO[0044] Found ClusterServiceVersion "my-project/memcached-operator.v0.0.2" phase: Succeeded
INFO[0044] Successfully upgraded to "memcached-operator.v0.0.2"
```

3. インストールされた Operator のクリーンアップ

```
$ operator-sdk cleanup memcached-operator
```

関連情報

- OLM を使用した 従来の Operator のインストール

5.7.5. OpenShift Dedicated のバージョンと Operator の互換性の管理

重要

Kubernetes は定期的に特定の API を非推奨とし、後続のリリースで削除します。Operator が非推奨の API を使用している場合、OpenShift Dedicated クラスターを API が削除された Kubernetes バージョンにアップグレードすると、Operator が機能しなくなる可能性があります。

Operator の作成者は、Kubernetes ドキュメントの 旧版の API 移行ガイド を確認し、非推奨および削除済みの API が使用されないように Operator プロジェクトを最新の状態に維持することが強く推奨されます。可能な限り、Operator の互換性が失われる今後のバージョンの OpenShift Dedicated がリリースされる前に、Operator を更新してください。

API が OpenShift Dedicated のバージョンから削除されると、削除された API をまだ使用しているクラスターバージョンで実行されている Operator が正しく動作しなくなります。Operator の作成者は、Operator ユーザーの中断を回避するために、API の非推奨および削除に対応するように Operator プロジェクトを更新する計画を立てる必要があります。

ヒント

Operator のイベントアラートを確認して、現在使用中の API に関する警告があるかどうかをチェックできます。次のリリースで削除される API が検出されると、以下のアラートが表示されます。

APIRemovedInNextReleaseInUse

次の OpenShift Dedicated リリースで削除される API。

APIRemovedInNextEUSReleaseInUse

次の OpenShift Dedicated Extended Update Support (EUS) リリースで削除される API。

クラスター管理者は Operator をインストールしたら、OpenShift Dedicated の次のバージョンにアップグレードする前に、その次のクラスターのバージョンと互換性のある Operator のバージョンがインストールされていることを確認する必要があります。非推奨の API や削除された API を使用しないように Operator プロジェクトを更新することを推奨します。それでも削除された API を含む Operator バンドルを公開する必要がある場合は、OpenShift Dedicated で引き続き使用できるように、バンドルが適切に設定されていることを確認してください。

次の手順は、管理者が互換性のないバージョンの OpenShift Dedicated に Operator のバージョンをインストールするのを回避するのに役立ちます。この手順を実施すると、クラスターに現在インストールされている Operator のバージョンと互換性のない、新しいバージョンの OpenShift Dedicated にアップグレードすることも回避できます。

この手順は、Operator の現在のバージョンが何らかの理由で特定の OpenShift Dedicated バージョンで正常に動作しないことがわかっている場合にも役立ちます。Operator の配信先のクラスターバージョンを定義することで、許可された範囲外のクラスターバージョンのカタログに Operator が表示されないようにします。

重要

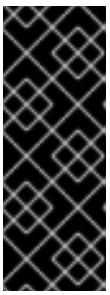
非推奨の API を使用する Operator があると、API がサポートされなくなった OpenShift Dedicated の将来のバージョンにクラスター管理者がアップグレードしたときに、重要なワークロードに悪影響が及ぶ可能性があります。Operator が非推奨の API を使用している場合は、できるだけ早く Operator プロジェクトで以下の設定を指定する必要があります。

前提条件

- 既存の Operator プロジェクト

手順

1. Operator の特定のバンドルがサポートされておらず、特定のクラスターバージョンよりも後の OpenShift Dedicated で正しく動作しないことがわかっている場合は、Operator と互換性のある OpenShift Dedicated の最新バージョンを設定します。Operator プロジェクトのクラスターサービスバージョン (CSV) で **olm.maxOpenShiftVersion** アノテーションを設定して、インストールされている Operator を互換性のあるバージョンにアップグレードする前に、管理者がクラスターをアップグレードできないようにします。



重要

Operator バンドルバージョンが新しいバージョンで機能しない場合にのみ、**olm.maxOpenShiftVersion** アノテーションを使用する必要があります。クラスター管理者は、ソリューションがインストールされている状態でクラスターをアップグレードできないことに注意してください。新しいバージョンおよび有効なアップグレードパスを指定しない場合、管理者は Operator をアンインストールし、クラスターのバージョンをアップグレードできます。

olm.maxOpenShiftVersion アノテーションを含む CSV の例

```
apiVersion: operators.coreos.com/v1alpha1
kind: ClusterServiceVersion
metadata:
  annotations:
    "olm.properties": '[{"type": "olm.maxOpenShiftVersion", "value": "<cluster_version>"}]' 1
```

1. Operator と互換性がある OpenShift Dedicated の最新のクラスターバージョンを指定します。たとえば、**value** を **4.9** に設定すると、このバンドルがクラスターにインストールされている場合、クラスターを 4.9 より後の OpenShift Dedicated バージョンにアップグレードできなくなります。

2. バンドルを Red Hat 提供の Operator カタログで配布する場合は、次のプロパティーを設定して、互換性のあるバージョンの OpenShift Dedicated を Operator に合わせて設定します。この設定により、OpenShift Dedicated の互換性のあるバージョンを対象とするカタログにのみ Operator が含まれるようになります。

注記

この手順は、Red Hat が提供するカタログに Operator を公開する場合にのみ有効です。バンドルがカスタムカタログのディストリビューションのみを目的としている場合には、この手順を省略できます。詳細は、Red Hat が提供する Operator カタログについてを参照してください。

a. プロジェクトの **bundle/metadata/annotations.yaml** ファイルに **com.redhat.openshift.versions** アノテーションを設定します。

互換性のあるバージョンを含む bundle/metadata/annotations.yaml ファイルの例

```
com.redhat.openshift.versions: "v4.7-v4.9" 1
```

1. 範囲または単一バージョンに設定します。

b. バンドルが互換性のないバージョンの OpenShift Dedicated に引き継がれないようにするには、Operator のバンドルイメージで適切な **com.redhat.openshift.versions** ラベルを使用してインデックスイメージが生成されていることを確認します。たとえば、プロジェクトが Operator SDK を使用して生成された場合は、**bundle.Dockerfile** ファイルを更新してください。

互換性のあるバージョンを含む bundle.Dockerfile の例

```
LABEL com.redhat.openshift.versions="<versions>" 1
```

1. 範囲または単一バージョンに設定します (例: **v4.7-v4.9**)。この設定は、Operator を配信する必要があるクラスターのバージョンを定義し、Operator は、範囲外にあるクラスターバージョンのカタログに表示されません。

Operator の新規バージョンをバンドルして、更新バージョンをカタログに公開して配布できるようになりました。

関連情報

- **Certified Operator Build Guide** の Managing OpenShift Versions
- インストール済み Operator の更新
- Red Hat が提供する Operator カタログ

5.7.6. 関連情報

- バンドル形式の詳細は、Operator Framework パッケージ形式 を参照してください。
- **opm** コマンドを使用してバンドルイメージをインデックスイメージに追加する方法の詳細は、カスタムカタログの管理 を参照してください。
- インストールされた Operator のアップグレードの仕組みについての詳細は、Operator Lifecycle Manager ワークフロー を参照してください。

5.8. POD セキュリティーアドミッションに準拠

Pod セキュリティーアドミッション は、Kubernetes Pod セキュリティー標準 の実装です。Pod のセキュリティーアドミッション は Pod の動作を制限します。グローバルまたは namespace レベルで定義された Pod のセキュリティーアドミッションに準拠していない Pod は、クラスターへの参加が許可されず、実行できません。

Operator プロジェクトの実行に昇格された権限が必要ない場合は、**restricted** Pod セキュリティーレベルに設定された namespace でワークロードを実行できます。Operator プロジェクトの実行に昇格された権限が必要な場合は、次のセキュリティーコンテキスト設定を設定する必要があります。

- Operator の namespace に対して許可される Pod セキュリティーアドミッションレベル
- ワークロードのサービスアカウントに許可されるセキュリティーコンテキスト制約 (SCC)

詳細は、Pod セキュリティーアドミッションの理解と管理 を参照してください。

5.8.1. Pod セキュリティーアドミッションについて

OpenShift Dedicated には、Kubernetes Pod のセキュリティーアドミッション が組み込まれています。グローバルまたは namespace レベルで定義された Pod のセキュリティーアドミッションに準拠していない Pod は、クラスターへの参加が許可されず、実行できません。

グローバルに、**privileged** プロファイルが適用され、**restricted** プロファイルが警告と監査に使用されます。

Pod のセキュリティーアドミッション設定を namespace レベルで設定することもできます。

重要

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

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

5.8.1.1. Pod のセキュリティーアドミッションモード

namespace に対して次の Pod セキュリティーアドミッションモードを設定できます。

表5.18 Pod のセキュリティーアドミッションモード

モード	ラベル	説明
enforce	**pod-security.kubernetes.io/enforce**	設定されたプロファイルに準拠していない Pod の受け入れを拒否します。
audit	**pod-security.kubernetes.io/audit**	Pod が設定されたプロファイルに準拠していない場合、監査イベントをログに記録します。
warn	**pod-security.kubernetes.io/warn**	Pod が設定されたプロファイルに準拠していない場合に警告を表示します。

5.8.1.2. Pod のセキュリティーアドミッションプロファイル

各 Pod セキュリティーアドミッションモードを次のプロファイルのいずれかに設定できます。

表5.19 Pod のセキュリティーアドミッションプロファイル

プロファイル	説明
privileged	最も制限の少ないポリシー。既知の権限昇格が可能になる
baseline	最小限の制限ポリシー。既知の権限昇格を防止する
restricted	最も制限的なポリシー。現在の Pod 強化のベストプラクティスに従う

5.8.1.3. 特権付きの namespace

次のシステム namespace は、常に **privileged** Pod セキュリティーアドミッションプロファイルに設定されます。

- **default**
- **kube-public**
- **kube-system**

これらの特権付き namespace の Pod セキュリティープロファイルを変更することはできません。

5.8.2. Pod セキュリティーアドミッション同期について

グローバル Pod セキュリティーアドミッションコントロール設定に加えて、コントローラーは、特定の namespace にあるサービスアカウントの SCC アクセス許可に従って、Pod セキュリティーアドミッションコントロールの **warn** および **audit** ラベルを namespace に適用します。

コントローラーは **ServiceAccount** オブジェクトのアクセス許可を確認して、各 namespace でセキュリティーコンテキストの制約を使用します。セキュリティーコンテキスト制約 (SCC) は、フィールド値に基づいて Pod セキュリティープロファイルにマップされます。コントローラーはこれらの変換さ

れたプロファイルを使用します。Pod のセキュリティー許可 **warn** と **audit** ラベルは、Pod の作成時に警告が表示されたり、監査イベントが記録されたりするのを防ぐために、namespace で最も特権のある Pod セキュリティープロファイルに設定されます。

namespace のラベル付けは、namespace ローカルサービスアカウントの権限を考慮して行われます。

Pod を直接適用すると、Pod を実行するユーザーの SCC 権限が使用される場合があります。ただし、自動ラベル付けではユーザー権限は考慮されません。

5.8.2.1. Pod セキュリティーアドミッション同期の namespace の除外

Pod セキュリティーアドミッション同期は、システムで作成された namespace および **openshift-*** 接頭辞が付いた namespace では永続的に無効になります。

クラスターペイロードの一部として定義されている namespace では、Pod セキュリティーアドミッションの同期が完全に無効になっています。次の namespace は永続的に無効になります。

- **default**
- **kube-node-lease**
- **kube-system**
- **kube-public**
- **openshift**
- **openshift-** という接頭辞が付いた、システムによって作成されたすべての namespace

5.8.3. Operator ワークロードが制限付き Pod セキュリティーレベルに設定された namespace で実行されるようにする

Operator プロジェクトがさまざまなデプロイメントおよび環境で確実に実行できるようにするには、**restricted** Pod セキュリティーレベルに設定された namespace で実行するように Operator のワークロードを設定します。

警告

runAsUser フィールドは空のままにしておく必要があります。イメージに特定のユーザーが必要な場合、制限付きセキュリティーコンテキスト制約 (SCC) および制限付き Pod セキュリティー適用の下ではイメージを実行できません。

手順

- **restricted** Pod セキュリティーレベルに設定された namespace で実行されるように Operator ワークロードを設定するには、次の例のように Operator の namespace 定義を編集します。

重要

Operator の namespace 定義で seccomp プロファイルを設定することが推奨されます。ただし、seccomp プロファイルの設定は OpenShift Dedicated 4.10 ではサポートされていません。

- OpenShift Dedicated 4.11 以降でのみ実行する必要がある Operator プロジェクトの場合は、次の例のように Operator の namespace 定義を編集します。

  config/manager/manager.yaml ファイル例

