



OpenShift Container Platform 4.8

Operator

OpenShift Container Platform での Operator の使用

OpenShift Container Platform 4.8 Operator

OpenShift Container Platform での Operator の使用

法律上の通知

Copyright © 2023 Red Hat, Inc.

The text of and illustrations in this document are licensed by Red Hat under a Creative Commons Attribution–Share Alike 3.0 Unported license ("CC-BY-SA"). An explanation of CC-BY-SA is available at

<http://creativecommons.org/licenses/by-sa/3.0/>

. In accordance with CC-BY-SA, if you distribute this document or an adaptation of it, you must provide the URL for the original version.

Red Hat, as the licensor of this document, waives the right to enforce, and agrees not to assert, Section 4d of CC-BY-SA to the fullest extent permitted by applicable law.

Red Hat, Red Hat Enterprise Linux, the Shadowman logo, the Red Hat logo, JBoss, OpenShift, Fedora, the Infinity logo, and RHCE are trademarks of Red Hat, Inc., registered in the United States and other countries.

Linux[®] is the registered trademark of Linus Torvalds in the United States and other countries.

Java[®] is a registered trademark of Oracle and/or its affiliates.

XFS[®] is a trademark of Silicon Graphics International Corp. or its subsidiaries in the United States and/or other countries.

MySQL[®] is a registered trademark of MySQL AB in the United States, the European Union and other countries.

Node.js[®] is an official trademark of Joyent. Red Hat is not formally related to or endorsed by the official Joyent Node.js open source or commercial project.

The OpenStack[®] Word Mark and OpenStack logo are either registered trademarks/service marks or trademarks/service marks of the OpenStack Foundation, in the United States and other countries and are used with the OpenStack Foundation's permission. We are not affiliated with, endorsed or sponsored by the OpenStack Foundation, or the OpenStack community.

All other trademarks are the property of their respective owners.

概要

本書では、OpenShift Container Platform での Operator の使用方法について説明します。これには、クラスター管理者向けの Operator のインストールおよび管理方法についての説明や、開発者向けの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方法についての情報が含まれます。また、Operator SDK を使用して独自の Operator をビルドする方法についてのガイダンスも含まれます。

目次

第1章 OPERATOR の概要	4
1.1. 開発者の場合	4
1.2. 管理者の場合	4
1.3. 次のステップ	5
第2章 OPERATOR について	6
2.1. OPERATOR について	6
2.2. OPERATOR FRAMEWORK パッケージ形式	8
2.3. OPERATOR FRAMEWORK の一般的な用語の用語集	12
2.4. OPERATOR LIFECYCLE MANAGER (OLM)	14
2.5. OPERATORHUB について	45
2.6. RED HAT が提供する OPERATOR カタログ	47
2.7. CRD	48
第3章 ユーザータスク	57
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	57
3.2. NAMESPACE への OPERATOR のインストール	58
第4章 管理者タスク	65
4.1. OPERATOR のクラスターへの追加	65
4.2. インストール済み OPERATOR の更新	72
4.3. クラスターからの OPERATOR の削除	73
4.4. OPERATOR LIFECYCLE MANAGER でのプロキシサポートの設定	76
4.5. OPERATOR ステータスの表示	80
4.6. OPERATOR 条件の管理	83
4.7. クラスター管理者以外のユーザーによる OPERATOR のインストールの許可	85
4.8. カスタムカタログの管理	91
4.9. ネットワークが制限された環境での OPERATOR LIFECYCLE MANAGER の使用	103
第5章 OPERATOR の開発	117
5.1. OPERATOR SDK について	117
5.2. OPERATOR SDK CLI のインストール	118
5.3. 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	119
5.4. GO ベースの OPERATOR	134
5.5. ANSIBLE ベース OPERATOR	154
5.6. HELM ベースの OPERATOR	182
5.7. クラスターサービスバージョン (CSV) の定義	196
5.8. バンドルイメージの使用	221
5.9. スコアカードツールを使用した OPERATOR の検証	232
5.10. PROMETHEUS による組み込みモニタリングの設定	240
5.11. リーダー選択の設定	243
5.12. 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	244
5.13. OPERATOR SDK CLI リファレンス	247
第6章 クラスター OPERATOR のリファレンス	254
6.1. CLOUD CREDENTIAL OPERATOR	254
6.2. CLUSTER AUTHENTICATION OPERATOR	254
6.3. CLUSTER AUTOSCALER OPERATOR	255
6.4. CLUSTER CONFIG OPERATOR	255
6.5. CLUSTER CSI SNAPSHOT CONTROLLER OPERATOR	255
6.6. CLUSTER IMAGE REGISTRY OPERATOR	255
6.7. CLUSTER MACHINE APPROVER OPERATOR	256

6.8. クラスターモニタリング OPERATOR	256
6.9. CLUSTER NETWORK OPERATOR	257
6.10. OPENSIFT CONTROLLER MANAGER OPERATOR	257
6.11. CLUSTER SAMPLES OPERATOR	257
6.12. CLUSTER STORAGE OPERATOR	258
6.13. CLUSTER VERSION OPERATOR	259
6.14. CONSOLE OPERATOR	259
6.15. DNS OPERATOR	259
6.16. ETCD CLUSTER OPERATOR	259
6.17. INGRESS OPERATOR	260
6.18. INSIGHTS OPERATOR	261
6.19. KUBERNETES API SERVER OPERATOR	261
6.20. KUBERNETES CONTROLLER MANAGER OPERATOR	262
6.21. KUBERNETES SCHEDULER OPERATOR	262
6.22. KUBERNETES STORAGE VERSION MIGRATOR OPERATOR	263
6.23. MACHINE API OPERATOR	263
6.24. MACHINE CONFIG OPERATOR	263
6.25. MARKETPLACE OPERATOR	264
6.26. NODE TUNING OPERATOR	264
6.27. OPENSIFT API SERVER OPERATOR	264
6.28. OPERATOR LIFECYCLE MANAGER OPERATOR	265
6.29. OPENSIFT SERVICE CA OPERATOR	267
6.30. VSPHERE PROBLEM DETECTOR OPERATOR	268

第1章 OPERATOR の概要

Operator は OpenShift Container Platform 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。

どちらも同様の Operator の概念と目標に従いますが、OpenShift Container Platform 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。

- 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 SDKCLI をインストールする](#)。
- [Go ベースの Operator](#)、[Ansible ベースの Operator](#)、および[Helm](#)ベースの Operator を作成する。
- [Operator SDK](#) を使用して、Operator をビルド、テスト、およびデプロイする。
- [Operator をインストールして namespace にサブスクライブする](#)。
- [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する](#)。

1.2. 管理者の場合

クラスター管理者は、次の Operator タスクを実行できます。

- [カスタムカタログを管理する](#)
- [クラスター管理者以外のユーザーによる Operator のインストールの許可](#)
- [Operator Hub から Operator をインストールする](#)
- [Operator のステータスを表示する](#)
- [Operator の状態を管理する](#)
- [インストールされている Operator をアップグレードする](#)

- [インストールされている Operator を削除する](#)
- [プロキシーサポートを設定する](#)
- [ネットワークが制限された環境での Operator Lifecycle Manager の使用](#)

Red Hat が提供するクラスター Operator の詳細は、[クラスター Operators リファレンス](#) を参照してください。

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拡張機能のように動作し、(OpenShift Container Platform などの) Kubernetes 環境を監視し、その最新状態に基づいてリアルタイムの意思決定を行います。高度な Operator はアップグレードをシームレスに実行し、障害に自動的に対応するように設計されており、時間の節約のためにソフトウェアのバックアッププロセスを省略するなどのショートカットを実行することはありません。

技術的に言うと、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展開する場所。

Kubernetes にデプロイする理由

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

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

これらの API は機能的に充実しており、すべてのプラットフォームのクライアントを持ち、クラスターのアクセス制御/監査機能にプラグインします。Operator は Kubernetes の拡張メカニズム、カスタムリソース定義 (CRD、Custom Resource Definition) を使用するの、 **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せん。

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設定されています。

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 Container Platform 4.8 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ンネルについての 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 Container Platform ではデフォルトでデプロイされます。

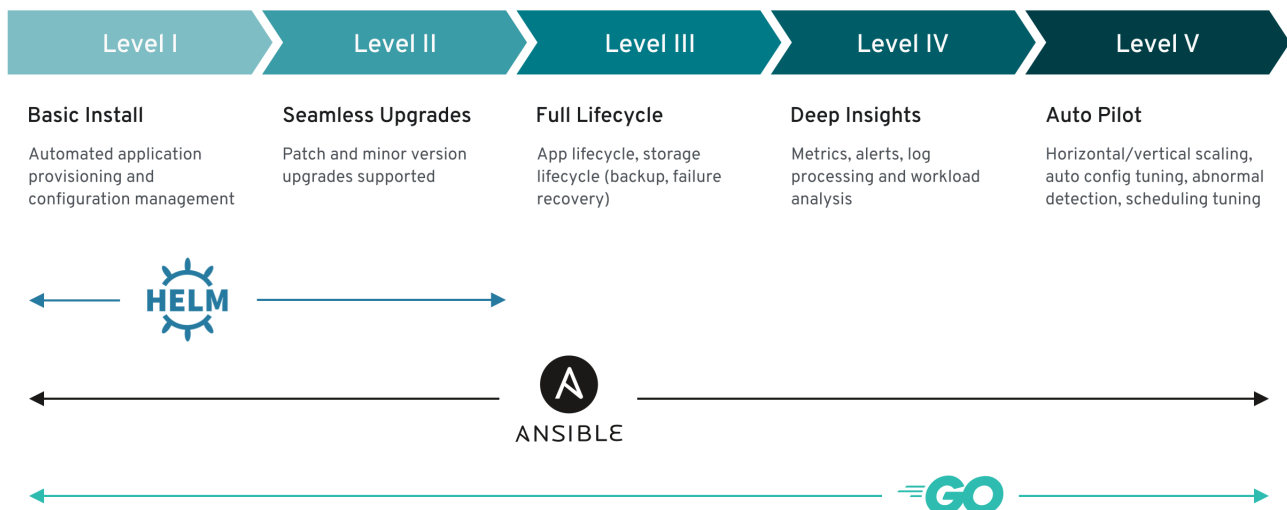
これらのツールは組み立て可能なツールとして設計されているため、役に立つと思われるツールを使用できます。

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

以下で、OpenShift Container Platform の Operator Lifecycle Manager (OLM) によってサポートされる Operator のパッケージ形式について説明します。



注記

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 Container Platform 4.8 以降で削除されます。パッケージマニフェスト形式の既存 Operator プロジェクトは、Operator SDK の **pkgman-to-bundle** コマンドを使用してバンドル形式に移行できます。詳細は、[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](#) を参照してください。

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イメージである **バンドルイメージ** としてコンテナ化され、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つ以上のチャン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**
- **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" ❶
  operators.operatorframework.io.bundle.manifests.v1: "manifests/" ❷
  operators.operatorframework.io.bundle.metadata.v1: "metadata/" ❸
  operators.operatorframework.io.bundle.package.v1: "test-operator" ❹
  operators.operatorframework.io.bundle.channels.v1: "beta,stable" ❺
  operators.operatorframework.io.bundle.channel.default.v1: "stable" ❻
```

- ❶ 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。
- ❷ 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。
- ❸ 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。
- ❹ バンドルのパッケージ名。
- ❺ 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ンネルの一覧。

- 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ンネル。



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ように権威を持つファイルになります。

2.2.1.3. 依存関係ファイル

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。Operator の依存関係には、サポートされる 2 つのタイプがあります。

- **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**: **gvk** 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。

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 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に相当する **index** と呼ばれるバンドルの一覧から 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、**インデックスイメージ** というコンテナイメージをコンテナ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。

インデックスには、コンテナ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 Container Platform では、Operator Lifecycle Manager (OLM) はインデックスイメージを **CatalogSource** オブジェクトで参照し、これをカタログとして使用できます。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については、[CLI ツール](#) を参照してください。

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、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です。

2.3.1.4. チャンネル

チャンネル 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。ヘッドはそのチャンネルの最新バージョンを参照します。たとえば **stable** チャンネルには、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と編成されます。

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

2.3.1.5. チャンネルヘッド

チャンネルヘッド は、特定のチャン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、またはアップグレードするために作成されるリソースの計算された一覧です。

2.3.1.10. 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 の一覧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。

2.3.1.11. 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。

2.3.1.12. 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。

2.3.1.13. 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。

2.3.1.14. 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)

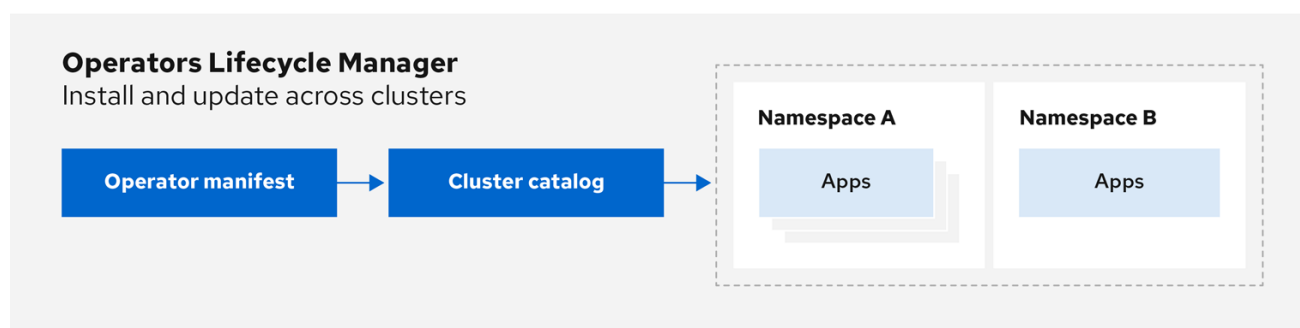
2.4.1. Operator Lifecycle Manager の概念およびリソース

以下で、OpenShift Container Platform での Operator Lifecycle Manager (OLM) に関連する概念について説明します。

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 Container Platform 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 Container Platform 4.8 でデフォルトで実行されます。これは、クラスター管理者がクラスターで実行されている Operator をインストールし、アップグレードし、アクセスをこれに付与するのに役立ちます。OpenShift Container Platform Web コンソールでは、クラスター管理者が Operator をインストールし、特定のプロジェクトアクセスを付与して、クラスターで利用可能な Operator のカタログを使用するための管理画面を利用できます。

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2.1 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。

リソース	短縮名	説明
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。
サブスクリプション	sub	パッケージのチャンネルを追跡して 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、またはアップグレードするために作成されるリソースの計算された一覧。
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 の一覧またはクラスター全体でカスタムリソース (CR) を監視できるように設定します。
OperatorConditions	-	OLM とそれが管理する Operator との間で通信チャンネルを作成します。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 Container Platform 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 Container Platform Web コンソールの OperatorHub は、カタログソースで提供される Operator も表示します。

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2.1 CatalogSource オブジェクトの例

```
apiVersion: operators.coreos.com/v1alpha1
kind: CatalogSource
metadata:
  generation: 1
  name: example-catalog ❶
  namespace: openshift-marketplace ❷
spec:
  displayName: Example Catalog ❸
  image: quay.io/example-org/example-catalog:v1 ❹
  priority: -400 ❺
  publisher: Example Org
  sourceType: grpc ❻
  updateStrategy:
    registryPoll: ❼
    interval: 30m0s
status:
  connectionState:
    address: example-catalog.openshift-marketplace.svc:50051
    lastConnect: 2021-08-26T18:14:31Z
    lastObservedState: READY ❽
  latestImageRegistryPoll: 2021-08-26T18:46:25Z ❾
  registryService: ❿
    createdAt: 2021-08-26T16:16:37Z
    port: 50051
    protocol: grpc
    serviceName: example-catalog
    serviceNamespace: openshift-marketplace
```

- ❶ **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。
- ❷ 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。
- ❸ Web コンソールおよび CLI でのカタログの表示名。
- ❹ カタログのインデックスイメージ。
- ❺ 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。
- ❻ 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。

7 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。

8 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。

9 カタログイメージを保存するコンテナ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。

10 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2.2 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](#)
- [カタログの優先順位](#)
- [CLI を使った Operator カタログソースのステータス表示](#)

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 パッケージのチャンネルや、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的に設定された場合、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行われるようにします。

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** などのチャンネルは、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ンネルの名前は Operator 間で異なる可能性があります、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、チャンネル名は 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。

OpenShift Container Platform Web コンソールから簡単に表示されるだけでなく、関連するサブスクリプションのステータスを確認して、Operator の新規バージョンが利用可能になるタイミングを特定できます。**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2.3 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: webserver.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
...