```
...
spec:
 securityContext:
   seccompProfile:
     type: RuntimeDefault (1)
   runAsNonRoot: true
 containers:
   - name: <operator_workload_container>
     securityContext:
       allowPrivilegeEscalation: false
       capabilities:
         drop:
           - ALL
...
```

  (1) seccomp プロファイルタイプを **RuntimeDefault** に設定すると、SCC はデフォルトで namespace の Pod セキュリティープロファイルになります。

- OpenShift Dedicated 4.10 でも実行する必要がある Operator プロジェクトの場合は、次の例のように Operator のnamespace 定義を編集します。

  config/manager/manager.yaml ファイル例

```
...
spec:
 securityContext: (1)
   runAsNonRoot: true
 containers:
   - name: <operator_workload_container>
     securityContext:
       allowPrivilegeEscalation: false
       capabilities:
         drop:
           - ALL
...
```

  (1) seccomp プロファイルタイプを未設定のままにすると、Operator プロジェクトを OpenShift Dedicated 4.10 で実行できるようになります。

関連情報

- Security Context Constraints の管理

5.8.4. エスカレーションされた権限を必要とする Operator ワークロードの Pod セキュリティーアドミッションの管理

Operator プロジェクトの実行に昇格されたアクセスパーミッションが必要な場合は、Operator のクラスターサービスバージョン (CSV) を編集する必要があります。

手順

1. 次の例のように、Operator の CSV でセキュリティーコンテキスト設定を必要なパーミッションレベルに設定します。

   ネットワーク管理者権限を持つ <operator_name>.clusterserviceversion.yaml ファイルの例

```
...
containers:
  - name: my-container
    securityContext:
      allowPrivilegeEscalation: false
      capabilities:
        add:
          - "NET_ADMIN"
...
```

2. 次の例のように、Operator のワークロードが必要なセキュリティーコンテキスト制約 (SCC) を使用できるようにするサービスアカウント権限を設定します。

   <operator_name>.clusterserviceversion.yaml ファイルの例

```
...
  install:
    spec:
      clusterPermissions:
      - rules:
        - apiGroups:
          - security.openshift.io
          resourceNames:
          - privileged
          resources:
          - securitycontextconstraints
          verbs:
          - use
        serviceAccountName: default
...
```

3. Operator の CSV 説明を編集して、Operator プロジェクトに次の例のような昇格された権限が必要な理由を説明します。

   <operator_name>.clusterserviceversion.yaml ファイルの例

```
...
spec:
```

```
  apiservicedefinitions:{}
  ...
description: The <operator_name> requires a privileged pod security admission label set on
the Operator's namespace. The Operator's agents require escalated permissions to restart
the node if the node needs remediation.
```

5.8.5. 関連情報

- Pod セキュリティーアドミッションの理解と管理

5.9. スコアカードツールを使用した OPERATOR の検証

Operator の作成者は、Operator SDK でスコアカードツールを使用して以下のタスクを実行できます。

- Operator プロジェクトに構文エラーがなく、正しくパッケージ化されていることを確認します。
- Operator を強化する方法についての提案を確認します。

5.9.1. スコアカードツールについて

Operator SDK **bundle validate** サブコマンドは、コンテンツおよび構造のローカルバンドルディレクトリーおよびリモートバンドルイメージを検証することができますが、**scorecard** コマンドを使用して設定ファイルおよびテストイメージに基づいて Operator でテストを実行できます。これらのテストは、スコアカードによって実行されるよう設定され、ビルドされるテストイメージ内に実装されます。

スコアカードは、OpenShift Dedicated などの設定済みの Kubernetes クラスターにアクセスして実行されることを前提としています。スコアカードは Pod 内で各テストを実行します。これにより Pod ログが集計され、テスト結果はコンソールに送信されます。スコアカードにはビルトインの基本的なテストおよび Operator Lifecycle Manager (OLM) テストがあり、カスタムテスト定義を実行する手段も提供します。

スコアカードのワークフロー

1. 関連するカスタムリソース (CR) および Operator に必要なすべてのリソースを作成する
2. プロキシーコンテナーを Operator のデプロイメントに作成し、API サーバーへの呼び出しを記録してテストを実行する
3. CR のパラメーターを検査する

スコアカードテストは、テスト中の Operator の状態を想定しません。Operator の作成および Operator の CR の作成は、スコアカード自体では扱っていません。ただし、スコアカードテストは、テストがリソース作成用に設計されている場合は、必要なリソースをなんでも作成できます。

scorecard コマンド構文

```
$ operator-sdk scorecard <bundle_dir_or_image> [flags]
```

スコアカードには、Operator バンドルへのディスク上のパスまたはバンドルイメージの名前のいずれかの位置引数が必要です。

フラグの詳細については、以下を実行します。

```
$ operator-sdk scorecard -h
```

5.9.2. スコアカードの設定

スコアカードツールでは、内部プラグインの設定を可能にする設定と、複数のグローバル設定オプションを使用します。テストは、**config.yaml** という名前の設定ファイルによって実行されます。これは、**bundle/** ディレクトリーにある **make bundle** コマンドによって生成されます。

```
./bundle
...
└── tests
    └── scorecard
        └── config.yaml
```

スコアカード設定ファイルの例

```
kind: Configuration
apiversion: scorecard.operatorframework.io/v1alpha3
metadata:
  name: config
stages:
- parallel: true
  tests:
  - image: quay.io/operator-framework/scorecard-test:v1.31.0
    entrypoint:
    - scorecard-test
    - basic-check-spec
    labels:
      suite: basic
      test: basic-check-spec-test
  - image: quay.io/operator-framework/scorecard-test:v1.31.0
    entrypoint:
    - scorecard-test
    - olm-bundle-validation
    labels:
      suite: olm
      test: olm-bundle-validation-test
```

設定ファイルは、スコアカードが実行可能な各テストを定義します。スコアカード設定ファイルの以下のフィールドは、以下のようにテストを定義します。

設定フィールド	説明
image	テストを実装するコンテナーイメージ名のテスト
entrypoint	テストを実行するために、テストイメージで呼び出されるコマンドおよび引数
labels	実行するテストを選択するスコアカードで定義されたラベルまたはカスタムラベル

5.9.3. ビルトインスコアカードのテスト

スコアカードには、スイート (基本的なテストスイートおよび Operator Lifecycle Manager (OLM) スイート) に編成される事前に定義されたテストが同梱されます。

表5.20 基本的なテストスイート

テスト	説明	短縮名
Spec Block Exists	このテストは、クラスターで作成されたカスタムリソース (CR) をチェックし、すべての CR に **spec** ブロックがあることを確認します。	**basic-check-spec-test**

表5.21 OLM テストスイート

テスト	説明	短縮名
Bundle Validation	このテストは、スコアカードに渡されるバンドルにあるバンドルマニフェストを検証します。バンドルの内容にエラーが含まれる場合、テスト結果の出力には検証ログと検証ライブラリーからのエラーメッセージが含まれます。	**olm-bundle-validation-test**
Provided APIs Have Validation	このテストは、提供された CR のカスタムリソース定義 (CRD) に検証セクションが含まれ、CR で検出される各 **spec** および **status** フィールドの検証があることを確認します。	**olm-crds-have-validation-test**
Owned CRDs Have Resources Listed	このテストでは、**cr-manifest** オプションが提供する各 CR の CRD に、ClusterServiceVersion (CSV) の **owned** CRD セクションの **resources** サブセクションがあることを確認します。テストでリソースセクションにリスト表示されていない使用済みのリソースを検出する場合、テストの最後にある提案にそれらのリソースをリスト表示します。このテストが合格となるには、初回のコード生成後に、resources セクションを記入する必要があります。	**olm-crds-have-resources-test**
Spec Fields With Descriptors	このテストは、CR の **spec** セクションのすべてのフィールドに、CSV にリスト表示される対応する記述子があることを確認します。	**olm-spec-descriptors-test**
Status Fields With Descriptors	このテストは、CR の **status** セクションのすべてのフィールドに、CSV にリスト表示される対応する記述子があることを確認します。	**olm-status-descriptors-test**

5.9.4. スコアカードツールの実行

Kustomize ファイルのデフォルトセットは、**init** コマンドの実行後に Operator SDK によって生成されます。生成されるデフォルトの **bundle/tests/scorecard/config.yaml** ファイルは、Operator に対してスコアカードツールを実行するためにすぐに使用できます。または、このファイルをテスト仕様に変更することができます。

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こと。

手順

1. Operator のバンドルマニフェストおよびメタデータを生成または再生成します。

```
$ make bundle
```

このコマンドは、テストを実行するために **scorecard** コマンドが使用するバンドルメタデータに、スコアカードアノテーションを自動的に追加します。

2. Operator バンドルへのディスク上のパスまたはバンドルイメージの名前に対してスコアカードを実行します。

```
$ operator-sdk scorecard <bundle_dir_or_image>
```

5.9.5. スコアカードの出力

scorecard コマンドの **--output** フラグは、スコアカード結果の出力形式 (**text** または **json**) を指定します。

例5.7 JSON 出力スニペットの例


```
{
  "apiVersion": "scorecard.operatorframework.io/v1alpha3",
  "kind": "TestList",
  "items": [
    {
      "kind": "Test",
      "apiVersion": "scorecard.operatorframework.io/v1alpha3",
      "spec": {
        "image": "quay.io/operator-framework/scorecard-test:v1.31.0",
        "entrypoint": [
          "scorecard-test",
          "olm-bundle-validation"
        ],
        "labels": {
          "suite": "olm",
          "test": "olm-bundle-validation-test"
        }
      },
      "status": {
        "results": [
          {
            "name": "olm-bundle-validation",
            "log": "time=\"2020-06-10T19:02:49Z\" level=debug msg=\"Found manifests directory\"
name=bundle-test\ntime=\"2020-06-10T19:02:49Z\" level=debug msg=\"Found metadata
directory\" name=bundle-test\ntime=\"2020-06-10T19:02:49Z\" level=debug msg=\"Getting
mediaType info from manifests directory\" name=bundle-test\ntime=\"2020-06-10T19:02:49Z\"
level=info msg=\"Found annotations file\" name=bundle-test\ntime=\"2020-06-10T19:02:49Z\"
level=info msg=\"Could not find optional dependencies file\" name=bundle-test\n",
            "state": "pass"
          }
```

```
        ]
      }
    }
  ]
}
```

例5.8 テキスト出力スニペットの例

```
-----------------------------------------------------------------------------
Image:      quay.io/operator-framework/scorecard-test:v1.31.0
Entrypoint: [scorecard-test olm-bundle-validation]
Labels:
 "suite":"olm"
 "test":"olm-bundle-validation-test"
Results:
 Name: olm-bundle-validation
 State: pass
 Log:
  time="2020-07-15T03:19:02Z" level=debug msg="Found manifests directory" name=bundle-test
  time="2020-07-15T03:19:02Z" level=debug msg="Found metadata directory" name=bundle-test
  time="2020-07-15T03:19:02Z" level=debug msg="Getting mediaType info from manifests
directory" name=bundle-test
  time="2020-07-15T03:19:02Z" level=info msg="Found annotations file" name=bundle-test
  time="2020-07-15T03:19:02Z" level=info msg="Could not find optional dependencies file"
name=bundle-test
```

注記

出力形式仕様は **Test** タイプのレイアウトに一致します。

5.9.6. テストの選択

スコアカードテストは、**--selector** CLI フラグをラベル文字列のセットに設定して選択されます。セレクターフラグが指定されていない場合は、スコアカード設定ファイル内のすべてのテストが実行されます。

テストは、テスト結果がスコアカードによって集計され、標準出力 (**stdout**) に書き込まれる形で連続的に実行されます。

手順

1. **basic-check-spec-test** などの単一のテストを選択するには、**--selector** フラグを使用してテストを指定します。

```
$ operator-sdk scorecard <bundle_dir_or_image> \
    -o text \
    --selector=test=basic-check-spec-test
```

2. テストのスイートを選択するには (例: **olm**)、すべての OLM テストで使用されるラベルを指定します。

```
$ operator-sdk scorecard <bundle_dir_or_image> \
    -o text \
    --selector=suite=olm
```

3. 複数のテストを選択するには、以下の構文を使用して **selector** フラグを使用し、テスト名を指定します。

```
$ operator-sdk scorecard <bundle_dir_or_image> \
    -o text \
    --selector='test in (basic-check-spec-test,olm-bundle-validation-test)'
```

5.9.7. 並列テストの有効化

Operator の作成者は、スコアカード設定ファイルを使用して、テスト用の個別のステージを定義できます。ステージは、設定ファイルで定義されている順序で順次実行します。ステージには、テストのリストと設定可能な **parallel** 設定が含まれます。

デフォルトで、またはステージが明示的に **parallel** を **false** に設定する場合は、ステージのテストは、設定ファイルで定義されている順序で順次実行されます。テストを一度に 1 つずつ実行することは、2 つのテストが対話したり、互いに競合したりしないことを保証する際に役立ちます。

ただし、テストが完全に分離されるように設計されている場合は、並列化することができます。

手順

- 分離されたテストのセットを並行して実行するには、これらを同じステージに追加して、**parallel** を **true** に設定します。

```
apiVersion: scorecard.operatorframework.io/v1alpha3
kind: Configuration
metadata:
  name: config
stages:
- parallel: true (1)
  tests:
  - entrypoint:
    - scorecard-test
    - basic-check-spec
    image: quay.io/operator-framework/scorecard-test:v1.31.0
    labels:
      suite: basic
      test: basic-check-spec-test
  - entrypoint:
    - scorecard-test
    - olm-bundle-validation
    image: quay.io/operator-framework/scorecard-test:v1.31.0
    labels:
      suite: olm
      test: olm-bundle-validation-test
```

(1) 並列テストを有効にします。

並列ステージのすべてのテストは同時に実行され、スコアカードはすべてが完了するのを待ってから次のステージへ進みます。これにより、非常に迅速にテストが実行されます。

5.9.8. カスタムスコアカードのテスト

スコアカードツールは、以下の義務付けられた規則に従うカスタムテストを実行できます。

- テストはコンテナーイメージ内に実装されます。
- テストは、コマンドおよび引数を含むエントリーポイントを受け入れます。
- テストは、テスト出力に不要なロギングがない JSON 形式で、**v1alpha3** スコアカード出力を生成します。
- テストは、/**bundle** の共有マウントポイントでバンドルコンテンツを取得できます。
- テストは、クラスター内のクライアント接続を使用して Kubernetes API にアクセスできます。

テストイメージが上記のガイドラインに従う場合は、他のプログラミング言語でカスタムテストを作成することができます。

以下の例は、Go で書かれたカスタムテストイメージを示しています。

例5.9 カスタムスコアカードテストの例

```
// Copyright 2020 The Operator-SDK Authors
//
// Licensed under the Apache License, Version 2.0 (the "License");
// you may not use this file except in compliance with the License.
// You may obtain a copy of the License at
//
//     http://www.apache.org/licenses/LICENSE-2.0
//
// Unless required by applicable law or agreed to in writing, software
// distributed under the License is distributed on an "AS IS" BASIS,
// WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
// See the License for the specific language governing permissions and
// limitations under the License.

package main

import (
 "encoding/json"
 "fmt"
 "log"
 "os"

 scapiv1alpha3 "github.com/operator-framework/api/pkg/apis/scorecard/v1alpha3"
 apimanifests "github.com/operator-framework/api/pkg/manifests"
)

// This is the custom scorecard test example binary
// As with the Redhat scorecard test image, the bundle that is under
// test is expected to be mounted so that tests can inspect the
// bundle contents as part of their test implementations.
// The actual test is to be run is named and that name is passed
```

```
// as an argument to this binary.  This argument mechanism allows
// this binary to run various tests all from within a single
// test image.

const PodBundleRoot = "/bundle"

func main() {
 entrypoint := os.Args[1:]
 if len(entrypoint) == 0 {
  log.Fatal("Test name argument is required")
 }

 // Read the pod's untar'd bundle from a well-known path.
 cfg, err := apimanifests.GetBundleFromDir(PodBundleRoot)
 if err != nil {
  log.Fatal(err.Error())
 }

 var result scapiv1alpha3.TestStatus

 // Names of the custom tests which would be passed in the
 // `operator-sdk` command.
 switch entrypoint[0] {
 case CustomTest1Name:
  result = CustomTest1(cfg)
 case CustomTest2Name:
  result = CustomTest2(cfg)
 default:
  result = printValidTests()
 }

 // Convert scapiv1alpha3.TestResult to json.
 prettyJSON, err := json.MarshalIndent(result, "", "    ")
 if err != nil {
  log.Fatal("Failed to generate json", err)
 }
 fmt.Printf("%s\n", string(prettyJSON))

}

// printValidTests will print out full list of test names to give a hint to the end user on what the valid
tests are.
func printValidTests() scapiv1alpha3.TestStatus {
 result := scapiv1alpha3.TestResult{}
 result.State = scapiv1alpha3.FailState
 result.Errors = make([]string, 0)
 result.Suggestions = make([]string, 0)

 str := fmt.Sprintf("Valid tests for this image include: %s %s",
  CustomTest1Name,
  CustomTest2Name)
 result.Errors = append(result.Errors, str)
 return scapiv1alpha3.TestStatus{
  Results: []scapiv1alpha3.TestResult{result},
 }
}
```