```

関連情報

- [クラスター管理者以外のユーザーによる Operator のインストールの許可](#)

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

以下では、OpenShift Container Platform における Operator Lifecycle Manager (OLM) のコンポーネントのアーキテクチャーを説明します。

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2.2 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、またはアップグレードするために作成されるリソースの計算された一覧。
CatalogSource	catsrc	カタログ	CSV、CRD、およびアプリケーションを定義するパッケージのリポジトリ。
サブスクリプション	sub	カタログ	パッケージのチャンネルを追跡して 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 の一覧またはクラスター全体でカスタムリソース (CR) を監視できるように設定します。

リソース	短縮名	所有する Operator	説明
------	-----	---------------	----

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2.3 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ン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。

チャン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、チャンネル、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。更新が見つかったら、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. 管理対象または必須の CRD のそれぞれについて、これを解決済みリソースとして追加します。
 - 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atalog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ン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、チャンネルは特定の 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 をチャン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

以下では、OpenShift Container Platform における Operator Lifecycle Manager (OLM) のワークロードについて説明します。

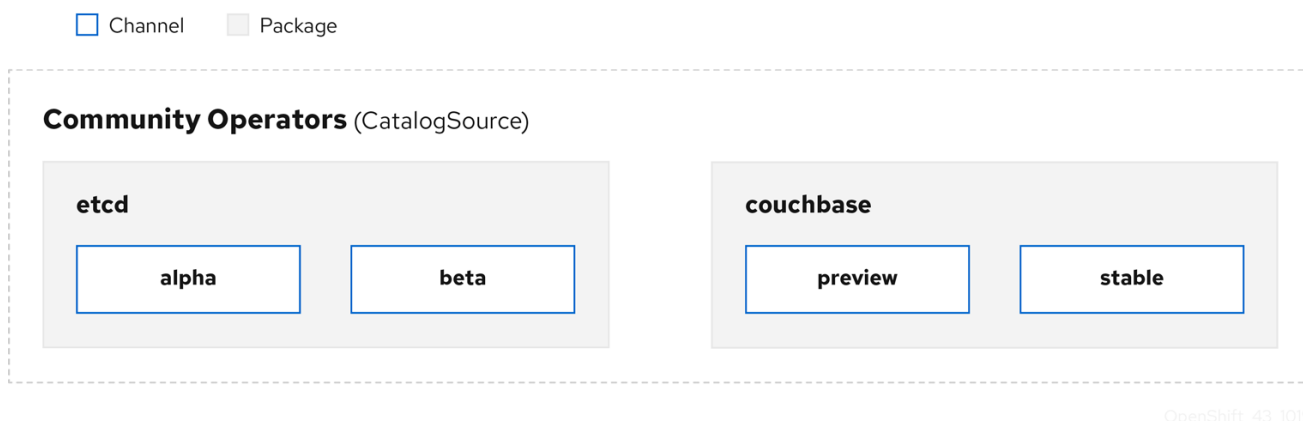
2.4.3.1. OLM での Operator のインストールおよびアップグレードのワークフロー

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**

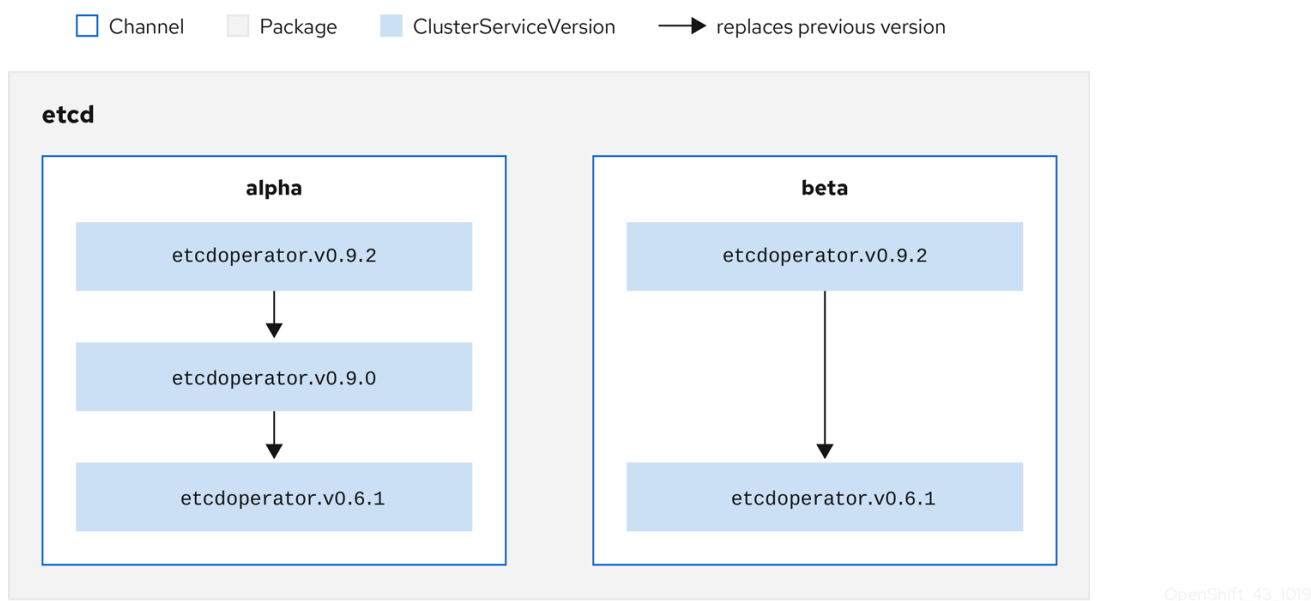
CSV で定義される Operator メタデータは、カタログソースというコレクションに保存できます。OLM はカタログソースを使用します。これは [Operator Registry API](#) を使用して利用可能な Operator やインストールされた Operator のアップグレードについてクエリーします。

図2.3 カタログソースの概要



カタログソース内で、Operator は **パッケージ** と **チャンネル** という更新のストリームに編成されます。これは、Web ブラウザーのような継続的なリリースサイクルの OpenShift Container Platform や他のソフトウェアで使用さ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ンネル



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ンネルを指定できます (例: **etcd** パッケージおよびその **alpha** チャンネル)。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。

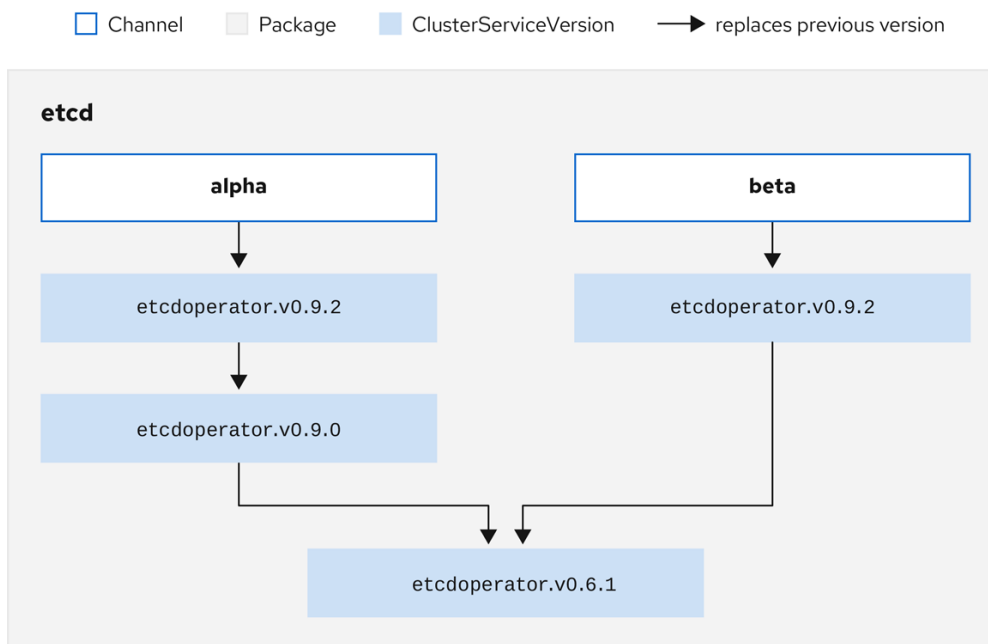


注記

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 リポジトリの場合と同様に、チャンネルの **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、更新がチャンネル間で共有されます。チャンネルは、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 利用可能なチャンネル更新についての OLM グラフ



OpenShift_43_1019

パッケージのチャンネルの例

```

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、チャンネルおよび 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ン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、チャンネルヘッドから 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、チャン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** はチャン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。

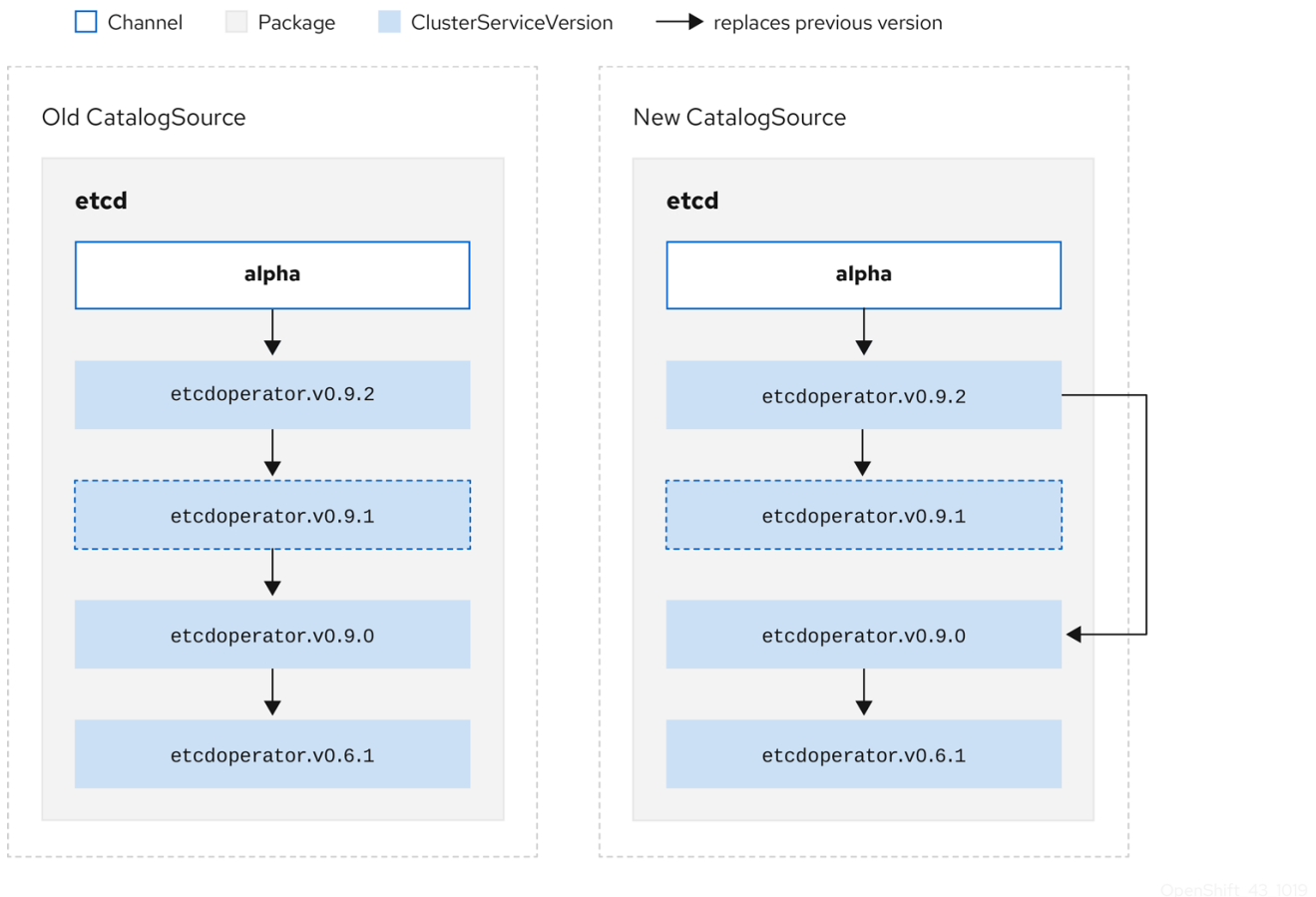
OLM は、新規カタログを送り、**省略された**リリースを追加することで、クラスターの状態や問題のある更新が発見されたかどうかにかかわらず、単一の固有の更新を常に取得することができます。

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ンネルヘッド (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

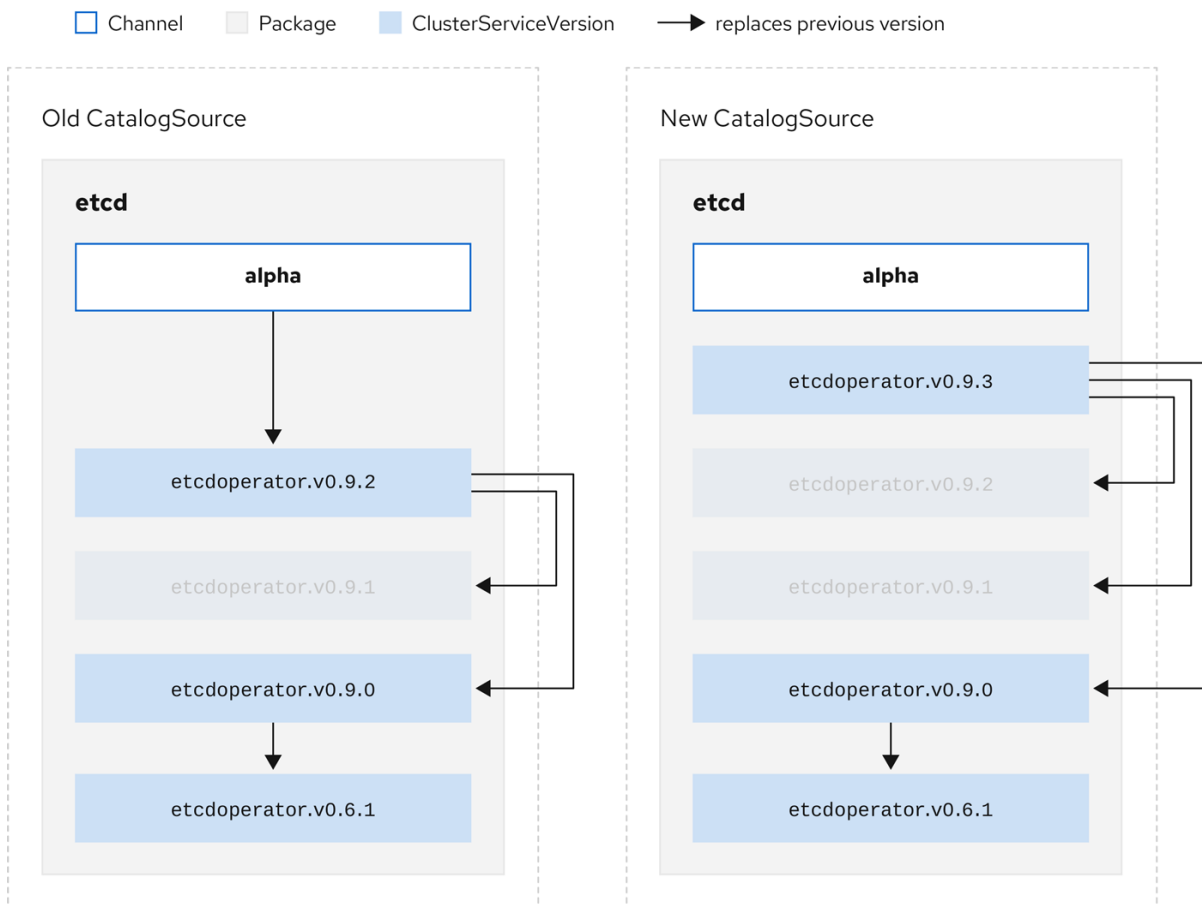
```
apiVersion: operators.coreos.com/v1alpha1
kind: ClusterServiceVersion
metadata:
  name: elasticsearch-operator.v4.1.2
  namespace: <namespace>
  annotations:
    olm.skipRange: '>=4.1.0 <4.1.2'
```

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

以下で、OpenShift Container Platform の Operator Lifecycle Manager (OLM) での依存関係の解決およびカスタムリソース定義 (CRD) アップグレードライフサイクルについて説明します。

2.4.4.1. 依存関係の解決

OLM は、実行中の Operator の依存関係の解決およびアップグレード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** などの他のオペレーティングシステム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。Operator 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。

つまり、OLM は以下を行うことができません。

- 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

2.4.4.2. 依存関係ファイル

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。Operator の依存関係には、サポートされる2つのタイプがあります。

- **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**: **gvk** 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。

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
```