```
const (
 CustomTest1Name = "customtest1"
 CustomTest2Name = "customtest2"
)

// Define any operator specific custom tests here.
// CustomTest1 and CustomTest2 are example test functions. Relevant operator specific
// test logic is to be implemented in similarly.

func CustomTest1(bundle *apimanifests.Bundle) scapiv1alpha3.TestStatus {
 r := scapiv1alpha3.TestResult{}
 r.Name = CustomTest1Name
 r.State = scapiv1alpha3.PassState
 r.Errors = make([]string, 0)
 r.Suggestions = make([]string, 0)
 almExamples := bundle.CSV.GetAnnotations()["alm-examples"]
 if almExamples == "" {
  fmt.Println("no alm-examples in the bundle CSV")
 }

 return wrapResult(r)
}

func CustomTest2(bundle *apimanifests.Bundle) scapiv1alpha3.TestStatus {
 r := scapiv1alpha3.TestResult{}
 r.Name = CustomTest2Name
 r.State = scapiv1alpha3.PassState
 r.Errors = make([]string, 0)
 r.Suggestions = make([]string, 0)
 almExamples := bundle.CSV.GetAnnotations()["alm-examples"]
 if almExamples == "" {
  fmt.Println("no alm-examples in the bundle CSV")
 }
 return wrapResult(r)
}

func wrapResult(r scapiv1alpha3.TestResult) scapiv1alpha3.TestStatus {
 return scapiv1alpha3.TestStatus{
  Results: []scapiv1alpha3.TestResult{r},
 }
}
```

5.10. OPERATOR バンドルの検証

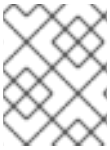
Operator の作成者は、Operator SDK で **bundle validate** コマンドを実行して Operator バンドルのコンテンツおよび形式を検証できます。リモート Operator バンドルイメージまたはローカル Operator バンドルディレクトリーでコマンドを実行できます。

5.10.1. bundle validate コマンドについて

Operator SDK **scorecard** コマンドは設定ファイルおよびテストイメージに基づいて Operator でテストを実行できますが、**bundle validate** サブコマンドは、ローカルバンドルディレクトリーおよびリモートバンドルイメージのコンテンツおよび構造を検証できます。

bundle validate コマンドの構文

```
$ operator-sdk bundle validate <bundle_dir_or_image> <flags>
```



注記

bundle validate コマンドは、**make bundle** コマンドを使用してバンドルをビルドすると自動的に実行されます。

バンドルイメージはリモートレジストリーからプルされ、検証前にローカルにビルドされます。ローカルバンドルディレクトリーには Operator メタデータおよびマニフェストが含まれている必要があります。バンドルメタデータとマニフェストには、以下のバンドルレイアウトと同様の構造が必要です。

バンドルレイアウトの例

```
./bundle
├── manifests
│   ├── cache.my.domain_memcacheds.yaml
│   └── memcached-operator.clusterserviceversion.yaml
└── metadata
    └── annotations.yaml
```

エラーが検出されない場合、バンドルテストは検証に合格し、終了コード **0** で終了します。

出力例

```
INFO[0000] All validation tests have completed successfully
```

エラーが検出されると、テストは検証に失敗し、終了コード **1** で終了します。

出力例

```
ERRO[0000] Error: Value cache.example.com/v1alpha1, Kind=Memcached: CRD
"cache.example.com/v1alpha1, Kind=Memcached" is present in bundle "" but not defined in CSV
```

警告が含まれるバンドルテストは、エラーが検出されていない限り、終了コード **0** で検証を終了することができます。テストはエラーが発生した場合にのみ失敗します。

出力例

```
WARN[0000] Warning: Value : (memcached-operator.v0.0.1) annotations not found
INFO[0000] All validation tests have completed successfully
```

bundle validate サブコマンドについての詳細を確認するには、以下のコマンドを実行してください。

```
$ operator-sdk bundle validate -h
```

5.10.2. ビルトインのバンドル検証テスト

Operator SDK には、スイートに編成された事前定義済みのバリデーターが同梱されています。バリデーターを指定せずに **bundle validate** コマンドを実行すると、デフォルトのテストが実行されます。

デフォルトテストは、バンドルが Operator Framework コミュニティーによって定義された仕様に準拠していることを確認します。詳細は、Bundle format を参照してください。

OperatorHub の互換性や非推奨の Kubernetes API などの問題の有無をテストするために、オプションのバリデーターを実行できます。オプションバリデーターは、必ずデフォルトのテストに追加して実行されます。

オプションのテストスイートの bundle validate コマンドの構文

```
$ operator-sdk bundle validate <bundle_dir_or_image>
  --select-optional <test_label>
```

表5.22 追加の **bundle validate** バリデーター

名前	説明	ラベル
Operator Framework	このバリデーターは、Operator Framework によって提供されるバリデーターのスイート全体に対して Operator バンドルをテストします。	**suite=operatorframework**
OperatorHub	このバリデーターは、OperatorHub との互換性に関して、Operator バンドルをテストします。	**name=operatorhub**
Good Practices	このバリデーターは、Operator バンドルが Operator Framework で定義されるグッドプラクティスに準拠するかどうかをテストします。これは、空の CRD 記述またはサポート対象外の Operator Lifecycle Manager (OLM) リソースなどの問題の有無をチェックします。	**name=good-practices**

関連情報

- Bundle Format

5.10.3. bundle validate コマンドの実行

デフォルトのバリデーターは、**bundle validate** コマンドを実行するたびにテストを実行します。オプションのバリデーターは、**--select-optional** フラグを使用して実行できます。オプションバリデーターは、デフォルトのテストに追加してテストを実行します。

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こと。

手順

1. ローカルバンドルディレクトリーに対してデフォルトのバリデーターを実行する場合は、Operator プロジェクトディレクトリーから以下のコマンドを入力します。

   ```
   $ operator-sdk bundle validate ./bundle
   ```

2. リモート Operator バンドルイメージに対してデフォルトのバリデーターを実行する必要がある場合は、以下のコマンドを入力します。

```
$ operator-sdk bundle validate \
  <bundle_registry>/<bundle_image_name>:<tag>
```

ここでは、以下のようになります。

<bundle_registry>

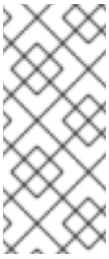
バンドルがホストされるレジストリーを指定します (例: **quay.io/example**)。

<bundle_image_name>

バンドルイメージの名前を指定します (例: **memcached-operator**)。

<tag>

v1.31.0 などのバンドルイメージのタグを指定します。



注記

Operator バンドルイメージを検証する必要がある場合は、イメージをリモートレジストリーでホストする必要があります。Operator SDK はイメージをプルし、テストを実行する前にこれをローカルにビルドします。**bundle validate** コマンドは、ローカルバンドルイメージのテストをサポートしません。

3. Operator バンドルに対して追加のバリデーターを実行する必要がある場合は、以下のコマンドを入力します。

```
$ operator-sdk bundle validate \
  <bundle_dir_or_image> \
  --select-optional <test_label>
```

ここでは、以下のようになります。

<bundle_dir_or_image>

~/projects/memcached または **quay.io/example/memcached-operator:v1.31.0** などのローカルバンドルディレクトリーまたはリモートバンドルイメージを指定します。

<test_label>

実行するバリデーターの名前を指定します (例: **name=good-practices**)。

出力例

```
ERRO[0000] Error: Value apiextensions.k8s.io/v1, Kind=CustomResource: unsupported
media type registry+v1 for bundle object
WARN[0000] Warning: Value k8sevent.v0.0.1: owned CRD
"k8sevents.k8s.k8sevent.com" has an empty description
```

5.11. 高可用性またはシングルノードのクラスターの検出およびサポート

OpenShift Container Platform クラスターの高可用性 (HA) モードと非 HA モードの両方で Operator を適切に実行するには、Operator SDK を使用してクラスターのインフラストラクチャートポロジーを検出し、クラスターのトポロジーに適合するリソース要件を設定します。

OpenShift Container Platform クラスターは、複数のノードを使用する高可用性 (HA) モード、または

シングルノードを使用する非 HA モードで設定できます。シングルノード OpenShift とも呼ばれるシングルノードクラスターには、より慎重なリソース制約がある可能性があります。したがって、シングルノードクラスターにインストールされた Operator がそれに応じて調整でき、正常に実行できることが重要です。

OpenShift Dedicated で提供されるクラスター高可用性モード API にアクセスすることにより、Operator の作成者は、Operator SDK を使用して、Operator がクラスターのインフラストラクチャートポロジー (HA モードまたは非 HA モード) を検出できるようにすることができます。カスタム Operator ロジックは、検出されたクラスタートポロジーを使用して、Operator およびそれが管理するオペランドまたはワークロードの両方のリソース要件を、トポロジーに最も適したプロファイルに自動的に切り替えるように開発することができます。

5.11.1. クラスター高可用性モード API について

OpenShift Dedicated は、Operator がインフラストラクチャートポロジーの検出に使用するクラスター高可用性モード API を備えています。インフラストラクチャー API は、インフラストラクチャーに関するクラスター全体の情報を保持します。Operator Lifecycle Manager(OLM) 管理の Operator は、高可用性モードに基づいてオペランドまたは管理ワークロードを異なる方法で設定する必要がある場合にインフラストラクチャー API を使用できます。

インフラストラクチャー API では、**infrastructureTopology** ステータスは、コントロールプレーンノードで実行されないインフラストラクチャーサービスの期待値を表します。通常、これは値が **master** 以外の ロール のノードセレクターでわかります。**controlPlaneTopology** ステータスは、通常コントロールプレーンノードで実行されるオペランドの期待値を表します。

ステータスがいずれの場合もデフォルト設定は **HighlyAvailable** で、複数ノードクラスターで Operator が行う動作を表します。**SingleReplica** 設定は、シングルノード OpenShift とも呼ばれるシングルノードクラスターで使用されます。この設定は、オペランドを高可用性運用向けに設定しないよう Operator に指示するものです。

OpenShift Dedicated インストーラーは、次のルールに従って、クラスターの作成時のレプリカ数に基づいて **controlPlaneTopology** および **infrastructureTopolog** ステータスフィールドを設定します。

- コントロールプレーンのレプリカ数が 3 未満の場合には、**controlPlaneTopology** のステータスは **SingleReplica** に設定されます。それ以外の場合は、**HighlyAvailable** に設定されます。
- ワーカーレプリカ数が 0 の場合に、コントロールプレーンノードもワーカーとして設定されます。したがって、**infrastructureTopology** のステータスは **controlPlaneTopology** ステータスと同じです。
- ワーカーレプリカ数が 1 の場合、**infrastructureTopology** は **SingleReplica** に設定されます。それ以外の場合は、**HighlyAvailable** に設定されます。

5.11.2. Operator プロジェクトでの API 使用状況の例

Operator の作成者は、以下の例のように、通常の Kubernetes コンストラクトおよび **controller-runtime** ライブラリーを使用してインフラストラクチャー API にアクセスできるように Operator プロジェクトを更新できます。

controller-runtime ライブラリーの例