2.4.4.3. 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。

2.4.4.3.1. カタログの優先順位

OpenShift Container Platform クラスターでは、OLM はカタログソースを読み取り、インストールに使用できる Operator 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
  image: example.com/my/operator-index:v1
  displayName: "My Operators"
  priority: 100
```

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。

2.4.4.3.2. チャネルの順序付け

カタログの Operator パッケージは、ユーザーが OpenShift Container Platform クラスターで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。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ンネルがあり、これは常にデフォルト以外のチャンネルよりも優先されます。デフォルトチャン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、チャンネル名の辞書式順序 (lexicographic order) で残りのチャンネルから検討されます。

2.4.4.3.3. チャンネル内での順序

ほとんどの場合、単一のチャン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ンネルの 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、それらはどのチャン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つのチャンネルだけに検索が絞られます。ただし、チャンネル内では、多くの Operator が依存関係を満たす可能性があります。

チャンネル内では、更新グラフでより上位にある新規 Operator が優先されます。チャン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。

2.4.4.3.4. 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が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。

2.4.4.3.4.1. 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、または既存 Operator の更新された状態を維持することを宣言します。

2.4.4.3.4.2. 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。

2.4.4.4. 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](#)

2.4.4.5. 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、または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 のチャン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](#)

2.4.4.6. 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。

2.4.4.7. 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

以下では、OpenShift Container Platform で Operator Lifecycle Manager (OLM) を使用した Operator グループの使用について説明します。

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** フィールドで設定されます。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2.4 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。

InstallMode タイプ	説明
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 を一覧表示したり、**spec.selector** でラベルセ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。
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 選択を一覧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>** で設定される文字列です。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
<operatorgroup_name>-admin	olm.opgroup.permissions/aggregate-to-admin: <operatorgroup_name>
<operatorgroup_name>-edit	olm.opgroup.permissions/aggregate-to-edit: <operatorgroup_name>
<operatorgroup_name>-view	olm.opgroup.permissions/aggregate-to-view: <operatorgroup_name>

以下の RBAC リソースは、CSV が **AllNamespaces** 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2.5 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 <ul style="list-style-type: none"> ● * 集計ラベル: <ul style="list-style-type: none"> ● 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
<code><kind>.<group>-<version>-edit</code>	<p><code><kind></code> の動詞</p> <ul style="list-style-type: none"> • <code>create</code> • <code>update</code> • <code>patch</code> • <code>delete</code> <p>集計ラベル:</p> <ul style="list-style-type: none"> • <code>rbac.authorization.k8s.io/aggregate-to-edit: true</code> • <code>olm.opgroup.permissions/aggregate-to-edit: <operatorgroup_name></code>
<code><kind>.<group>-<version>-view</code>	<p><code><kind></code> の動詞</p> <ul style="list-style-type: none"> • <code>get</code> • <code>list</code> • <code>watch</code> <p>集計ラベル:</p> <ul style="list-style-type: none"> • <code>rbac.authorization.k8s.io/aggregate-to-view: true</code> • <code>olm.opgroup.permissions/aggregate-to-view: <operatorgroup_name></code>
<code><kind>.<group>-<version>-view-crdview</code>	<p>Verbs on <code>apiextensions.k8s.io customresourcedefinitions <crd-name></code>:</p> <ul style="list-style-type: none"> • <code>get</code> <p>集計ラベル:</p> <ul style="list-style-type: none"> • <code>rbac.authorization.k8s.io/aggregate-to-view: true</code> • <code>olm.opgroup.permissions/aggregate-to-view: <operatorgroup_name></code>

表2.6 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
----------	----

クラスターロール	設定
<code><kind>.<group>-<version>-admin</code>	<p><code><kind></code> の動詞</p> <ul style="list-style-type: none"> ● * <p>集計ラベル:</p> <ul style="list-style-type: none"> ● <code>rbac.authorization.k8s.io/aggregate-to-admin: true</code> ● <code>olm.opgroup.permissions/aggregate-to-admin: <operatorgroup_name></code>
<code><kind>.<group>-<version>-edit</code>	<p><code><kind></code> の動詞</p> <ul style="list-style-type: none"> ● <code>create</code> ● <code>update</code> ● <code>patch</code> ● <code>delete</code> <p>集計ラベル:</p> <ul style="list-style-type: none"> ● <code>rbac.authorization.k8s.io/aggregate-to-edit: true</code> ● <code>olm.opgroup.permissions/aggregate-to-edit: <operatorgroup_name></code>
<code><kind>.<group>-<version>-view</code>	<p><code><kind></code> の動詞</p> <ul style="list-style-type: none"> ● <code>get</code> ● <code>list</code> ● <code>watch</code> <p>集計ラベル:</p> <ul style="list-style-type: none"> ● <code>rbac.authorization.k8s.io/aggregate-to-view: true</code> ● <code>olm.opgroup.permissions/aggregate-to-view: <operatorgroup_name></code>

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、またはそれらのソース 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。

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 がブ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

OpenShift Container Platform では、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する場合のサポートは限定されます。Operator はコントロールプレーンの拡張機能です。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。そのため、マルチテナント環境のテナントも Operator を共有する必要があります。

Operator Lifecycle Manager(OLM) は、複数の異なる namespace に Operator を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 の異なるメジャーバージョンには、互換性のないカスタムリソース定義 (CRD) が含まれることがよくあります。これが原因で、OLM を迅速に検証することが困難になります。

2.4.5.10.1. 関連情報

- [クラスター管理者以外のユーザーによる Operator のインストールの許可](#)

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、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。

インストール計画で無効な Operator グループが検出されると、CSV は生成されず、**InstallPlan** リソースは関連するメッセージを出して失敗します。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、または 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. 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。

2.4.6.1. 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。

2.4.6.2. 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。

2.4.6.2.1. 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があるカスタムリソース (CR) の移行を実行している場合

Upgradeable Operator 条件の例

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

- ❶ 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。

2.4.6.3. 関連情報

- [Operator 条件の管理](#)
- [Operator 条件の有効化](#)

2.4.7. Operator Lifecycle Manager メトリック

2.4.7.1. 公開されるメトリック

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

表2.7 OLM によって公開されるメトリック

名前	説明
catalog_source_count	カタログソースの数。
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、このメトリック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、またはそうでないか (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。
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。

2.4.8. 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 開発者がそれぞれの Operator の Webhook を定義する方法や OLM で実行される際の考慮事項についての詳細は、[クラスターサービスバージョン \(CSV\) の生成](#) を参照してください。

2.4.8.1. 関連情報

- [Webhook 受付プラグインのタイプ](#)
- 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 Container Platform の Web コンソールインターフェイスであり、これを使用してクラスター管理者は 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 Operator	redhat-openshift-ecosystem/community-operators-prod/operators GitHub リポジトリに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 Container Platform 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。このリソースを変更して、デフォルトのカatalogを有効または無効にすることができます。これは、ネットワークが制限された環境で OpenShift Container Platform を設定する際に役立ちます。

OperatorHub カスタムリソースの例

```
apiVersion: config.openshift.io/v1
kind: OperatorHub
metadata:
  name: cluster
spec:
  disableAllDefaultSources: true ❶
  sources: [ ❷
    {
      name: "community-operators",
      disabled: false
    }
  ]
```

❶ **disableAllDefaultSources** は、OpenShift Container Platform のインストール時にデフォルトで設定されるすべてのデフォルトカatalogの可用性を制御するオーバーライドです。

❷ ソースごとに **disabled** パラメーター値を変更して、デフォルトのカatalogを個別に無効にします。

2.5.3. 関連情報

- [カatalog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

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。OpenShift Container Platform 4.6 の時点で、Red Hat が提供するカタログは **インデックスイメージ** を使用して提供されています。

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。さらに OLM がネットワークが制限された環境の OpenShift Container Platform クラスターで実行される場合、最新のコンテンツをプルするためにインターネットからカタログに直接アクセス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、今後のリリースで削除される特定の API を定期的に非推奨にします。その結果、Operator は API を削除した Kubernetes バージョンを使用する OpenShift Container Platform のバージョン以降、削除された API を使用できなくなります。

クラスターがカスタムカタログを使用している場合に、Operator の作成者がプロジェクトを更新してワークロードの問題や、互換性のないアップグレードを回避できるようにする方法については [Operator の互換性の OpenShift Container Platform バージョンへの制御](#) を参照してください。



注記

レガシー形式をしようしたカスタムのカタログなど、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 Container Platform 4.8 以降で削除されます。

カスタムカタログイメージを作成する場合、OpenShift Container Platform 4 の以前のバージョンでは、複数のリリースで非推奨となった **oc adm catalog build** コマンドの使用が必要でしたが、これは削除されました。OpenShift Container Platform 4.6 以降で Red Hat が提供するインデックスイメージが利用可能になると、カタログビルダーは **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](#)
- [ネットワークが制限された環境での Operator Lifecycle Manager の使用](#)

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
redhat-operators	registry.redhat.io/redhat/redhat-operator-index:v4.8	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:v4.8	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:v4.8	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:v4.8	redhat-openshift-ecosystem/community-operators-prod/operators GitHub リポジトリで、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。

2.7. CRD

2.7.1. カスタムリソース定義による Kubernetes API の拡張

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、Operator によって管理されるカスタムオブジェクトは、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。以下では、CRD を作成し、管理することで、クラスター管理者が OpenShift Container Platform クラスターをどのように拡張できるかについて説明します。

2.7.1.1. 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。

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

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

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



注記

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

2.7.1.2. カスタムリソース定義の作成

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

前提条件

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

手順

CRD を作成するには、以下を実行します。

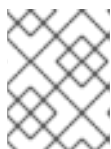
1. 以下の例のようなフィールドタイプを含む YAML ファイルを作成します。

CRD の YAML ファイルの例

```
apiVersion: apiextensions.k8s.io/v1 ❶
kind: CustomResourceDefinition
metadata:
  name: crontabs.stable.example.com ❷
spec:
  group: stable.example.com ❸
  versions:
    name: v1 ❹
  scope: Namespaced ❺
  names:
```

```
plural: crontabs 6
singular: crontab 7
kind: CronTab 8
shortNames:
- ct 9
```

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



注記

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

2. CRD オブジェクトを作成します。

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

新規の RESTful API エンドポイントは以下のように作成されます。

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

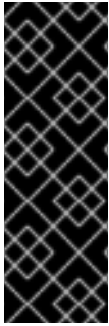
たとえば、サンプルファイルを使用すると、以下のエンドポイントが作成されます。

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

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

2.7.1.3. カスタムリソース定義のクラスターロールの作成

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



重要

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

前提条件

- CRD を作成します。

手順

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

カスタムロール定義の YAML ファイルサンプル

```
kind: ClusterRole
apiVersion: rbac.authorization.k8s.io/v1 ❶
metadata:
  name: aggregate-cron-tabs-admin-edit ❷
  labels:
    rbac.authorization.k8s.io/aggregate-to-admin: "true" ❸
    rbac.authorization.k8s.io/aggregate-to-edit: "true" ❹
rules:
- apiGroups: ["stable.example.com"] ❺
  resources: ["crontabs"] ❻
  verbs: ["get", "list", "watch", "create", "update", "patch", "delete", "deletecollection"] ❼
---
kind: ClusterRole
apiVersion: rbac.authorization.k8s.io/v1
metadata:
  name: aggregate-cron-tabs-view ❸
  labels:
    # Add these permissions to the "view" default role.
    rbac.authorization.k8s.io/aggregate-to-view: "true" ❹
    rbac.authorization.k8s.io/aggregate-to-cluster-reader: "true" ❺
rules:
- apiGroups: ["stable.example.com"] ❻
  resources: ["crontabs"] ❼
  verbs: ["get", "list", "watch"] ❺
```

- ❶ **rbac.authorization.k8s.io/v1** API を使用します。

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

2. クラスターロールを作成します。

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

2.7.1.4. ファイルからのカスタムリソースの作成

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

前提条件

- CRD がクラスター管理者によってクラスターに追加されている。

手順

- 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
```

- 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。
- 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がある条件を実装できるようにします。
- 5 オブジェクトのタイプに固有の条件を指定します。

2. ファイルの作成後に、オブジェクトを作成します。

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

2.7.1.5. カスタムリソースの検査

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

前提条件

- CR オブジェクトがアクセスできる namespace にあること。

手順

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

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

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' ❶
    image: my-awesome-cron-image ❷

```

❶ ❷ 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。

2.7.2. カスタムリソース定義からのリソースの管理

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

2.7.2.1. カスタムリソース定義

Kubernetes API では、**リソース** は特定の種類の API オブジェクトのコレクションを保管するエンドポイントです。たとえば、ビルトインされた **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。またクラスター管理者は、Operator のライフサイクル外にあるクラスターに CRD を手動で追加でき、これらをすべてのユーザーに利用可能にすることができます。



注記

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

2.7.2.2. 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、カスタムリソース (CR) は CR 仕様を使用するファイルを使って CLI で作成できます。

前提条件

- CRD がクラスター管理者によってクラスターに追加されている。

手順

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

CR の YAML ファイルサンプル

```
apiVersion: "stable.example.com/v1" ❶
kind: CronTab ❷
metadata:
  name: my-new-cron-object ❸
  finalizers: ❹
  - finalizer.stable.example.com
spec: ❺
  cronSpec: "* * * * /5"
  image: my-awesome-cron-image
```

- ❶ 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。
- ❷ CRD にタイプを指定します。
- ❸ オブジェクトの名前を指定します。
- ❹ 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がある条件を実装できるようにします。
- ❺ オブジェクトのタイプに固有の条件を指定します。

2. ファイルの作成後に、オブジェクトを作成します。

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

2.7.2.3. カスタムリソースの検査

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

前提条件

- CR オブジェクトがアクセスできる namespace にあること。

手順

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

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

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' ❶
    image: my-awesome-cron-image ❷
```

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

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 Container Platform 4.8 クラスターへのアクセス
- 管理者によってクラスター全体に etcd Operator がすでにインストールされている。

手順

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

ヒント

以下を使用して、CLI でこの一覧を取得できます。

```
$ oc get csv
```

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

5. **example** etcd クラスターをクリックしてから **Resources** タブをクリックして、プロジェクトに Operator によって自動的に作成され、設定された数多くのリソースが含まれることを確認します。
Kubernetes サービスが作成され、プロジェクトの他の Pod からデータベースにアクセスできることを確認します。
6. 所定プロジェクトで **edit** ロールを持つすべてのユーザーは、クラウドサービスのようにセルフサービス方式でプロジェクトにすでに作成されている Operator によって管理されるアプリケーションのインスタンス (この例では etcd クラスター) を作成し、管理し、削除することができます。この機能を持つ追加のユーザーを有効にする必要がある場合、プロジェクト管理者は以下のコマンドを使用してこのロールを追加できます。

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

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

3.2. NAMESPACE への OPERATOR のインストール

クラスター管理者が Operator のインストールパーミッションをお使いのアカウントに委任している場合、セルフサービス方式で Operator をインストールし、これを namespace にサブスクライブできます。

3.2.1. 前提条件

- クラスター管理者は、namespace へのセルフサービス Operator のインストールを許可するために OpenShift Container Platform ユーザーアカウントに特定のパーミッションを追加する必要があります。詳細は、[クラスター管理者以外のユーザーによる Operator のインストールの許可](#)を参照してください。

3.2.2. 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。

適切なパーミッションを持つユーザーとして、OpenShift Container Platform Web コンソールまたは CLI を使用して OperatorHub から Operator をインストールできます。

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

Operator をインストールする特定の namespace を選択します。

更新チャネル

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

承認ストラテジー

自動 (Automatic) または手動 (Manual) のいずれかの更新を選択します。

インストールされた Operator について自動更新を選択する場合、Operator の新規バージョンが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。

手動更新を選択する場合、Operator の新規バージョンが利用可能になると、OLM は更新要求を作成します。クラスター管理者は、Operator が新規バージョンに更新されるように更新要求を手動で承認する必要があります。

- [OperatorHub について](#)

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

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

前提条件

- Operator インストールパーミッションを持つアカウントを使用して OpenShift Container Platform クラスターにアクセスできる。

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、またはキーワードを **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. Operator をインストールする特定の単一 namespace を選択します。Operator は監視のみを実行し、この単一 namespace で使用されるように利用可能になります。
 - b. **Update Channel** を選択します (複数を選択できる場合)。
 - c. 前述のように、**自動 (Automatic)** または **手動 (Manual)** の承認ストラテジーを選択します。
6. **Install** をクリックし、Operator をこの OpenShift Container Platform クラスターの選択した namespace で利用可能にします。
 - 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。

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

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

前提条件

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

手順

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

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

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ンネルを確認します。

```
$ 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 ❶
spec:
  channel: <channel_name> ❷
```

```

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 **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。それ以外の場合は、**SingleNamespaces** 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ンネルの名前。
- 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atalogソースの namespace。デフォルトの OperatorHub カatalog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. **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](#)
- [チャンネル名](#)

3.2.5. 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

- Operator インストールパーミッションを持つアカウントを使用して OpenShift Container Platform クラスターにアクセスできる。
- OpenShift CLI (**oc**) がインストール済みであること。

手順

1. **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 3.4.0 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-v3.4
  installPlanApproval: Manual 1
  name: quay-operator
```

```
source: redhat-operators  
sourceNamespace: openshift-marketplace  
startingCSV: quay-operator.v3.4.0 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、承認戦略を **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。

2. **Subscription** オブジェクトを作成します。

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

3. 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](#)

第4章 管理者タスク

4.1. OPERATOR のクラスターへの追加

クラスター管理者は、OperatorHub を使用して Operator を namespace にサブスクライブすることで、Operator を OpenShift Container Platform クラスターにインストールすることができます。

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。

適切なパーミッションを持つユーザーとして、OpenShift Container Platform Web コンソールまたは CLI を使用して OperatorHub から Operator をインストールできます。

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

Operator をインストールする特定の namespace を選択します。

更新チャンネル

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

承認ストラテジー

自動 (Automatic) または手動 (Manual) のいずれかの更新を選択します。

インストールされた Operator について自動更新を選択する場合、Operator の新規バージョンが選択されたチャン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。

手動更新を選択する場合、Operator の新規バージョンが利用可能になると、OLM は更新要求を作成します。クラスター管理者は、Operator が新規バージョンに更新されるように更新要求を手動で承認する必要があります。

- [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 Container Platform Web コンソールを使用して OperatorHub から Operator をインストールし、これをサブスクライブできます。

前提条件

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

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、またはキーワードを **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. Operator をインストールする特定の単一 namespace を選択します。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。
 - c. **Update Channel** を選択します (複数を選択できる場合)。
 - d. 前述のように、**自動 (Automatic)** または **手動 (Manual)** の承認ストラテジーを選択します。
6. **Install** をクリックし、Operator をこの OpenShift Container Platform クラスターの選択した namespace で利用可能にします。
 - 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 Container Platform Web コンソールを使用する代わりに、CLI を使用して OperatorHub から Operator をインストールできます。**oc** コマンドを使用して、**Subscription** オブジェクトを作成または更新します。

前提条件

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

手順

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

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

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 ❶
spec:
  channel: <channel_name> ❷
  name: <operator_name> ❸
  source: redhat-operators ❹
  sourceNamespace: openshift-marketplace ❺
  config:
    env: ❻
    - 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 **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。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ンネルの名前。
- 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. **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

- Operator インストールパーミッションを持つアカウントを使用して OpenShift Container Platform クラスターにアクセスできる。
- OpenShift CLI (**oc**) がインストール済みであること。

手順

1. **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 3.4.0 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-v3.4
  installPlanApproval: Manual ❶
  name: quay-operator
  source: redhat-operators
  sourceNamespace: openshift-marketplace
  startingCSV: quay-operator.v3.4.0 ❷
```

- ❶ 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、承認戦略を **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。
- ❷ Operator CSV の特定バージョンを設定します。

2. **Subscription** オブジェクトを作成します。

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

3. 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](#)

4.1.5. 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、ノードセクター、ティント、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、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、マシンセットを使用するか、またはノードを直接編集してティントをノードに追加します。ティ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** ティントは、ティ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クターで設定され、ティ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、プロジェクトで **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

- ティントおよび容認の追加を [ノードに手動で実行](#)、または [マシンセットを使用する](#)
- [プロジェクトスコープのノードセクターの作成](#)

- [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](#)

4.2. インストール済み OPERATOR の更新

クラスター管理者は、OpenShift Container Platform クラスターで Operator Lifecycle Manager (OLM) を使用し、以前にインストールされた Operator を更新できます。

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ンネルを指定します。更新チャンネルを変更して、新しいチャン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ンネルの名前は Operator 間で異なる可能性があります。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、チャンネル名は 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、現在のチャンネルよりも古いチャン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 プラットフォーム Operator Update Information Checker](#)

このアプリケーションを使用して、Operator Lifecycle Manager ベースの Operator を検索し、OpenShift Container Platform の異なるバージョン間で更新チャンネルごとに利用可能な Operator 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 の更新チャンネルの変更

OpenShift Container Platform Web コンソールを使用して、Operator の更新チャン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、選択したチャンネルで新規 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. 更新チャンネルを変更する 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. **Channel** の下にある更新チャン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ンネルをクリックし、**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ン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 Container Platform 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、**アップグレードステータス** 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、OpenShift Container Platform クラスターで Operator Lifecycle Manager (OLM) を使用して、以前にインストールされた Operator をアップグレードする方法を説明します。

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

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

手順

1. **Operators** → **Installed Operators** ページからスクロールするか、または **Filter by name** にキーワードを入力して必要な Operator を見つけます。次に、それをクリックします。
2. **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、以下が通知されます。

Operator を削除しても、そのカスタムリソース定義や管理リソースは削除されません。Operator がクラスターにアプリケーションをデプロイしているか、またはクラスター外のリソースを設定している場合、それらは引き続き実行され、手動でクリーンアップする必要があります。

このアクションにより、Operator および Operator のデプロイメントおよび Pod が削除されます (ある場合)。CRD および CR を含む Operator によって管理される Operand およびリソースは削除されません。Web コンソールは、一部の Operator のダッシュボードおよびナビゲーションアイテムを有効にします。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。

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

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

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

手順

1. サブスクリプションされた Operator (例: **jaeger**) の現行バージョンを **currentCSV** フィールドで確認します。

```
$ oc get subscription jaeger -n openshift-operators -o yaml | grep currentCSV
```

出力例

```
currentCSV: jaeger-operator.v1.8.2
```

2. サブスクリプション (例: **jaeger**) を削除します。

```
$ oc delete subscription jaeger -n openshift-operators
```

出力例

```
subscription.operators.coreos.com "jaeger" deleted
```

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

```
$ oc delete clusterserviceversion jaeger-operator.v1.8.2 -n openshift-operators
```

出力例

```
clusterserviceversion.operators.coreos.com "jaeger-operator.v1.8.2" 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
```

- サブスクリプションを削除します。

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

- 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>
```

- 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
```

- 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。

- 設定マップを削除します。

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

- 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 でのプロキシサポートの設定

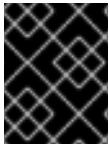
グローバルプロキシが OpenShift Container Platform クラスターで設定されている場合、Operator Lifecycle Manager (OLM) はクラスター全体のプロキシで管理する Operator を自動的に設定します。ただし、インストールされた Operator をグローバルプロキシを上書きするか、またはカスタム CA 証明書を挿入するように設定することもできます。

関連情報

- [クラスター全体のプロキシの設定](#)
- [カスタム PKI の設定](#) (カスタム CA 証明書)

4.4.1. Operator のプロキシ設定の上書き

クラスター全体の egress プロキシ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設定を継承します。クラスター管理者は、Operator のサブスクリプションを設定してこれらのプロキシ設定を上書きすることもできます。



重要

Operator は、管理対象オペランドの Pod でのプロキシ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

- **cluster-admin** 権限を持つアカウントを使用して OpenShift Container Platform クラスターにアクセスできる。

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@sha256:66a37fd61a06a43969854ee6d3e21088a98b93838e284a6086b13917f96bd9c
...
```

4.4.2. カスタム CA 証明書の挿入

クラスター管理者が設定マップ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 はユーザーによってプロビジョニングされる証明書およびシステム CA 証明書を単一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 プロキシがある場合に役立ちます。

前提条件

- **cluster-admin** パーミッションを持つアカウントを使用して OpenShift Container Platform クラスターにアクセスできる。
- 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 ❶
labels:
  config.openshift.io/inject-trusted-cabundle: "true" ❷
```

- ❶ 設定マップの名前。
- ❷ 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: ❶
  selector:
    matchLabels:
      <labels_for_pods> ❷
  volumes: ❸
  - name: trusted-ca
    configMap:
      name: trusted-ca
      items:
        - key: ca-bundle.crt ❹
          path: tls-ca-bundle.pem ❺
  volumeMounts: ❻
  - name: trusted-ca
    mountPath: /etc/pki/ca-trust/extracted/pem
    readOnly: true
```

- ❶ **config** セクションがない場合に、これを追加します。
- ❷ Operator が所有する Pod に一致するラベルを指定します。
- ❸ **trusted-ca** ボリュームを作成します。
- ❹ **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。

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。



注記

デフォルトの OpenShift Container Platform クラスター Operator は Cluster Version Operator (CVO) によって管理され、これらの 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、それらには **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

- cluster-admin** ロールを持つユーザーとしてクラスターにアクセスできる。
- 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一覧表示します。

```
$ 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

```
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 Container Platform クラスター Operator は Cluster Version Operator (CVO) によって管理され、これらの 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、それらには **Subscription** オブジェクトがあります。

4.5.3. CLI を使った 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 を使って確認できます。

前提条件

- **cluster-admin** ロールを持つユーザーとしてクラスターにアクセスできる。
- 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一覧表示します。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
...
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-vsxd/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-jyryyg2-f76d1-fgdbq-worker-b-vsxd
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](#)
- [プライベートレジストリーからの Operator のイメージへのアクセス](#)

4.6. OPERATOR 条件の管理

クラスター管理者は、Operator Lifecycle Manager (OLM) を使用して Operator 条件を管理できます。

4.6.1. Operator 条件の上書き

クラスター管理者には、Operator が報告するサポートされている Operator 条件を無視することをお勧めします。Operator 条件が存在する場合、**Spec.Overrides** 配列の Operator 条件は **Spec.Conditions** 配列の条件を上書きし、これによりクラスター管理者は、Operator が Operator Lifecycle Manager (OLM) に状態を誤って報告する状況に対応することができます。



注記

デフォルトでは、**Spec.Overrides**配列は、クラスター管理者によって追加されるまで、**Operator Condition**オブジェクトには存在しません。**Spec.Conditions**配列も、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お勧めします。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 条件を上書きすることによって実行できます。

前提条件

- OLM を使用してインストールされた **OperatorCondition** オブジェクトを含む

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 条件の上書きの例