```
// Simple query
 nn := types.NamespacedName{
 Name: "cluster",
 }
 infraConfig := &configv1.Infrastructure{}
```

```
 err = crClient.Get(context.Background(), nn, infraConfig)
 if err != nil {
 return err
 }
 fmt.Printf("using crclient: %v\n", infraConfig.Status.ControlPlaneTopology)
 fmt.Printf("using crclient: %v\n", infraConfig.Status.InfrastructureTopology)
```

Kubernetes のコンストラクトの例

```
operatorConfigInformer := configinformer.NewSharedInformerFactoryWithOptions(configClient,
2*time.Second)
 infrastructureLister = operatorConfigInformer.Config().V1().Infrastructures().Lister()
 infraConfig, err := configClient.ConfigV1().Infrastructures().Get(context.Background(), "cluster",
metav1.GetOptions{})
 if err != nil {
 return err
 }
// fmt.Printf("%v\n", infraConfig)
 fmt.Printf("%v\n", infraConfig.Status.ControlPlaneTopology)
 fmt.Printf("%v\n", infraConfig.Status.InfrastructureTopology)
```

5.12. PROMETHEUS による組み込みモニタリングの設定

Operator SDK には、Prometheus Operator を使用したモニタリングサポートが組み込まれています。これを使用すると、Operator のカスタムメトリクスを公開できます。

警告

デフォルトでは、OpenShift Dedicated は **openshift-user-workload-monitoring** プロジェクトに Prometheus Operator を提供します。この Prometheus インスタンスを使用して、OpenShift Dedicated のユーザーワークロードを監視してください。

openshift-monitoring プロジェクトの Prometheus Operator は使用しないでください。この Prometheus インスタンスは、Red Hat Site Reliability Engineer (SRE) がコアクラスターコンポーネントの監視に使用します。

関連情報

- Go ベースの Operator のカスタムメトリクスの公開 (OpenShift Container Platform ドキュメント)
- Ansible ベースの Operator のカスタムメトリクスの公開 (OpenShift Container Platform ドキュメント)
- OpenShift Dedicated の モニタリングスタックについて

5.13. 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、リーダーがそのロール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、削除される場合にガベージコレクションを使用してリーダーシップを放棄します。この実装は (スプリットブレインとしても知られる) 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ことを防ぎます。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上にある場合、リーダー Pod に **node.kubernetes.io/unreachable** および **node.kubernetes.io/not-ready** 許容を指定し、**tolerationSeconds** 値を使用して、リーダー Pod がノードから削除され、ステップダウンする時間を指定できます。これらの許容値は、デフォルトで、受付時に 5 分の **tolerationSeconds** 値で Pod に追加されます。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。

5.13.1. Operator リーダー選出の例

次の例では、Operator のリーダー選出オプション (Leader-for-life と Leader-with-lease) 2 つの使用方法を説明します。

5.13.1.1. Leader-for-life 選択

Leader-for-life 選択の実装の場合、**leader.Become()** の呼び出しは、**memcached-operator-lock** という名前の設定マップを作成して、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 の名前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。

5.13.1.2. Leader-with-lease 選択

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設定マップ を作成するために 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。

5.14. GO ベースの OPERATOR 用のオブジェクトプルーニングユーティリティー

operator-libプルーニングユーティリティーを使用すると、Go ベースの Operator は、オブジェクトが不要になったときにオブジェクトをクリーンアップまたはプルーニングできます。Operator の作成者は、ユーティリティーを使用してカスタムフックと戦略を作成することもできます。

5.14.1. operator-lib プルーニングユーティリティーについて

ジョブや Pod などのオブジェクトは、Operator ライフサイクルの通常の部分として作成されます。**dedicated-admin** ロールを持つ管理者またはOperator がこれらのオブジェクトを削除しないと、オブジェクトはクラスター内に留まり、リソースを消費する可能性があります。

以前は、不要なオブジェクトの整理に次のオプションを使用できました。

- Operator の作成者は、Operator 向けに独自のプルーニングソリューションを作成する必要がありました。
- クラスター管理者は、自分でオブジェクトをクリーンアップする必要がありました。

operator-libプルーニングユーティリティーでは、特定の namespace の Kubernetes クラスターからオブジェクトを削除します。このライブラリーは、Operator Framework の一部として**operator-lib**ライブラリーのバージョン**0.9.0**で追加されました。

5.14.2. プルーニングユーティリティーの設定

operator-libプルーニングユーティリティーは Go で記述されており、Go ベースの Operator の一般的なプルーニング戦略が含まれています。

設定例

-

```
cfg = Config{
        log:           logf.Log.WithName("prune"),
        DryRun:        false,
        Clientset:     client,
        LabelSelector: "app=<operator_name>",
        Resources: []schema.GroupVersionKind{
                {Group: "", Version: "", Kind: PodKind},
        },
        Namespaces: []string{"<operator_namespace>"},
        Strategy: StrategyConfig{
                Mode:            MaxCountStrategy,
                MaxCountSetting: 1,
        },
        PreDeleteHook: myhook,
}
```

プルーニングユーティリティー設定ファイルは、次のフィールドを使用してプルーニングアクションを定義します。

設定フィールド	説明
log	ライブラリーログメッセージの処理に使用されるロガー。
DryRun	リソースを削除するかどうかを決定するブール値。**true**に設定すると、ユーティリティーは実行されますが、リソースは削除されません。
Clientset	Client-Kubernetes API 呼び出しに使用される Client-go Kubernetes ClientSet。
LabelSelector	プルーニングするリソースの検索時に使用される Kubernetes ラベルセレクター式。
Resources	Kubernetes リソースの種類。**PodKind**と**JobKind**は現在サポートされています。
Namespaces	リソースを検索する Kubernetes namespace のリスト。
ストラテジー	実行するプルーニングストラテジー。
Strategy.Mode	**Max Count Strategy**、**Max Age Strategy**、または**Custom Strategy**が現在サポートされています。
Strategy.MaxCountSetting	プルーニングユーティリティーの実行後に残るリソース数を指定する **MaxCountStrategy** の整数値。
Strategy.MaxAgeSetting	リソースのプルーニングの有効期限を指定する Go **time.Duration** の文字列。例: **48h**
Strategy.CustomSettings	カスタムストラテジー関数に指定可能な Go マップの値

設定フィールド	説明
PreDeleteHook	オプション: リソースのプルーニング前に呼び出す Go 関数
CustomStrategy	オプション: カスタムプルーニングストラテジーを実装する Go 関数

プルーニングの実行

プルーニング設定で execute 関数を実行して、プルーニングアクションを呼び出すことができます。

```
err := cfg.Execute(ctx)
```

cron パッケージを使用するか、トリガーイベントを指定してプルーニングユーティリティーを呼び出して、プルーニングアクションを呼び出すこともできます。

5.15. パッケージマニフェストプロジェクトのバンドル形式への移行

Operator のレガシー パッケージマニフェスト形式 のサポートは、OpenShift Dedicated 4.8 以降では削除されています。パッケージマニフェスト形式で最初に作成された Operator プロジェクトがある場合、Operator SDK を使用してプロジェクトをバンドル形式に移行できます。バンドル形式は、OpenShift Dedicated 4.6 以降の Operator Lifecycle Manager (OLM) の推奨パッケージ形式です。

5.15.1. パッケージ形式の移行について

Operator SDK の **pkgman-to-bundle** コマンドは、Operator Lifecycle Manager (OLM) パッケージマニフェストをバンドルに移行する際に役立ちます。このコマンドは、入力パッケージマニフェストディレクトリーを取得し、入力ディレクトリーにあるマニフェストの各バージョンのバンドルを生成します。その後、生成されるバンドルごとにバンドルイメージをビルドすることもできます。

たとえば、パッケージマニフェスト形式のプロジェクトの以下の **packagemanifests/** ディレクトリーについて見てみましょう。

Package Manifest Format のレイアウトの例

```
packagemanifests/
└── etcd
    ├── 0.0.1
    │   ├── etcdcluster.crd.yaml
    │   └── etcdoperator.clusterserviceversion.yaml
    ├── 0.0.2
    │   ├── etcdbackup.crd.yaml
    │   ├── etcdcluster.crd.yaml
    │   ├── etcdoperator.v0.0.2.clusterserviceversion.yaml
    │   └── etcdrestore.crd.yaml
    └── etcd.package.yaml
```

移行の実行後に、以下のバンドルが **bundle/** ディレクトリーに生成されます。

Bundle Format のレイアウトの例