```
apiVersion: operators.coreos.com/v1
kind: OperatorCondition
metadata:
  name: my-operator
  namespace: operators
spec:
  overrides:
    - type: Upgradeable 1
      status: "True"
      reason: "upgradelsSafe"
      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** クラスター管理者は、アップグレードの準備状態を **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 条件を Operator 作成者として設定するためのロジックの作成についての詳細は、Operator SDK ドキュメント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. クラスター管理者以外のユーザーによる OPERATOR のインストールの許可

クラスター管理者は、**Operator グループ** を使用して、通常のユーザーが Operator をインストールできるようにすることができます。

関連情報

- [Operator グループ](#)

4.7.1. Operator インストールポリシーについて

Operator の実行には幅広い権限が必要になる可能性があり、必要な権限はバージョン間で異なる場合があります。Operator Lifecycle Manager (OLM) は、**cluster-admin** 権限で実行されます。デフォルトで、Operator の作成者はクラスターサービスバージョン (CSV) で任意のパーミッションのセットを指定でき、OLM はこれを Operator に付与します。

Operator がクラスタースコープの権限を取得できず、ユーザーが OLM を使用して権限を昇格できないようにするために、クラスター管理者は Operator をクラスターに追加する前に手動で監査できます。また、クラスター管理者には、サービスアカウントを使用した Operator のインストールまたはアップグレード時に許可されるアクションを判別し、制限するための各種ツールが提供されます。

クラスター管理者は、一連の権限が付与されたサービスアカウントに Operator グループを関連付けることができます。サービスアカウントは、ロールベースのアクセス制御 (RBAC) ルールを使用して、事前に定義された境界内でのみ実行されるように、Operator にポリシーを設定します。その結果、Operator は、それらのルールによって明示的に許可されていないことはいずれも実行できません。

Operator グループを採用することで、十分な権限を持つユーザーは、限られた範囲で Operator をインストールできます。その結果、より多くの Operator Framework ツールをより多くのユーザーが安全に利用できるようになり、Operator を使用してアプリケーションを構築するためのより豊かなエクスペリエンスが提供されます。



注記

Subscription オブジェクトのロールベースのアクセス制御 (RBAC) は、namespace で **edit** または **admin** のロールを持つすべてのユーザーに自動的に付与されます。ただし、RBAC は **OperatorGroup** オブジェクトには存在しません。この不在が、通常のユーザーが Operator をインストールできない理由です。Operator グループを事前にインストールすることで、実質的にインストール権限が付与されます。

Operator グループをサービスアカウントに関連付ける際は、次の点に注意してください。

- **APIService** および **CustomResourceDefinition** リソースは、**cluster-admin** ロールを使用して OLM によって常に作成されます。Operator グループに関連付けられたサービスアカウントには、これらのリソースを作成するための権限を付与できません。
- この Operator グループに関連付けられる Operator は、指定されたサービスアカウントに付与されるパーミッションに制限されるようになりました。Operator がサービスアカウントの範囲外のアクセス許可を要求した場合、インストールは適切なエラーで失敗するため、クラスター管理者は問題をトラブルシューティングして解決できます。

4.7.1.1. インストールシナリオ

Operator をクラスターでインストールまたはアップグレードできるかどうかを決定する際に、Operator Lifecycle Manager (OLM) は以下のシナリオを検討します。

- クラスター管理者は新規の Operator グループを作成し、サービスアカウントを指定します。この Operator グループに関連付けられるすべての Operator がサービスアカウントに付与される権限に基づいてインストールされ、実行されます。
- クラスター管理者は新規の Operator グループを作成し、サービスアカウントを指定しません。OpenShift Container Platform は後方互換性を維持します。そのため、デフォルト動作はそのまま残り、Operator のインストールおよびアップグレードは許可されます。
- サービスアカウントを指定しない既存の Operator グループの場合、デフォルトの動作は残り、Operator のインストールおよびアップグレードは許可されます。
- クラスター管理者は既存の Operator グループを更新し、サービスアカウントを指定します。OLM により、既存の Operator は現在の権限で継続して実行されます。このような既存 Operator がアップグレードされる場合、これは再インストールされ、新規 Operator のようにサービスアカウントに付与される権限に基づいて実行されます。
- Operator グループで指定されるサービスアカウントは、パーミッションの追加または削除によって変更されるか、または既存のサービスアカウントは新しいサービスアカウントに切り替わります。既存の Operator がアップグレードされる場合、これは再インストールされ、新規 Operator のように更新されたサービスアカウントに付与される権限に基づいて実行されます。
- クラスター管理者は、サービスアカウントを Operator グループから削除します。デフォルトの動作は残り、Operator のインストールおよびアップグレードは許可されます。

4.7.1.2. インストールワークフロー

Operator グループがサービスアカウントに関連付けられ、Operator がインストールまたはアップグレードされると、Operator Lifecycle Manager (OLM) は以下のワークフローを使用します。

1. 指定された **Subscription** オブジェクトは OLM によって選択されます。
2. OLM はこのサブスクリプションに関連する Operator グループをフェッチします。
3. OLM は Operator グループにサービスアカウントが指定されていることを判別します。
4. OLM はサービスアカウントにスコープが設定されたクライアントを作成し、スコープ設定されたクライアントを使用して Operator をインストールします。これにより、Operator で要求されるパーミッションは常に Operator グループのそのサービスアカウントのパーミッションに制限されるようになります。
5. OLM は CSV で指定されたパーミッションセットを使用して新規サービスアカウントを作成し、これを Operator に割り当てます。Operator は割り当てられたサービスアカウントで実行されます。

4.7.2. Operator インストールのスコープ設定

Operator の Operator Lifecycle Manager (OLM) での Operator のインストールおよびアップグレードについてのスコープ設定ルールを提供するには、サービスアカウントを Operator グループに関連付けます。

この例では、クラスター管理者は一連の Operator を指定された namespace に制限できます。

手順

1. 新規の namespace を作成します。

```
$ cat <<EOF | oc create -f -
apiVersion: v1
kind: Namespace
metadata:
  name: scoped
EOF
```

2. Operator を制限する必要があるパーミッションを割り当てます。これには、新規サービスアカウント、関連するロール、およびロールバインディングの作成が必要になります。

```
$ cat <<EOF | oc create -f -
apiVersion: v1
kind: ServiceAccount
metadata:
  name: scoped
  namespace: scoped
EOF
```

以下の例では、単純化するために、サービスアカウントに対し、指定される namespace ですべてのを実行できるパーミッションを付与します。実稼働環境では、より粒度の細かいパーミッションセットを作成する必要があります。

```
$ cat <<EOF | oc create -f -
apiVersion: rbac.authorization.k8s.io/v1
kind: Role
metadata:
  name: scoped
  namespace: scoped
rules:
- apiGroups: ["*"]
  resources: ["*"]
  verbs: ["*"]
---
apiVersion: rbac.authorization.k8s.io/v1
kind: RoleBinding
metadata:
  name: scoped-bindings
  namespace: scoped
roleRef:
  apiGroup: rbac.authorization.k8s.io
  kind: Role
  name: scoped
subjects:
- kind: ServiceAccount
  name: scoped
  namespace: scoped
EOF
```

3. 指定された namespace に **OperatorGroup** オブジェクトを作成します。この Operator グループは指定された namespace をターゲットにし、そのテナンシーがこれに制限されるようにします。

さらに、Operator グループはユーザーがサービスアカウントを指定できるようにします。直前の手順で作成したサービスアカウントを指定します。

```
$ cat <<EOF | oc create -f -
apiVersion: operators.coreos.com/v1
kind: OperatorGroup
metadata:
  name: scoped
  namespace: scoped
spec:
  serviceAccountName: scoped
  targetNamespaces:
  - scoped
EOF
```

指定された namespace にインストールされる Operator はこの Operator グループに関連付けられ、指定されるサービスアカウントに関連付けられます。

4. 指定された namespace で **Subscription** オブジェクトを作成し、Operator をインストールします。

```
$ cat <<EOF | oc create -f -
apiVersion: operators.coreos.com/v1alpha1
kind: Subscription
metadata:
  name: etcd
  namespace: scoped
spec:
  channel: singlenamespace-alpha
  name: etcd
  source: <catalog_source_name> ❶
  sourceNamespace: <catalog_source_namespace> ❷
EOF
```

- ❶ 指定された namespace にすでにあるカタログソース、またはグローバルカタログ namespace にあるものを指定します。
- ❷ カatalogソースが作成された namespace を指定します。

この Operator グループに関連付けられる Operator は、指定されたサービスアカウントに付与されるパーミッションに制限されます。Operator がサービスアカウントの範囲外のパーミッションを要求する場合、インストールは関連するエラーを出して失敗します。

4.7.2.1. 粒度の細かいパーミッション

Operator Lifecycle Manager (OLM) は Operator グループで指定されたサービスアカウントを使用し、インストールされる Operator に関連する以下のリソースを作成または更新します。

- **ClusterServiceVersion**
- **サブスクリプション**
- **Secret**
- **ServiceAccount**

- **Service**
- **ClusterRole** および **ClusterRoleBinding**
- **Role** および **RoleBinding**

Operator を指定された namespace に制限するため、クラスター管理者は以下のパーミッションをサービスアカウントに付与して起動できます。



注記

以下のロールは一般的なサンプルであり、特定の Operator に基づいて追加のルールが必要になる可能性があります。

```
kind: Role
rules:
- apiGroups: ["operators.coreos.com"]
  resources: ["subscriptions", "clusterserviceversions"]
  verbs: ["get", "create", "update", "patch"]
- apiGroups: [""]
  resources: ["services", "serviceaccounts"]
  verbs: ["get", "create", "update", "patch"]
- apiGroups: ["rbac.authorization.k8s.io"]
  resources: ["roles", "rolebindings"]
  verbs: ["get", "create", "update", "patch"]
- apiGroups: ["apps"] ❶
  resources: ["deployments"]
  verbs: ["list", "watch", "get", "create", "update", "patch", "delete"]
- apiGroups: [""] ❷
  resources: ["pods"]
  verbs: ["list", "watch", "get", "create", "update", "patch", "delete"]
```

❶❷ ここで、デプロイメントおよび Pod などの他のリソースを作成するためのパーミッションを追加します。

さらに、Operator がプルシークレットを指定する場合、以下のパーミッションも追加する必要があります。

```
kind: ClusterRole ❶
rules:
- apiGroups: [""]
  resources: ["secrets"]
  verbs: ["get"]
---
kind: Role
rules:
- apiGroups: [""]
  resources: ["secrets"]
  verbs: ["create", "update", "patch"]
```

❶ シークレットを OLM namespace から取得するために必要です。

4.7.3. Operator カタログのアクセス制御

Operator カタログがグローバルカタログ namespace **openshift-marketplace** で作成されると、カタログの Operator がクラスター全体ですべての namespace で使用できるようになります。他の namespace で作成されたカタログは、カタログの同じ namespace でのみ Operator を使用できるようにします。

クラスター管理者以外のユーザーに Operator のインストール権限が委任されているクラスターでは、クラスター管理者は、それらのユーザーがインストールできる Operator のセットをさらに制御または制限しないといけない場合があります。これは、次のアクションで実現できます。

1. デフォルトのグローバルカタログをすべて無効にします。
2. 関連する Operator グループがプリインストールされているのと同じ namespace で、キュレートされたカスタムカタログを有効にします。

関連情報

- [デフォルトの OperatorHub ソースの無効化](#)
- [クラスターへのカタログソースの追加](#)

4.7.4. パーミッションに関する失敗のトラブルシューティング

パーミッションがないために Operator のインストールが失敗する場合は、以下の手順を使用してエラーを特定します。

手順

1. **Subscription** オブジェクトを確認します。このステータスには、Operator の必要な **[Cluster]Role[Binding]** オブジェクトの作成を試行した **InstallPlan** オブジェクトをポイントするオブジェクト参照 **installPlanRef** があります。

```
apiVersion: operators.coreos.com/v1
kind: Subscription
metadata:
  name: etcd
  namespace: scoped
status:
  installPlanRef:
    apiVersion: operators.coreos.com/v1
    kind: InstallPlan
    name: install-4plp8
    namespace: scoped
    resourceVersion: "117359"
    uid: 2c1df80e-afea-11e9-bce3-5254009c9c23
```

2. **InstallPlan** オブジェクトのステータスでエラーの有無を確認します。

```
apiVersion: operators.coreos.com/v1
kind: InstallPlan
status:
  conditions:
    - lastTransitionTime: "2019-07-26T21:13:10Z"
      lastUpdateTime: "2019-07-26T21:13:10Z"
```

```

message: 'error creating clusterrole etcdoperator.v0.9.4-clusterwide-dsfx4:
clusterroles.rbac.authorization.k8s.io
  is forbidden: User "system:serviceaccount:scoped:scoped" cannot create resource
  "clusterroles" in API group "rbac.authorization.k8s.io" at the cluster scope'
reason: InstallComponentFailed
status: "False"
type: Installed
phase: Failed

```

エラーメッセージは、以下を示しています。

- リソースの API グループを含む、作成に失敗したリソースのタイプ。この場合、これは **rbac.authorization.k8s.io** グループの **clusterroles** です。
- リソースの名前。
- エラーのタイプ: **is forbidden** は、ユーザーに操作を実行するための十分なパーミッションがないことを示します。
- リソースの作成または更新を試みたユーザーの名前。この場合、これは Operator グループで指定されたサービスアカウントを参照します。
- 操作の範囲が **cluster scope** かどうか。
ユーザーは、不足しているパーミッションをサービスアカウントに追加してから、繰り返すことができます。



注記

現時点で、Operator Lifecycle Manager (OLM) は最初の試行でエラーの詳細の一覧を提供しません。

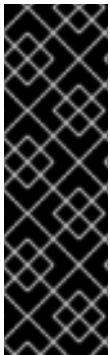
4.8. カスタムカタログの管理

以下では、OpenShift Container Platform で Operator Lifecycle Manager (OLM) の [Bundle Format](#) を使用してパッケージ化された Operator のカスタムパッケージを使用する方法について説明します。



注記

レガシー形式をしようしたカスタムのカタログなど、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 Container Platform 4.8 以降で削除されます。



重要

Kubernetes は、今後のリリースで削除される特定の API を定期的に非推奨にします。その結果、Operator は API を削除した Kubernetes バージョンを使用する OpenShift Container Platform のバージョン以降、削除された API を使用できなくなります。

クラスターがカスタムカタログを使用している場合に、Operator の作成者がプロジェクトを更新してワークロードの問題や、互換性のないアップグレードを回避できるようにする方法については [Operator の互換性の OpenShift Container Platform バージョンへの制御](#) を参照してください。

関連情報

- [Red Hat が提供する Operator カタログ](#)

4.8.1. 前提条件

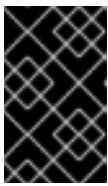
- **opm CLI** をインストールします。

4.8.2. インデックスイメージの作成

opm CLI を使用してインデックスイメージを作成できます。

前提条件

- **opm** version 1.12.3+
- **podman** version 1.9.3+
- **Docker v2-2** をサポートするレジストリーにビルドされ、プッシュされるバンドルイメージ。



重要

OpenShift Container Platform クラスターの内部レジストリーはターゲットレジストリーとして使用できません。これは、ミラーリングプロセスで必要となるタグを使わないプッシュをサポートしないためです。

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> \
  --tag <registry>/<namespace>/<index_image_name>:<tag> \
  [--binary-image <registry_base_image>]
```

- 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一覧。
- 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>
```

4.8.3. インデックスイメージからのカタログの作成

インデックスイメージから Operator カタログを作成し、これを Operator Lifecycle Manager (OLM) で使用するために OpenShift Container Platform クラスターに適用できます。

前提条件

- レジストリーにビルドされ、プッシュされるインデックスイメージ。

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 ❶
spec:
  sourceType: grpc
  image: <registry>:<port>/<namespace>/redhat-operator-index:v4.8 ❷
  displayName: My Operator Catalog
  publisher: <publisher_name> ❸
  updateStrategy:
    registryPoll: ❹
    interval: 30m
```

- ❶ 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。
- ❷ インデックスイメージを指定します。
- ❸ カタログを公開する名前または組織名を指定します。
- ❹ 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
```

OpenShift Container Platform Web コンソールで、**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

- インデックスイメージがプライベートレジストリーでホストされ、認証が必要な場合は、[プライベートレジストリーからの Operator のイメージへのアクセス](#) を参照してください。

4.8.4. インデックスイメージの更新

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 を設定した後に、クラスター管理者はバンドルイメージをインデックスイメージに追加して、クラスターで利用可能な Operator を最新の状態に維持することができます。

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** version 1.12.3+
- podman** version 1.9.3+
- レジストリーにビルドされ、プッシュされるインデックスイメージ。
- 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。

手順

- 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>

4.8 など、以前にプッシュされたイメージタグを指定します。

<updated_tag>

4.8.1 など、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。

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:4.8 \
  --tag mirror.example.com/abc/abc-redhat-operator-index:4.8.1 \
  --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
```

4.8.5. インデックスイメージのプルーニング

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** version 1.9.3+
- **grpcurl**(サードパーティーのコマンドラインツール)
- **opm** バージョン 1.18.0+
- **Docker v2-2** をサポートするレジストリーへのアクセス



重要

OpenShift Container Platform クラスターの内部レジストリーはターゲットレジストリーとして使用できません。これは、ミラーリングプロセスで必要となるタグを使わないプッシュをサポートしないためです。

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一覧を判別します。

- a. コンテナ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:v4.8
```

出力例

```
Trying to pull registry.redhat.io/redhat/redhat-operator-index:v4.8...
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一覧を取得します。

```
$ 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一覧から特定します。以下に例を示します。

パッケージ一覧のスニペットの例

```
...
{
```

```

    "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:v4.8 \ ❶
  -p advanced-cluster-management,jaeger-product,quay-operator \ ❷
  [-i registry.redhat.io/openshift4/ose-operator-registry:v4.8] \ ❸
  -t <target_registry>:<port>/<namespace>/redhat-operator-index:v4.8 ❹

```

- ❶ プルーニングするインデックス。
- ❷ 保持するパッケージのコンマ区切りリスト。
- ❸ IBM Power Systems および IBM Z イメージのみに必要です: ターゲット OpenShift Container Platform クラスターのメジャーバージョンおよびマイナーバージョンに一致するタグを使用する Operator レジストリーのベースイメージです。
- ❹ 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:v4.8

```

ここで、**<namespace>** はレジストリー上の既存の namespace になります。

4.8.6. プライベートレジストリーからの Operator のイメージへのアクセス

Operator Lifecycle Manager (OLM) によって管理される Operator に関連する特定のイメージが、認証コンテナイメージレジストリー (別名プライベートレジストリー) でホストされる場合、OLM および OperatorHub はデフォルトではイメージをプルできません。アクセスを有効にするために、レジストリーの認証情報が含まれるプルシークレットを作成できます。カタログソースの1つ以上のプルシークレットを参照することで、OLM はシークレットの配置を Operator およびカタログ namespace で処理し、インストールを可能にします。

Operator またはそのオペランドに必要な他のイメージでも、プライベートレジストリーへのアクセスが必要になる場合があります。OLM は、このシナリオのターゲットテナント namespace ではシークレットの配置を処理しませんが、認証情報をグローバルクラスタープルシークレットまたは個別の

namespace サービスアカウントに追加して、必要なアクセスを有効にできます。

OLM によって管理される Operator に適切なプルアクセスがあるかどうかを判別する際に、以下のタイプのイメージを考慮する必要があります。

インデックスイメージ

CatalogSource オブジェクトは、インデックスイメージを参照できます。このイメージは、Operator のバンドル形式を使用し、イメージレジストリーでホストされるコンテナイメージとしてパッケージ化されるカタログソースです。インデックスイメージがプライベートレジストリーでホストされる場合、シークレットを使用してプルアクセスを有効にすることができます。

バンドルイメージ

Operator バンドルイメージは、Operator の一意のバージョンを表すコンテナイメージとしてパッケージ化されるメタデータおよびマニフェストです。カタログソースで参照されるバンドルイメージが1つ以上のプライベートレジストリーでホストされる場合、シークレットを使用してプルアクセスを有効にすることができます。

Operator イメージおよびオペランドイメージ

カタログソースからインストールされた Operator が、(Operator イメージ自体に、または監視するオペランドイメージの1つに) プライベートイメージを使用する場合、デプロイメントは必要なレジストリー認証にアクセスできないため、Operator はインストールに失敗します。カタログソースのシークレットを参照することで、OLM はオペランドがインストールされているターゲットテナント namespace にシークレットを配置することはできません。

代わりに、認証情報を **openshift-config** namespace のグローバルクラスタープルシークレットに追加できます。これにより、クラスターのすべての namespace へのアクセスが提供されます。または、クラスター全体へのアクセスの提供が許容されない場合、プルシークレットをターゲットテナント namespace の **default** のサービスアカウントに追加できます。

前提条件

- プライベートレジストリーで、以下のうち少なくとも1つがホストされます。
 - インデックスイメージまたはカタログイメージ。
 - Operator のバンドルイメージ
 - Operator またはオペランドのイメージ。

手順

1. 必要な各プライベートレジストリーのシークレットを作成します。
 - a. プライベートレジストリーにログインして、レジストリー認証情報ファイルを作成または更新します。

```
$ podman login <registry>:<port>
```



注記

レジストリー認証情報のファイルパスは、レジストリーへのログインに使用されるコンテナツールによって異なります。**podman** CLI の場合、デフォルトの場所は **\${XDG_RUNTIME_DIR}/containers/auth.json** です。**docker** CLI の場合、デフォルトの場所は **/root/.docker/config.json** です。

- b. シークレットごとに1つのレジストリーに対してのみ認証情報を追加し、別のシークレットで複数のレジストリーの認証情報を管理することが推奨されます。これ以降の手順で、複数のシークレットを **CatalogSource** オブジェクトに含めることができ、OpenShift Container Platform はイメージのプル時に使用する単一の仮想認証情報ファイルにシークレットをマージします。
- レジストリー認証情報ファイルは、デフォルトで複数のレジストリーの詳細を保存することができます。ファイルの現在の内容を確認します。以下に例を示します。

2つのレジストリーの認証情報を保存するファイル

```
{
  "auths": {
    "registry.redhat.io": {
      "auth": "FrNHNYdQXdzclNqdg=="
    },
    "quay.io": {
      "auth": "Xd2lhdsbnRib21iMQ=="
    }
  }
}
```

これ以降の手順で、シークレットの作成にこのファイルが使用されるため、保存できる詳細は1つのファイルにつき1つのレジストリーのみであることを確認してください。これには、以下の方法の1つを使用します。

- **podman logout <registry>** コマンドを使用して、必要な1つのレジストリーのみになるまで、追加のレジストリーの認証情報を削除します。
- レジストリー認証情報ファイルを編集し、レジストリーの詳細を分離して、複数のファイルに保存します。以下に例を示します。

1つのレジストリーの認証情報を保存するファイル

```
{
  "auths": {
    "registry.redhat.io": {
      "auth": "FrNHNYdQXdzclNqdg=="
    }
  }
}
```

別のレジストリーの認証情報を保存するファイル

```
{
  "auths": {
    "quay.io": {
      "auth": "Xd2lhdsbnRib21iMQ=="
    }
  }
}
```

- c. プライベートレジストリーの認証情報が含まれるシークレットを **openshift-marketplace** namespace に作成します。

```
$ oc create secret generic <secret_name> \
  -n openshift-marketplace \
  --from-file=.dockerconfigjson=<path/to/registry/credentials> \
  --type=kubernetes.io/dockerconfigjson
```

この手順を繰り返して、他の必要なプライベートレジストリーの追加シークレットを作成し、**--from-file** フラグを更新して別のレジストリー認証情報ファイルのパスを指定します。

2. 1つ以上のシークレットを参照するように既存の **CatalogSource** オブジェクトを作成または更新します。

```
apiVersion: operators.coreos.com/v1alpha1
kind: CatalogSource
metadata:
  name: my-operator-catalog
  namespace: openshift-marketplace
spec:
  sourceType: grpc
  secrets: ❶
  - "<secret_name_1>"
  - "<secret_name_2>"
  image: <registry>:<port>/<namespace>/<image>:<tag>
  displayName: My Operator Catalog
  publisher: <publisher_name>
  updateStrategy:
    registryPoll:
      interval: 30m
```

- ❶ **spec.secrets** セクションを追加し、必要なシークレットを指定します。

3. サブスクリブされた Operator によって参照される Operator イメージまたはオペランドイメージにプライベートレジストリーへのアクセスが必要な場合は、クラスター内のすべての namespace または個々のターゲットテナント namespace のいずれかにアクセスを提供できます。
 - クラスター内のすべての namespace へアクセスを提供するには、認証情報を **openshift-config** namespace のグローバルクラスタープルシークレットに追加します。



警告

クラスターリソースは新規のグローバルプルシークレットに合わせて調整する必要がありますが、これにより、クラスターのユーザービリティが一時的に制限される可能性があります。

- a. グローバルプルシークレットから **.dockerconfigjson** ファイルを展開します。

```
$ oc extract secret/pull-secret -n openshift-config --confirm
```


- b. **.dockerconfigjson** ファイルを、必要なプライベートレジストリーまたはレジストリーの認証情報で更新し、これを新規ファイルとして保存します。

```
$ cat .dockerconfigjson | \
jq --compact-output '.auths["<registry>:<port>/<namespace>/" ] |= . + {"auth":'
<token>"}' ❶
> new_dockerconfigjson
```

- ❶ **<registry>:<port>/<namespace>** をプライベートレジストリーの詳細に置き換え、**<token>** を認証情報に置き換えます。

- c. 新規ファイルでグローバルプルシークレットを更新します。

```
$ oc set data secret/pull-secret -n openshift-config \
--from-file=.dockerconfigjson=new_dockerconfigjson
```

- 個別の namespace を更新するには、ターゲットテナント namespace でアクセスが必要な Operator のサービスアカウントにプルシークレットを追加します。

- a. テナント namespace で **openshift-marketplace** 用に作成したシークレットを再作成します。

```
$ oc create secret generic <secret_name> \
-n <tenant_namespace> \
--from-file=.dockerconfigjson=<path/to/registry/credentials> \
--type=kubernetes.io/dockerconfigjson
```

- b. テナント namespace を検索して、Operator のサービスアカウントの名前を確認します。

```
$ oc get sa -n <tenant_namespace> ❶
```

- ❶ Operator が個別の namespace にインストールされていた場合、その namespace を検索します。Operator がすべての namespace にインストールされていた場合、**openshift-operators** namespace を検索します。

出力例

```
NAME          SECRETS  AGE
builder       2        6m1s
default       2        6m1s
deployer      2        6m1s
etcd-operator 2        5m18s ❶
```

- ❶ インストールされた etcd Operator のサービスアカウント。

- c. シークレットを Operator のサービスアカウントにリンクします。

```
$ oc secrets link <operator_sa> \
-n <tenant_namespace> \
<secret_name> \
```

```
--for=pull
```

関連情報

- レジストリーの認証情報に使用されるものを含むシークレットの種類に関する詳細は、[シークレットの概要](#) を参照してください。
- このシークレットの変更が与える影響についての詳細は、[グローバルクラスタープルシークレットの更新](#) を参照してください。
- namespace ごとにプルシークレットをサービスアカウントにリンクする方法に関する詳細は、[Pod が他のセキュリティ保護されたレジストリーからイメージを参照できるようにする設定](#) を参照してください。

4.8.7. デフォルトの OperatorHub ソースの無効化

Red Hat によって提供されるコンテンツを調達する Operator カタログおよびコミュニティプロジェクトは、OpenShift Container Platform のインストール時にデフォルトで OperatorHub に設定されます。クラスター管理者は、デフォルトカタログのセットを無効にすることができます。

手順

- **disableAllDefaultSources: true** を **OperatorHub** オブジェクトに追加して、デフォルトカタログのソースを無効にします。