```
bundle/
├── bundle-0.0.1
```

```
│   ├── bundle.Dockerfile
│   ├── manifests
│   │   ├── etcdcluster.crd.yaml
│   │   └── etcdoperator.clusterserviceversion.yaml
│   ├── metadata
│   │   └── annotations.yaml
│   └── tests
│       └── scorecard
│           └── config.yaml
└── bundle-0.0.2
    ├── bundle.Dockerfile
    ├── manifests
    │   ├── etcdbackup.crd.yaml
    │   ├── etcdcluster.crd.yaml
    │   ├── etcdoperator.v0.0.2.clusterserviceversion.yaml
    │   └── etcdrestore.crd.yaml
    ├── metadata
    │   └── annotations.yaml
    └── tests
        └── scorecard
            └── config.yaml
```

この生成されたレイアウトに基づいて、両方のバンドルのバンドルイメージも以下の名前でビルドされます。

- **quay.io/example/etcd:0.0.1**
- **quay.io/example/etcd:0.0.2**

関連情報

- Operator Framework パッケージ形式

5.15.2. パッケージマニフェストプロジェクトのバンドル形式への移行

Operator の作成者は Operator SDK を使用して、パッケージマニフェスト形式 Operator プロジェクトをバンドル形式のプロジェクトに移行できます。

前提条件

- Operator SDK CLI がインストールされている。
- Operator プロジェクトが初回にパッケージマニフェスト形式の Operator SDK を使用して生成されている

手順

- Operator SDK を使用してパッケージマニフェストプロジェクトをバンドル形式に移行し、バンドルイメージを生成します。

```
$ operator-sdk pkgman-to-bundle <package_manifests_dir> \ 1
  [--output-dir <directory>] \ 2
  --image-tag-base <image_name_base> 3
```

1. **packagemanifests/** または **manifests/** などのプロジェクトのパッケージマニフェストディレクトリーの場所を指定します。
2. オプション: デフォルトで、生成されたバンドルはローカルで **bundle/** ディレクトリーに書き込まれます。**--output-dir** フラグを使用して、別の場所を指定することができます。
3. **--image-tag-base** フラグを設定して、バンドルに使用される **quay.io/example/etcd** などのイメージ名のベースを提供します。イメージのタグはバンドルのバージョンに応じて設定されるため、タグを指定せずに名前を指定します。たとえば、完全なバンドルイメージ名は **<image_name_base>:<bundle_version>** の形式で生成されます。

検証

- 生成されたバンドルイメージが正常に実行されることを確認します。

```
$ operator-sdk run bundle <bundle_image_name>:<tag>
```

出力例

```
INFO[0025] Successfully created registry pod: quay-io-my-etcd-0-9-4
INFO[0025] Created CatalogSource: etcd-catalog
INFO[0026] OperatorGroup "operator-sdk-og" created
INFO[0026] Created Subscription: etcdoperator-v0-9-4-sub
INFO[0031] Approved InstallPlan install-5t58z for the Subscription: etcdoperator-v0-9-4-sub
INFO[0031] Waiting for ClusterServiceVersion "default/etcdoperator.v0.9.4" to reach
'Succeeded' phase
INFO[0032]   Waiting for ClusterServiceVersion "default/etcdoperator.v0.9.4" to appear
INFO[0048]   Found ClusterServiceVersion "default/etcdoperator.v0.9.4" phase: Pending
INFO[0049]   Found ClusterServiceVersion "default/etcdoperator.v0.9.4" phase: Installing
INFO[0064]   Found ClusterServiceVersion "default/etcdoperator.v0.9.4" phase: Succeeded
INFO[0065] OLM has successfully installed "etcdoperator.v0.9.4"
```

5.16. OPERATOR SDK CLI リファレンス

Operator SDK コマンドラインインターフェイス (CLI) は、Operator の作成を容易にするために設計された開発キットです。

Operator SDK CLI 構文

```
$ operator-sdk <command> [<subcommand>] [<argument>] [<flags>]
```

Kubernetes ベースのクラスター (OpenShift Dedicated など) へのクラスター管理者アクセス権を持つ Operator 作成者は、Operator SDK CLI を使用して、Go、Ansible、または Helm ベースの独自の Operator を開発できます。Kubebuilder は Go ベースの Operator のスキャフォールディングソリューションとして Operator SDK に組み込まれます。つまり、既存の Kubebuilder プロジェクトは Operator SDK でそのまま使用でき、引き続き機能します。

5.16.1. bundle

operator-sdk bundle コマンドは Operator バンドルメタデータを管理します。

5.16.1.1. validate

bundle validate サブコマンドは Operator バンドルを検証します。

表5.23 **bundle validate** フラグ

フラグ	説明
-h、**--help**	**bundle validate** サブコマンドのヘルプ出力。
--index-builder (文字列)	バンドルイメージをプルおよびデプロイメントするためのツール。バンドルイメージを検証する場合にのみ使用されます。使用できるオプションは、**docker** (デフォルト)、**podman**、または **none** です。
--list-optional	利用可能なすべてのオプションのバリデーターをリスト表示します。これが設定されている場合、バリデーターは実行されません。
--select-optional (文字列)	実行するオプションのバリデーターを選択するラベルセレクター。**--list-optional** フラグを指定して実行する場合は、利用可能なオプションのバリデーターをリスト表示します。

5.16.2. cleanup

operator-sdk cleanup コマンドは、**run** コマンドでデプロイされた Operator 用に作成されたリソースを破棄し、削除します。

表5.24 **cleanup** フラグ

フラグ	説明
-h、**--help**	**run bundle** サブコマンドのヘルプ出力。
--kubeconfig (文字列)	CLI 要求に使用する **kubeconfig** ファイルへのパス。
-n, **--namespace** (文字列)	CLI 要求がある場合の CLI 要求を実行する namespace。
--timeout <duration>	コマンドが失敗せずに完了するまでの待機時間。デフォルト値は **2m0s** です。

5.16.3. 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5.25 **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5.26 **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。

以下に例を示します。

```
$ operator-sdk completion bash
```

出力例

```
# bash completion for operator-sdk                  -*- shell-script -*-
...
# ex: ts=4 sw=4 et filetype=sh
```

5.16.4. create

operator-sdk create コマンドは、Kubernetes API の作成または スキャフォールディング に使用されます。

5.16.4.1. api

create api サブコマンドは Kubernetes API をスキャフォールディングします。サブコマンドは、 **init** コマンドで初期化されたプロジェクトで実行する必要があります。

表5.27 **create api** フラグ

フラグ	説明
-h、--help	**run bundle** サブコマンドのヘルプ出力。

5.16.5. 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。

5.16.5.1. bundle

generate bundle サブコマンドは、Operator プロジェクトのバンドルマニフェスト、メタデータ、および **bundle.Dockerfile** ファイルのセットを生成します。

注記

通常は、最初に **generate kustomize manifests** サブコマンドを実行して、**generate bundle** サブコマンドで使用される入力された Kustomize ベースを生成します。ただし、初期化されたプロジェクトで **make bundle** コマンドを使用して、これらのコマンドの順次の実行を自動化できます。

表5.28 **generate bundle** フラグ

フラグ	説明
--channels (文字列)	バンドルが属するチャネルのコンマ区切りリスト。デフォルト値は **alpha** です。
--crds-dir (文字列)	**CustomResoureDefinition** マニフェストのルートディレクトリー。
--default-channel (文字列)	バンドルのデフォルトチャネル。
--deploy-dir (文字列)	デプロイメントや RBAC などの Operator マニフェストのルートディレクトリー。このディレクトリーは、**--input-dir** フラグに渡されるディレクトリーとは異なります。
-h、--help	**generate bundle** のヘルプ
--input-dir (文字列)	既存のバンドルを読み取るディレクトリー。このディレクトリーは、バンドル **manifests** ディレクトリーの親であり、**--deploy-dir** ディレクトリーとは異なります。
--kustomize-dir (文字列)	バンドルマニフェストの Kustomize ベースおよび **kustomization.yaml** ファイルを含むディレクトリー。デフォルトのパスは **config/manifests** です。
--manifests	バンドルマニフェストを生成します。
--metadata	バンドルメタデータと Dockerfile を生成します。
--output-dir (文字列)	バンドルを書き込むディレクトリー。
--overwrite	バンドルメタデータおよび Dockerfile を上書きします (ある場合)。デフォルト値は **true** です。
--package (文字列)	バンドルのパッケージ名。
-q、--quiet	quiet モードで実行します。
--stdout	バンドルマニフェストを標準出力に書き込みます。
--version (文字列)	生成されたバンドルの Operator のセマンティックバージョン。新規バンドルを作成するか、Operator をアップグレードする場合にのみ設定します。

関連情報

- **generate bundle** サブコマンドを呼び出すための **make bundle** コマンドの使用を含む詳細な手順については、Operator のバンドル を参照してください。

5.16.5.2. kustomize

generate kustomize サブコマンドには、Operator の Kustomize データを生成するサブコマンドが含まれます。

5.16.5.2.1. manifests

generate kustomize manifests は Kustomize ベースを生成または再生成し、**kustomization.yaml** ファイルを **config/manifests** ディレクトリーに生成または再生成します。これは、他の Operator SDK コマンドでバンドルマニフェストをビルドするために使用されます。このコマンドは、ベースがすでに存在しない場合や **--interactive=false** フラグが設定されていない場合に、デフォルトでマニフェストベースの重要なコンポーネントである UI メタデータを対話的に要求します。

表5.29 **generate kustomize manifests** フラグ

フラグ	説明
--apis-dir (文字列)	API タイプ定義のルートディレクトリー。
-h、--help	**generate kustomize manifests** のヘルプ。
--input-dir (文字列)	既存の Kustomize ファイルを含むディレクトリー。
--interactive	**false** に設定すると、Kustomize ベースが存在しない場合は、対話式コマンドプロンプトがカスタムメタデータを受け入れるように表示されます。
--output-dir (文字列)	Kustomize ファイルを書き込むディレクトリー。
--package (文字列)	パッケージ名。
-q、--quiet	quiet モードで実行します。

5.16.6. init

operator-sdk init コマンドは Operator プロジェクトを初期化し、指定されたプラグインのデフォルトのプロジェクトディレクトリーレイアウトを生成または スキャフォールド します。

このコマンドは、以下のファイルを作成します。

- ボイラープレートライセンスファイル
- ドメインおよびリポジトリーを含む **PROJECT** ファイル
- プロジェクトをビルドする **Makefile**
- プロジェクト依存関係のある **go.mod** ファイル
- マニフェストをカスタマイズするための **kustomization.yaml** ファイル
- マネージャーマニフェストのイメージをカスタマイズするためのパッチファイル
- Prometheus メトリクスを有効にするためのパッチファイル
- 実行する **main.go** ファイル

表5.30 **init** フラグ

フラグ	説明
--help, -h	**init** コマンドのヘルプ出力。
--plugins (文字列)	プロジェクトを初期化するプラグインの名前およびオプションのバージョン。利用可能なプラグインは **ansible.sdk.operatorframework.io/v1**、**go.kubebuilder.io/v2**、**go.kubebuilder.io/v3**、および **helm.sdk.operatorframework.io/v1** です。
--project-version	プロジェクトのバージョン。使用できる値は **2** および **3-alpha** (デフォルト) です。

5.16.7. run

operator-sdk run コマンドは、さまざまな環境で Operator を起動できるオプションを提供します。

5.16.7.1. bundle

run bundle サブコマンドは、Operator Lifecycle Manager (OLM) を使用してバンドル形式で Operator をデプロイします。

表5.31 **run bundle** フラグ

フラグ	説明
--index-image (文字列)	バンドルを挿入するインデックスイメージ。デフォルトのイメージは **quay.io/operator-framework/upstream-opm-builder:latest** です。
--install-mode <install_mode_value>	Operator のクラスターサービスバージョン (CSV) によってサポートされるインストールモード (例: **AllNamespaces** または **SingleNamespace**)。
--timeout <duration>	インストールのタイムアウト。デフォルト値は **2m0s** です。
--kubeconfig (文字列)	CLI 要求に使用する **kubeconfig** ファイルへのパス。
-n, **--namespace** (文字列)	CLI 要求がある場合の CLI 要求を実行する namespace。
--security-context-config <security_context>	カタログ Pod に使用するセキュリティーコンテキストを指定します。許可される値には、**restricted** および **legacy** が含まれます。デフォルト値は **legacy** です。[1]
-h、**--help**	**run bundle** サブコマンドのヘルプ出力。

1. **restricted** セキュリティーコンテキストは、**default** namespace と互換性がありません。実稼働環境で Operator の Pod セキュリティーアドミッションを設定する場合は、「Pod セキュリティーアドミッションに準拠」を参照してください。Pod セキュリティーアドミッションの詳細は、「Pod セキュリティーアドミッションの理解と管理」を参照してください。

関連情報

- 使用可能なインストールモードに関する詳細は、Operator グループメンバーシップ を参照してください。
- Pod セキュリティーアドミッションに準拠
- Pod セキュリティーアドミッションの理解と管理

5.16.7.2. bundle-upgrade

run bundle-upgrade サブコマンドは、以前に Operator Lifecycle Manager (OLM) を使用してバンドル形式でインストールされた Operator をアップグレードします。

表5.32 **run bundle-upgrade** フラグ

フラグ	説明
--timeout <duration>	アップグレードのタイムアウト。デフォルト値は **2m0s** です。
--kubeconfig (文字列)	CLI 要求に使用する **kubeconfig** ファイルへのパス。
-n, --namespace (文字列)	CLI 要求がある場合の CLI 要求を実行する namespace。
--security-context-config <security_context>	カタログ Pod に使用するセキュリティーコンテキストを指定します。許可される値には、**restricted** および **legacy** が含まれます。デフォルト値は **legacy** です。[1]
-h、--help	**run bundle** サブコマンドのヘルプ出力。

1. **restricted** セキュリティーコンテキストは、**default** namespace と互換性がありません。実稼働環境で Operator の Pod セキュリティーアドミッションを設定する場合は、「Pod セキュリティーアドミッションに準拠」を参照してください。Pod セキュリティーアドミッションの詳細は、「Pod セキュリティーアドミッションの理解と管理」を参照してください。

関連情報

- Pod セキュリティーアドミッションに準拠
- Pod セキュリティーアドミッションの理解と管理

5.16.8. scorecard

operator-sdk scorecard コマンドは、スコアカードツールを実行して Operator バンドルを検証し、改善に向けた提案を提供します。このコマンドは、バンドルイメージまたはマニフェストおよびメタデータを含むディレクトリーのいずれかの引数を取ります。引数がイメージタグを保持する場合は、イメージはリモートに存在する必要があります。

表5.33 **scorecard** フラグ

フラグ	説明
-c, **--config** (文字列)	スコアカード設定ファイルへのパス。デフォルトのパスは **bundle/tests/scorecard/config.yaml** です。
-h、**--help**	**scorecard** コマンドのヘルプ出力。
--kubeconfig (文字列)	**kubeconfig** ファイルへのパス。
-L, **--list**	実行可能なテストをリスト表示します。
-n, --namespace (文字列)	テストイメージを実行する namespace。
-o, **--output** (文字列)	結果の出力形式。使用できる値はデフォルトの **text**、および **json** です。
--pod-security <security_context>	指定されたセキュリティーコンテキストでスコアカードを実行するオプション。許可される値には、**restricted** および **legacy** が含まれます。デフォルト値は **legacy** です。[1]
-l, **--selector** (文字列)	実行されるテストを決定するラベルセレクター。
-s, **--service-account** (文字列)	テストに使用するサービスアカウント。デフォルト値は **default** です。
-x, **--skip-cleanup**	テストの実行後にリソースクリーンアップを無効にします。
-w, **--wait-time <duration>**	テストが完了するのを待つ秒数 (例: **35s**)。デフォルト値は **30s** です。

1. **restricted** セキュリティーコンテキストは、**default** namespace と互換性がありません。実稼働環境で Operator の Pod セキュリティーアドミッションを設定する場合は、「Pod セキュリティーアドミッションに準拠」を参照してください。Pod セキュリティーアドミッションの詳細は、「Pod セキュリティーアドミッションの理解と管理」を参照してください。

関連情報

- スコアカードツールの実行に関する詳細は、スコアカードを使用した Operator の検証 を参照してください。
- Pod セキュリティーアドミッションに準拠
- Pod セキュリティーアドミッションの理解と管理

5.17. OPERATOR SDK V0.1.0 への移行

以下では、Operator SDK v0.0.x を使用してビルドされた Operator プロジェクトを Operator SDK v0.1.0 で必要なプロジェクト構造に移行する方法について説明します。

プロジェクトの移行で推奨される方法は、以下の通りです。

1. 新規 v0.1.0 プロジェクトを初期化します。
2. コードを新規プロジェクトにコピーします。
3. v0.1.0 について説明されているように新規プロジェクトを変更します。

ここでは、Operator SDK のサンプルプロジェクトである **memcached-operator** を使用して、移行手順を説明します。移行前および移行後のそれぞれの例については、v0.0.7 memcached-operator および v0.1.0 memcached-operator プロジェクト構造を参照してください。

5.17.1. 新規 Operator SDK v0.1.0 プロジェクトの作成

Operator SDK v0.0.x プロジェクトの名前を変更し、代わりに新規の v0.1.0 プロジェクトを作成します。

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 v0.1.0 CLI
- 以前のバージョンの Operator SDK を使用して以前にデプロイされた **memcached-operator** プロジェクト

手順

1. SDK バージョンが v0.1.0 であることを確認します。

```
$ operator-sdk --version
operator-sdk version 0.1.0
```

2. 新しい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/
$ mv memcached-operator old-memcached-operator
$ operator-sdk new memcached-operator --skip-git-init
$ ls
memcached-operator old-memcached-operator
```

3. 古いプロジェクトから **.git** をコピーします。

```
$ cp -rf old-memcached-operator/.git memcached-operator/.git
```

5.17.2. カスタムタイプの pkg/apis からの移行

プロジェクトのカスタムタイプを更新された Operator SDK v0.1.0 の使用に移行します。

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 v0.1.0 CLI
- 以前のバージョンの Operator SDK を使用して以前にデプロイされた **memcached-operator** プロジェクト
- Operator SDK v0.1.0 を使用して作成される新規プロジェクト

手順

1. カスタムタイプの Scaffolding (スキャフォールディング) API を作成します。

   a. **operator-sdk add api --api-version=<apiversion> --kind=<kind>** を使用して新規プロジェクトにカスタムリソース (CR) の API を作成します。

```
$ cd memcached-operator
$ operator-sdk add api --api-version=cache.example.com/v1alpha1 --kind=Memcached

$ tree pkg/apis
pkg/apis/
├── addtoscheme_cache_v1alpha1.go
├── apis.go
└── cache
    └── v1alpha1
        ├── doc.go
        ├── memcached_types.go
        ├── register.go
        └── zz_generated.deepcopy.go
```

   b. 古いプロジェクトで定義したカスタムタイプについて、直前のコマンドを繰り返します。それぞれのタイプはファイル **pkg/apis/<group>/<version>/<kind>_types.go** に定義されます。

2. タイプの内容をコピーします。

   a. **pkg/apis/<group>/<version>/types.go** ファイルの **Spec** および **Status** の内容を、古いプロジェクトから新規プロジェクトの **pkg/apis/<group>/<version>/<kind>_types.go** ファイルにコピーします。

   b. それぞれの **<kind>_types.go** ファイルには **init()** 関数があります。これはこのタイプを Manager のスキームに登録するために使用されるため、削除しないでください。

```
func init() {
 SchemeBuilder.Register(&Memcached{}, &MemcachedList{})
```

5.17.3. 調整 (reconcile) コードの移行

プロジェクトの調整コードを更新 Operator SDK v0.1.0 の使用に移行します。

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 v0.1.0 CLI
- 以前のバージョンの Operator SDK を使用して以前にデプロイされた **memcached-operator** プロジェクト
- カスタムタイプの **pkg/apis/** からの移行

手順

1. **CR** を監視するコントローラーを追加します。
   V0.0.x プロジェクトでは、監視されるリソースは以前は **cmd/<operator-name>/main.go** に定義されました。

```
sdk.Watch("cache.example.com/v1alpha1", "Memcached", "default",
time.Duration(5)*time.Second)
```