```
$ oc patch OperatorHub cluster --type json \
  -p '[{"op": "add", "path": "/spec/disableAllDefaultSources", "value": true}]'
```


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。**Administration** → **Cluster Settings** → **Global Configuration** → **OperatorHub** ページから、**Sources** タブをクリックして、個別のソースを作成し、削除し、無効にし、有効にすることができます。

4.8.8. カスタムカタログの削除

クラスター管理者は、関連するカタログソースを削除して、以前にクラスターに追加されたカスタム Operator カタログを削除できます。

手順

1. Web コンソールの **Administrator** パースペクティブで、**Administration** → **Cluster Settings** に移動します。
2. **Global Configuration** タブをクリックしてから、**OperatorHub** をクリックします。
3. **Sources** タブをクリックします。
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。

4.9. ネットワークが制限された環境での OPERATOR LIFECYCLE MANAGER の使用

ネットワークが制限された環境 (非接続クラスターとしても知られる) にインストールされている OpenShift Container Platform クラスターの場合、デフォルトで Operator Lifecycle Manager (OLM) はリモートレジストリーでホストされる Red Hat が提供する OperatorHub ソースにアクセスできません。それらのリモートソースには完全なインターネット接続が必要であるためです。

ただし、クラスター管理者は、完全なインターネットアクセスのあるワークステーションがある場合には、クラスターがネットワークが制限された環境で OLM を使用できるようにできます。ワークステーションは、リモートソースのローカルミラーを準備するために使用され、コンテンツをミラーレジストリーにプッシュしますが、これにはリモートの OperatorHub コンテンツをプルするのに完全なインターネットアクセスが必要になります。

ミラーレジストリーは bastion ホストに配置することができます。bastion ホストには、ワークステーションと非接続クラスターの両方への接続、または完全に切断されたクラスター、またはミラーリングされたコンテンツを非接続環境に物理的に移動するためにリムーバブルメディアが必要な エアギャップ ホストへの接続が必要です。

以下では、ネットワークが制限された環境で OLM を有効にするために必要な以下のプロセスについて説明します。

- OLM のデフォルトのリモート OperatorHub ソースを無効にします。
- 完全なインターネットアクセスのあるワークステーションを使用して、OperatorHub コンテンツのローカルミラーを作成し、これをミラーレジストリーにプッシュします。
- OLM を、デフォルトのリモートソースからではなくミラーレジストリーのローカルソースから Operator をインストールし、管理するように設定します。

ネットワークが制限された環境で OLM を有効にした後も、引き続き制限のないワークステーションを使用して、Operator の新しいバージョンが更新されるとローカルの OperatorHub ソースを更新された状態に維持することができます。

重要

OLM はローカルソースから Operator を管理できますが、指定された Operator がネットワークが制限された環境で正常に実行されるかどうかは Operator 自体に依存します。以下は、Operator の特長です。

- 関連するイメージ、または 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他のコンテナイメージを **ClusterServiceVersion (CSV)** オブジェクトの **relatedImages** パラメーターで一覧表示します。
- 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。

Infrastructure Features で **Disconnected** フィルターを選択することにより、切断モードでの実行をサポートする Red Hat Operator のリストを Red Hat Ecosystem Catalog で検索できます。

<https://catalog.redhat.com/software/operators/search>

関連情報

- [Red Hat が提供する Operator カタログ](#)

- ネットワークが制限された環境についての Operator の有効化

4.9.1. 前提条件

- **cluster-admin** 権限を持つユーザーとして OpenShift Container Platform クラスターにログインします。
- デフォルトカタログをプルーニングし、Operator のサブセットのみを選択的にミラーリングするには、[opm CLI](#) をインストールします。



注記

IBM Z のネットワークが制限された環境で OLM を使用している場合は、レジストリーを配置するディレクトリーに 12 GB 以上を割り当てる必要があります。

4.9.2. デフォルトの OperatorHub ソースの無効化

Red Hat によって提供されるコンテンツを調達する Operator カタログおよびコミュニティープロジェクトは、OpenShift Container Platform のインストール時にデフォルトで OperatorHub に設定されます。ネットワークが制限された環境では、クラスター管理者としてデフォルトのカタログを無効にする必要があります。その後、OperatorHub をローカルカタログソースを使用するように設定できます。

手順

- **disableAllDefaultSources: true** を **OperatorHub** オブジェクトに追加して、デフォルトカタログのソースを無効にします。

```
$ oc patch OperatorHub cluster --type json \
  -p '[{"op": "add", "path": "/spec/disableAllDefaultSources", "value": true}]'
```

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。**Administration** → **Cluster Settings** → **Global Configuration** → **OperatorHub** ページから、**Sources** タブをクリックして、個別のソースを作成し、削除し、無効にし、有効にすることができます。

4.9.3. インデックスイメージのプルーニング

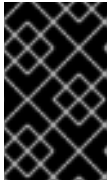
Operator Bundle Format に基づくインデックスイメージは、Operator カタログのコンテナ化されたスナップショットです。パッケージの指定された一覧以外のすべてのインデックスをプルーニングできます。これにより、必要な Operator のみが含まれるソースインデックスのコピーを作成できます。

ネットワークが制限された環境の OpenShift Container Platform クラスターでミラーリングされたコンテンツを使用するように Operator Lifecycle Manager (OLM) を設定する場合、デフォルトカタログから Operator のサブセットのみをミラーリングする必要がある場合に、このプルーニング方法を使用します。

この手順のステップでは、ターゲットレジストリーは、ネットワークアクセスが無制限のワークステーションからアクセスできる既存のミラーレジストリーです。この例では、デフォルトの **redhat-operators** カタログのインデックスイメージのプルーニングも示していますが、このプロセスはすべてのインデックスイメージに対して同じです。

前提条件

- ネットワークアクセスが無制限のワークステーション
- **podman** version 1.9.3+
- **grpcurl**(サードパーティーのコマンドラインツール)
- **opm** バージョン 1.18.0+
- **Docker v2-2** をサポートするレジストリーへのアクセス



重要

OpenShift Container Platform クラスターの内部レジストリーはターゲットレジストリーとして使用できません。これは、ミラーリングプロセスで必要となるタグを使わないプッシュをサポートしないためです。

手順

1. **registry.redhat.io** で認証します。

```
$ podman login registry.redhat.io
```

2. ターゲットレジストリーで認証します。

```
$ podman login <target_registry>
```

3. 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一覧を判別します。

- a. コンテナ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:v4.8
```

出力例

```
Trying to pull registry.redhat.io/redhat/redhat-operator-index:v4.8...
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一覧を取得します。

```
$ 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一覧から特定します。以下に例を示します。

パッケージ一覧のスニペットの例

```
...
```

```
{
  "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** を押してコンテナプロセスを停止します。
4. 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:v4.8 \1
  -p advanced-cluster-management,jaeger-product,quay-operator \2
  [-i registry.redhat.io/openshift4/ose-operator-registry:v4.8] \3
  -t <target_registry>:<port>/<namespace>/redhat-operator-index:v4.8 \4
```

- ❶ プルーニングするインデックス。
 - ❷ 保持するパッケージのコンマ区切りリスト。
 - ❸ IBM Power Systems および IBM Z イメージのみに必要です: ターゲット OpenShift Container Platform クラスターのメジャーバージョンおよびマイナーバージョンに一致するタグを使用する Operator レジストリーのベースイメージです。
 - ❹ 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。
5. 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:v4.8
```

ここで、**<namespace>** はレジストリー上の既存の namespace になります。たとえば、**olm-mirror** namespace を作成し、ミラーリングされたすべてのコンテンツをプッシュすることができます。

4.9.4. Operator カタログのミラーリング

oc adm catalog mirror コマンドを使用して、Red Hat が提供するカタログまたはカスタムカタログの Operator コンテンツをコンテナイメージレジストリーにミラーリングできます。ターゲットレジストリーは [Docker v2-2](#) をサポートする必要があります。ネットワークが制限された環境のクラスターの場合、このレジストリーには、ネットワークが制限されたクラスターのインストール時に作成されたミラーレジストリーなど、クラスターにネットワークアクセスのあるレジストリーを使用できます。



重要

OpenShift Container Platform クラスターの内部レジストリーはターゲットレジストリーとして使用できません。これは、ミラーリングプロセスで必要となるタグを使わないプッシュをサポートしないためです。

oc adm catalog mirror コマンドは、Red Hat が提供するインデックスイメージであるか、または独自のカスタムビルドされたインデックスイメージであるかに関係なく、ミラーリングプロセス中に指定されるインデックスイメージをターゲットレジストリーに自動的にミラーリングします。次に、ミラーリングされたインデックスイメージを使用して、Operator Lifecycle Manager (OLM) がミラーリングされたカタログを OpenShift Container Platform クラスターにロードできるようにするカタログソースを作成できます。

前提条件

- ネットワークアクセスが無制限のワークステーション
- **podman** バージョン 1.9.3 以降。
- [Docker v2-2](#) をサポートするミラーレジストリーへのアクセス。
- ミラーリングされた Operator コンテンツを保存するために使用するミラーレジストリー上の namespace を決定します。たとえば、**olm-mirror** namespace を作成できます。
- ミラーレジストリーにインターネットアクセスがない場合は、ネットワークアクセスが無制限のワークステーションにリムーバブルメディアを接続します。
- **registry.redhat.io** などのプライベートレジストリーを使用している場合、後続の手順で使用するために **REG_CREDS** 環境変数をレジストリー認証情報のファイルパスに設定します。たとえば **podman** CLI の場合は、以下のようになります。

```
$ REG_CREDS=${XDG_RUNTIME_DIR}/containers/auth.json
```

手順

1. Red Hat が提供するカタログをミラーリングする場合は、ネットワークアクセスが無制限のワークステーションで以下のコマンドを実行し、**registry.redhat.io** で認証します。

```
$ podman login registry.redhat.io
```

2. **oc adm catalog mirror** コマンドは、インデックスイメージのコンテンツを抽出し、ミラーリングに必要なマニフェストを生成します。コマンドのデフォルト動作で、マニフェストを生成し、インデックスイメージからのすべてのイメージコンテンツを、インデックスイメージと同様にミラーレジストリーに対して自動的にミラーリングします。または、ミラーレジストリーが完全に非接続または **エアギャップ** 環境のホスト上にある場合、最初にコンテンツをリムーバブルメディアにミラーリングし、メディアを非接続環境に移行してから、メディアからレジストリーにコンテンツをレジストリーに対してミラーリングできます。

- **オプション A: ミラーレジストリーがネットワークアクセスが無制限のワークステーションと同じネットワーク上にある** 場合、ワークステーションで以下のアクションを実行します。
 - a. ミラーレジストリーに認証が必要な場合は、以下のコマンドを実行してレジストリーにログインします。


```
$ podman login <mirror_registry>
```

- b. 以下のコマンドを実行してコンテンツをミラーリングします。

```
$ oc adm catalog mirror \
  <index_image> \1
  <mirror_registry>:<port>/<namespace> \2
  [-a ${REG_CREDS}] \3
  [--insecure] \4
  [--index-filter-by-os='<platform>/<arch>'] \5
  [--manifests-only] \6
```

- 1 ミラーリングするカタログのインデックスイメージを指定します。たとえば、これは以前に作成したプルーニングされたインデックスイメージ、または **registry.redhat.io/redhat/redhat-operator-index:v4.8** などのデフォルトカタログのソースインデックスイメージのいずれかである可能性があります。
- 2 Operator コンテンツをミラーリングするターゲットレジストリーおよび namespace の完全修飾ドメイン名 (FQDN) を指定します。ここで、**<namespace>** はレジストリーの既存の namespace です。たとえば、**olm-mirror** namespace を作成し、ミラーリングされたすべてのコンテンツをプッシュすることができます。
- 3 オプション: 必要な場合は、レジストリー認証情報ファイルの場所を指定します。**registry.redhat.io** には、**{REG_CREDS}** が必要です。
- 4 オプション: ターゲットレジストリーの信頼を設定しない場合は、**--insecure** フラグを追加します。
- 5 オプション: 複数のバリエーションが利用可能な場合に、選択するインデックスイメージのプラットフォームおよびアーキテクチャーを指定します。イメージは **'<platform>/<arch>/<variant>'** として渡されます。これはインデックスで参照されるイメージには適用されません。使用できる値は、**linux/amd64**、**linux/ppc64le**、**linux/s390x** および ***** です。
- 6 オプション: ミラーリングに必要なマニフェストのみを生成し、実際にはイメージコンテンツをレジストリーにミラーリングしません。このオプションは、ミラーリングする内容を確認するのに役立ちます。また、パッケージのサブセットのみが必要な場合に、マッピングの一覧に変更を加えることができます。次に、**mapping.txt** ファイルを **oc image mirror** コマンドで使用し、後のステップでイメージの変更済みの一覧をミラーリングできます。このフラグは、カタログからのコンテンツの高度で選択可能なミラーリングにのみ使用することが意図されています。**opm index prune** をインデックスイメージをプルーニングするために以前にしている場合、これはほとんどのカタログ管理のユースケースに適しています。

出力例

```
src image has index label for database path: /database/index.db
using database path mapping: /database/index.db:/tmp/153048078
wrote database to /tmp/153048078 \1
...
wrote mirroring manifests to manifests-redhat-operator-index-1614211642 \2
```


- 1 コマンドで生成された一時的な **index.db** データベースのディレクトリー。
- 2 生成される manifests ディレクトリー名を記録します。このディレクトリー名は、後の手順で使用されます。



注記

Red Hat Quay では、ネストされたリポジトリはサポート対象外です。その結果、**oc adm catalog mirror** コマンドを実行すると、**401 unauthorized** エラーで失敗します。回避策として、**oc adm catalog mirror** コマンドを実行するときに **--max-components = 2** オプションを使用して、ネストされたリポジトリの作成を無効にすることができます。この回避策の詳細は、[Unauthorized error thrown while using catalog mirror command with Quay registry](#) のナレッジソリューション記事を参照してください。

- オプション B: ミラーレジストリーが非接続ホストにある場合は、以下のアクションを実行します。
 - a. ネットワークアクセスが無制限のワークステーションで以下のコマンドを実行し、コンテンツをローカルファイルにミラーリングします。

```
$ oc adm catalog mirror \
  <index_image> \1
  file:///local/index \2
  [-a ${REG_CREDS}] \
  [--insecure] \
  [--index-filter-by-os='<platform>/<arch>']
```

- 1 ミラーリングするカタログのインデックスイメージを指定します。たとえば、これは以前に作成したプルーニングされたインデックスイメージ、または **registry.redhat.io/redhat/redhat-operator-index:v4.8** などのデフォルトカタログのソースインデックスイメージのいずれかである可能性があります。
- 2 現在のディレクトリーのローカルファイルにコンテンツをミラーリングします。

出力例

```
...
info: Mirroring completed in 5.93s (5.915MB/s)
wrote mirroring manifests to manifests-my-index-1614985528 1

To upload local images to a registry, run:

oc adm catalog mirror file:///local/index/myrepo/my-index:v1
REGISTRY/REPOSITORY 2
```

- 1 生成される manifests ディレクトリー名を記録します。このディレクトリー名は、後の手順で使用されます。
- 2 提供されたインデックスイメージをベースとする、拡張された **file://** パスを記録します。このパスは、後のステップで使用されます。

- b. 現在のディレクトリーに生成される **v2/** ディレクトリーをリムーバブルメディアにコピーします。
- c. メディアを物理的に削除して、これをミラーレジストリーにアクセスできる非接続環境のホストに割り当てます。
- d. ミラーレジストリーに認証が必要な場合は、非接続環境のホストで以下のコマンドを実行し、レジストリーにログインします。

```
$ podman login <mirror_registry>
```

- e. **v2/** ディレクトリーを含む親ディレクトリーから以下のコマンドを実行し、ローカルファイルからミラーレジストリーにイメージをアップロードします。

```
$ oc adm catalog mirror \
  file://local/index/<repo>/<index_image>:<tag> \ ❶
  <mirror_registry>:<port>/<namespace> \ ❷
  [-a ${REG_CREDS}] \
  [--insecure] \
  [--index-filter-by-os='<platform>/<arch>']
```

❶ 直前のコマンド出力の **file://** パスを指定します。

❷ Operator コンテンツをミラーリングするターゲットレジストリーおよび namespace の完全修飾ドメイン名 (FQDN) を指定します。ここで、**<namespace>** はレジストリーの既存の namespace です。たとえば、**olm-mirror** namespace を作成し、ミラーリングされたすべてのコンテンツをプッシュすることができます。



注記

Red Hat Quay では、ネストされたリポジトリはサポート対象外です。その結果、**oc adm catalog mirror** コマンドを実行すると、**401 unauthorized** エラーで失敗します。回避策として、**oc adm catalog mirror** コマンドを実行するときに **--max-components = 2** オプションを使用して、ネストされたリポジトリの作成を無効にすることができます。この回避策の詳細は、[Unauthorized error thrown while using catalog mirror command with Quay registry](#) のナレッジソリューション記事を参照してください。

- f. **oc adm catalogmirror** コマンドを再度実行します。新しくミラーリングされたインデックスイメージをソースとして使用し、前の手順で使用したのと同じミラーレジストリーの namespace をターゲットとして使用します。