V0.1.0 プロジェクトの場合、コントローラー を定義してリソースを監視する必要があります。

a. **operator-sdk add controller --api-version=<apiversion> --kind=<kind>** を使用して CR タイプを監視するコントローラーを追加します。

```
$ operator-sdk add controller --api-version=cache.example.com/v1alpha1 --
kind=Memcached

$ tree pkg/controller
pkg/controller/
├── add_memcached.go
├── controller.go
└── memcached
    └── memcached_controller.go
```

b. **pkg/controller/<kind>/<kind>_controller.go** ファイルで **add()** 関数を検査します。

```
import (
    cachev1alpha1 "github.com/example-inc/memcached-
operator/pkg/apis/cache/v1alpha1"
    ...
)

func add(mgr manager.Manager, r reconcile.Reconciler) error {
    c, err := controller.New("memcached-controller", mgr, controller.Options{Reconciler: r})

    // Watch for changes to the primary resource Memcached
    err = c.Watch(&source.Kind{Type: &cachev1alpha1.Memcached{}},
&handler.EnqueueRequestForObject{})

    // Watch for changes to the secondary resource pods and enqueue reconcile requests
for the owner Memcached
    err = c.Watch(&source.Kind{Type: &corev1.Pod{}},
&handler.EnqueueRequestForOwner{
  IsController: true,
  OwnerType:    &cachev1alpha1.Memcached{},
 })
}
```

2 つ目の **Watch()** を削除するか、またはこれを CR が所有する 2 つ目のリソースタイプを監視するように変更します。

複数のリソースを監視すると、アプリケーションに関連する複数リソースの調整ループ (reconcile loop) をトリガーできます。詳細は、監視および eventhandling についてのドキュメントおよび Kubernetes の コントローラーの規則 についてのドキュメントを参照してください。

Operator が複数の CR タイプを監視している場合、アプリケーションに応じて以下のいずれかを実行できます。

- CR がプライマリー CR によって所有されている場合、同じコントローラーでこれをセカンダリーリソースとして監視し、プライマリーリソースの調整ループをトリガーします。

```
// Watch for changes to the primary resource Memcached
    err = c.Watch(&source.Kind{Type: &cachev1alpha1.Memcached{}},
&handler.EnqueueRequestForObject{})

    // Watch for changes to the secondary resource AppService and enqueue
reconcile requests for the owner Memcached
    err = c.Watch(&source.Kind{Type: &appv1alpha1.AppService{}},
&handler.EnqueueRequestForOwner{
  IsController: true,
  OwnerType:    &cachev1alpha1.Memcached{},
 })
```

- 新規のコントローラーを追加して、他の CR とは別に CR を監視し、調整します。

```
$ operator-sdk add controller --api-version=app.example.com/v1alpha1 --
kind=AppService
```

```
  // Watch for changes to the primary resource AppService
    err = c.Watch(&source.Kind{Type: &appv1alpha1.AppService{}},
&handler.EnqueueRequestForObject{})
```

2. **pkg/stub/handler.go** から調整コード **(reconcile code)** をコピーし、変更します。v0.1.0 プロジェクトでは、調整コードはコントローラーの Reconciler の **Reconcile()** メソッドで定義されます。これは、古いプロジェクトの **Handle()** 関数に似ています。引数と戻り値の違いに注意してください。

- Reconcile (調整):

```
    func (r *ReconcileMemcached) Reconcile(request reconcile.Request)
(reconcile.Result, error)
```

- Handle (処理):

```
    func (h *Handler) Handle(ctx context.Context, event sdk.Event) error
```

sdk.Event (オブジェクトを含む) を受信する代わりに、**Reconcile()** 関数は Request (**Name**/**Namespace** キー) を受信してオブジェクトを検索します。

Reconcile() 関数がエラーを返すと、コントローラーは **Request** を再度キューに入れ、再試行します。エラーが返されない場合は、Result に応じて、コントローラーは **Request** を再試行しないか、即時に再試行するか、または指定された期間後に再試行します。

a. 古いプロジェクトの **Handle()** 関数のコードを、コントローラーの **Reconcile()** 関数の既存のコードにコピーします。**Request** のオブジェクトを検索し、これが削除されているかどうかをチェックする **Reconcile()** コードの最初のセクションを保持するようにします。

```
import (
    apierrors "k8s.io/apimachinery/pkg/api/errors"
    cachev1alpha1 "github.com/example-inc/memcached-
operator/pkg/apis/cache/v1alpha1"
```

```
    ...
)
func (r *ReconcileMemcached) Reconcile(request reconcile.Request) (reconcile.Result,
error) {
    // Fetch the Memcached instance
 instance := &cachev1alpha1.Memcached{}
    err := r.client.Get(context.TODO()
    request.NamespacedName, instance)
    if err != nil {
        if apierrors.IsNotFound(err) {
            // Request object not found, could have been deleted after reconcile request.
            // Owned objects are automatically garbage collected.
            // Return and don't requeue
            return reconcile.Result{}, nil
        }
        // Error reading the object - requeue the request.
        return reconcile.Result{}, err
    }

    // Rest of your reconcile code goes here.
    ...
}
```

b. 調整コードの戻り値を変更します。

   i. **return err** を **return reconcile.Result{}, err** に置き換えます。

   ii. **return nil** を **return reconcile.Result{}, nil** に置き換えます。

c. コントローラーで CR を定期的に調整するには、**reconcile.Result** の RequeueAfter フィールドを設定します。これにより、コントローラーは **Request** を再度キューに入れ、必要な期間の後に調整をトリガーします。デフォルト値の **0** は、再キューが実行されないことを意味することに注意してください。

```
reconcilePeriod := 30 * time.Second
reconcileResult := reconcile.Result{RequeueAfter: reconcilePeriod}
...

// Update the status
err := r.client.Update(context.TODO(), memcached)
if err != nil {
    log.Printf("failed to update memcached status: %v", err)
    return reconcileResult, err
}
return reconcileResult, nil
```

d. SDK クライアントへの呼び出し (Create、 Update、Delete、Get、List) を reconciler のクライアントに置き換えます。
詳細は、以下の例および **operator-sdk** プロジェクトの **controller-runtime** クライアント API ドキュメント を参照してください。

```
// Create
dep := &appsv1.Deployment{...}
err := sdk.Create(dep)
// v0.0.1
```

```
err := r.client.Create(context.TODO(), dep)

// Update
err := sdk.Update(dep)
// v0.0.1
err := r.client.Update(context.TODO(), dep)

// Delete
err := sdk.Delete(dep)
// v0.0.1
err := r.client.Delete(context.TODO(), dep)

// List
podList := &corev1.PodList{}
labelSelector := labels.SelectorFromSet(labelsForMemcached(memcached.Name))
listOps := &metav1.ListOptions{LabelSelector: labelSelector}
err := sdk.List(memcached.Namespace, podList, sdk.WithListOptions(listOps))
// v0.1.0
listOps := &client.ListOptions{Namespace: memcached.Namespace, LabelSelector:
labelSelector}
err := r.client.List(context.TODO(), listOps, podList)

// Get
dep := &appsv1.Deployment{APIVersion: "apps/v1", Kind: "Deployment", Name: name,
Namespace: namespace}
err := sdk.Get(dep)
// v0.1.0
dep := &appsv1.Deployment{}
err = r.client.Get(context.TODO(), types.NamespacedName{Name: name, Namespace:
namespace}, dep)
```

e. 他のフィールドを **Handler** 構造体から **Reconcile<Kind>** 構造体にコピーして初期化します。

```
// newReconciler returns a new reconcile.Reconciler
func newReconciler(mgr manager.Manager) reconcile.Reconciler {
 return &ReconcileMemcached{client: mgr.GetClient(), scheme: mgr.GetScheme(), foo:
"bar"}
}

// ReconcileMemcached reconciles a Memcached object
type ReconcileMemcached struct {
    client client.Client
    scheme *runtime.Scheme
    // Other fields
    foo string
}
```

3. **main.go** からの変更をコピーします。
cmd/manager/main.go の v0.1.0 Operator の main 関数が、カスタムリソースを登録し、すべてのコントローラーを起動する Manager をセットアップします。

ロジックがコントローラーに定義されているため、SDK 関数 **sdk.Watch()**、**sdk.Handle()**、および **sdk.Run()** を古い **main.go** から移行する必要はありません。

ただし、古い **main.go** ファイルに Operator 固有のフラグまたは設定が定義されている場合は、それらをコピーします。

SDK のスキームに登録されているサードパーティーのリソースタイプがある場合、**operator-sdk** プロジェクトの Advanced Topics を参照し、それを新しいプロジェクトの Manager のスキームに登録する方法を確認してください。

4. ユーザー定義ファイルをコピーします。
古いプロジェクトにユーザー定義の **pkg**、スクリプト、またはドキュメントがある場合は、それらのファイルを新しいプロジェクトにコピーします。

5. デプロイメントマニフェストへの変更をコピーします。
古いプロジェクトの次のマニフェストに加えられた更新がある場合は、その変更を新しいプロジェクトの対応するファイルにコピーします。ファイルを直接上書きせずに、必要な変更を検査し、その変更を加えるようにしてください。

   - **tmp/build/Dockerfile** から **build/Dockerfile** へ
     - 新規プロジェクトのレイアウトには tmp ディレクトリーはありません。
   - RBAC ルールは **deploy/rbac.yaml** から **deploy/role.yaml** および **deploy/role_binding.yaml** に更新されます。
   - **deploy/cr.yaml** から **deploy/crds/<group>_<version>_<kind>_cr.yaml** へ
   - **deploy/crd.yaml** から **deploy/crds/<group>_<version>_<kind>_crd.yaml** へ

6. ユーザー定義の依存関係をコピーします。
古いプロジェクトの **Gopkg.toml** に追加されたユーザー定義の依存関係がある場合は、それをコピーして新しいプロジェクトの **Gopkg.toml** に追加します。**dep ensure** を実行し、新規プロジェクトのベンダーを更新します。

7. 変更を確認します。
Operator をビルドして実行し、動作することを確認します。