```
$ oc adm catalog mirror \
  <mirror_registry>:<port>/<index_image> \
  <mirror_registry>:<port>/<namespace> \
  --manifests-only \ ❶
  [-a ${REG_CREDS}] \
  [--insecure]
```

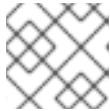
❶ コマンドがミラーリングされたすべてのコンテンツを再度コピーしないように、このステップには **--manifests-only** フラグが必要です。



重要

前のステップで生成された **imageContentSourcePolicy.yaml** ファイルのイメージマッピングをローカルパスから有効なミラー位置に更新する必要があるため、このステップが必要です。そうしないと、後のステップで **imageContentSourcePolicy** オブジェクトを作成するときにエラーが発生します。

3. コンテンツをレジストリーにミラーリングした後に、現在のディレクトリーに生成される manifests ディレクトリーを検査します。



注記

manifests ディレクトリー名は後の手順で使用されます。

直前の手順で同じネットワークのレジストリーにコンテンツをミラーリングする場合、ディレクトリー名は以下の形式になります。

```
manifests-<index_image_name>-<random_number>
```

直前の手順で非接続ホストのレジストリーにコンテンツをミラーリングする場合、ディレクトリー名は以下の形式になります。

```
manifests-index/<namespace>/<index_image_name>-<random_number>
```

manifests ディレクトリーには以下のファイルが含まれており、これらの一部にはさらに変更が必要になる場合があります。

- **catalogSource.yaml** ファイルは、インデックスイメージタグおよび他の関連するメタデータで事前に設定される **CatalogSource** オブジェクトの基本的な定義です。このファイルは、カタログソースをクラスターに追加するためにそのまま使用したり、変更したりできます。



重要

ローカルファイルにコンテンツをミラーリングする場合は、**catalogSource.yaml** ファイルを変更して **metadata.name** フィールドからバックスラッシュ (/) 文字を削除する必要があります。または、オブジェクトの作成を試みると、invalid resource name (無効なリソース名) を示すエラーを出して失敗します。

- これにより、**imageContentSourcePolicy.yaml** ファイルは **ImageContentSourcePolicy** オブジェクトを定義します。このオブジェクトは、ノードを Operator マニフェストおよびミラーリングされたレジストリーに保存されるイメージ参照間に変換できるように設定します。



注記

クラスターが **ImageContentSourcePolicy** オブジェクトを使用してリポジトリのミラーリングを設定する場合、ミラーリングされたレジストリーにグローバルプルシークレットのみを使用できます。プロジェクトにプルシークレットを追加することはできません。

- **mapping.txt** ファイルには、すべてのソースイメージが含まれ、これはそれらのイメージをターゲットレジストリー内のどこにマップするかを示します。このファイルは **oc image mirror** コマンドと互換性があり、ミラーリング設定をさらにカスタマイズするために使用できます。



重要

ミラーリングのプロセスで **--manifests-only** フラグを使用しており、ミラーリングするパッケージのサブセットをさらにトリミングするには、**mapping.txt** ファイルの変更および **oc image mirror** コマンドでのファイルの使用について、OpenShift Container Platform 4.7 ドキュメントの [Package Manifest Format カタログイメージのミラーリング](#) の手順を参照してください。これらの追加のアクションを実行した後に、この手順を続行できます。

4. 非接続クラスターへのアクセスのあるホストで、以下のコマンドを実行して manifests ディレクトリーで **imageContentSourcePolicy.yaml** ファイルを指定し、**ImageContentSourcePolicy** (ICSP) オブジェクトを作成します。

```
$ oc create -f <path/to/manifests/dir>/imageContentSourcePolicy.yaml
```

ここで、**<path/to/manifests/dir>** は、ミラーリングされたコンテンツについての manifests ディレクトリーへのパスです。

ミラーリングされたインデックスイメージおよび Operator コンテンツを参照する **CatalogSource** を作成できるようになりました。

関連情報

- [非接続インストールのイメージのミラーリング](#)
- [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](#)

4.9.5. インデックスイメージからのカタログの作成

インデックスイメージから Operator カタログを作成し、これを Operator Lifecycle Manager (OLM) で使用するために OpenShift Container Platform クラスターに適用できます。

前提条件

- レジストリーにビルドされ、プッシュされるインデックスイメージ。

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。 **oc adm catalog mirror** コマンドを使用してカタログをターゲットレジストリーにミラーリングする場合、開始点として生成される **catalogSource.yaml** ファイルをそのまま使用することができます。
 - 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 ❶
namespace: openshift-marketplace ❷
spec:
  sourceType: grpc
  image: <registry>:<port>/<namespace>/redhat-operator-index:v4.8 ❸
  displayName: My Operator Catalog
  publisher: <publisher_name> ❹
  updateStrategy:
    registryPoll: ❺
    interval: 30m

```

- ❶ レジストリーにアップロードする前にローカルファイルにコンテンツをミラーリングする場合は、**metadata.name** フィールドからバックスラッシュ (/) 文字を削除し、オブジェクトの作成時に invalid resource name エラーを回避します。
- ❷ 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。
- ❸ インデックスイメージを指定します。
- ❹ カタログを公開する名前または組織名を指定します。
- ❺ 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-6njsx6          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
```

OpenShift Container Platform Web コンソールで、**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

- インデックスイメージがプライベートレジストリーでホストされ、認証が必要な場合は、[プライベートレジストリーからの Operator のイメージへのアクセス](#) を参照してください。

4.9.6. インデックスイメージの更新

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、クラスター管理者はバンドルイメージをインデックスイメージに追加して、クラスターで利用可能な Operator を最新の状態に維持することができます。

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。ネットワークが制限された環境の場合、更新されたコンテンツもクラスターにミラーリングする必要があります。

前提条件

- opm** version 1.12.3+
- podman** version 1.9.3+
- レジストリーにビルドされ、プッシュされるインデックスイメージ。
- 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。

手順

- 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> \
  --from-index <registry>/<namespace>/<existing_index_image>:<existing_tag> \
  --tag <registry>/<namespace>/<existing_index_image>:<updated_tag> \
  --pull-tool podman
```

- 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。
- 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。
- 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>

4.8 など、以前にプッシュされたイメージタグを指定します。

<updated_tag>

4.8.1 など、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。

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:4.8 \
  --tag mirror.example.com/abc/abc-redhat-operator-index:4.8.1 \
  --pull-tool podman
```

- 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>
```

- Operator カタログのミラーリング**の手順にあるステップを再度実行し、更新されたコンテンツをミラーリングします。ただし、**ImageContentSourcePolicy** (ICSP) オブジェクトの作成手順を参照する場合、**oc create** コマンドの代わりに **oc replace** コマンドを使用します。以下に例を示します。

```
$ oc replace -f ./manifests-redhat-operator-index-
<random_number>/imageContentSourcePolicy.yaml
```

この変更は、オブジェクトがすでに存在し、更新する必要があるために必要になります。



注記

通常、**oc apply** コマンドを使用して、**oc apply** を使用して以前に作成された既存のオブジェクトを更新できます。ただし、ICSP オブジェクトの **metadata.annotations** フィールドのサイズに関する既知の問題により、現時点では **oc replace** コマンドをこの手順で使用する必要があります。

4. 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
```

関連情報

- [Operator カatalogのミラーリング](#)

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
- Prometheus Operator がデプロイされているクラスターでできるように、生成された Go ベースの Operator にメトリックが自動的にセットアップします。

Kubernetes ベースのクラスター (OpenShift Container Platform など) へのクラスター管理者のアクセスのある Operator の作成者は、Operator SDK CLI を使用して Go、Ansible、または Helm をベースに独自の 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 Container Platform 4.8 は Operator SDK v1.8.0 以降をサポートします。

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。

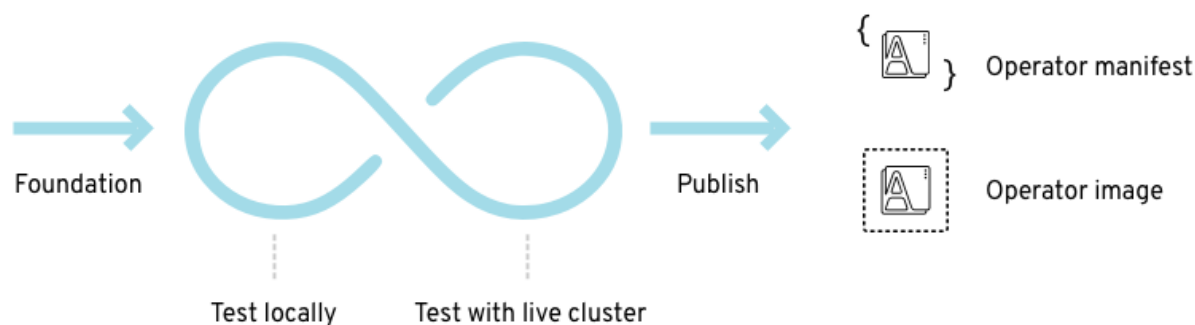
5.1.2. 開発ワークフロー

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

Operator SDK *Build, test, iterate*



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。



注記

OpenShift Container Platform 4.8 は Operator SDK v1.8.0 をサポートします。

5.2.1. 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](#) v1.16+
- **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. **4.8.4** 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-v1.8.0-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. 展開された **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: "v1.8.0-ocp", ...
```

5.3. 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 Container Platform 4.8 は Operator SDK v1.8.0 をサポートします。ワークステーションに v1.3.0 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 v1.8.0 にアップグレードできます。

ただし、既存の Operator プロジェクトが Operator SDK v1.8.0 との互換性を維持するには、v1.3.0 以降に導入された関連する重大な変更に対し、アップグレード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 v1.3.0 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。

5.3.1. Operator SDK v1.8.0 のプロジェクトのアップグレード

v1.8.0 との互換性を確保して既存の Operator プロジェクトをアップグレードするには、以下のアップグレード手順を実行する必要があります。

前提条件

- Operator SDK v1.8.0 がインストールされている
- 以前に Operator SDK v1.3.0 で作成または維持された Operator プロジェクト

手順

1. **PROJECT** ファイルに対して以下の変更を実行します。

- a. **PROJECT** ファイルの **plugins** オブジェクトを、**manifests** と **scorecard** を使用するよう
に更新します。

Operator Lifecycle Manager (OLM) およびスコアカードマニフェストを作成する **manifests** および **scorecard** プラグインには、関連ファイルを作成する **create** サブコマンドを実行するためのプラグインプラグインが含まれるようになりました。

- Go ベースの Operator プロジェクトでは、既存の Go ベースのプラグイン設定オブジェクトがすでに存在します。以前の設定は引き続きサポートされますが、設定オプションがそれぞれのプラグインに追加されるため、これらの新しいオブジェクトは今後役に立ちます。

以前の設定

```
version: 3-alpha
...
plugins:
  go.sdk.operatorframework.io/v2-alpha: {}
```

新規設定

```
version: 3-alpha
...
plugins:
  manifests.sdk.operatorframework.io/v2: {}
  scorecard.sdk.operatorframework.io/v2: {}
```

- オプション: Ansible および Helm ベースの Operator プロジェクトの場合に、プラグイン設定オブジェクトは存在しませんでした。プラグイン設定オブジェクトを追加する必要はありませんが、設定オプションがそれぞれのプラグインに追加されるため、これらの新しいオブジェクトは今後役に立ちます。

```
version: 3-alpha
...
plugins:
  manifests.sdk.operatorframework.io/v2: {}
  scorecard.sdk.operatorframework.io/v2: {}
```

- b. **PROJECT** 設定バージョン **3-alpha** は **3** にアップグレードする必要があります。**PROJECT** ファイルの **version** キーは、**PROJECT** の設定バージョンを表します。

以前の PROJECT ファイル

```
version: 3-alpha
resources:
- crdVersion: v1
...
```

バージョン **3alpha** は **バージョン 3** として安定しており、プロジェクトの完全記述に十分な設定フィールドが含まれています。この変更は、仕様がアルファバージョンであるため、技術的には重大ではありませんが、デフォルトで **operator-sdk** コマンドで使用されていたので、Breaking とマークして、便利なアップグレードパスを設定する必要があります。

- i. **alpha config-3alpha-to-3** コマンドを実行して、**PROJECT** ファイルの多くをバージョン **3-alpha** から **3** に変換します。

```
$ operator-sdk alpha config-3alpha-to-3
```

出力例

```
Your PROJECT config file has been converted from version 3-alpha to 3. Please
make sure all config data is correct.
```

このコマンドは、自動変換が不可能な方向を示すコメントも出力します。

- ii. 変更内容を確認します。

新規の PROJECT ファイル

```
version: "3"
resources:
- api:
  crdVersion: v1
...
```

2. **config/manager/manager.yaml** ファイルに以下の変更を加えます。

- a. Ansible および Helm ベースの Operator プロジェクトの場合は、liveness および readiness プローブを追加します。

Operator SDK でビルドされた新規プロジェクトには、デフォルトでプローブが設定されます。指定のイメージベースで、エンドポイント **/healthz** および **/readyz** が利用可能になりました。**Dockerfile** を更新して最新のベースイメージを使用するように既存のプロジェクトを更新し、以下を **config/manager/manager.yaml** ファイルの **manager** コンテナに追加できます。

例5.1 Ansible ベースの Operator プロジェクトの設定

```
livenessProbe:
  httpGet:
    path: /healthz
    port: 6789
  initialDelaySeconds: 15
  periodSeconds: 20
readinessProbe:
```

```

httpGet:
  path: /readyz
  port: 6789
initialDelaySeconds: 5
periodSeconds: 10

```

例5.2 Helm ベースの Operator プロジェクトの設定

```

livenessProbe:
  httpGet:
    path: /healthz
    port: 8081
  initialDelaySeconds: 15
  periodSeconds: 20
readinessProbe:
  httpGet:
    path: /readyz
    port: 8081
  initialDelaySeconds: 5
  periodSeconds: 10

```

- b. Ansible および Helm ベースの Operator プロジェクトについては、セキュリティーコンテキストをマネージャーのデプロイメントに追加します。

config/manager/manager.yaml ファイルに、以下のセキュリティーコンテキストを追加します。

例5.3 config/manager/manager.yaml ファイル

```

spec:
  ...
  template:
    ...
    spec:
      securityContext:
        runAsNonRoot: true
      containers:
        - name: manager
          securityContext:
            allowPrivilegeEscalation: false

```

3. Makefile に以下の変更を加えます。

- a. Ansible および Helm ベースの Operator プロジェクトの場合には、**Makefile** で **helm-operator** および **ansible-operator** URL を更新します。

- Ansible ベース Operator のプロジェクトの場合:

```

https://github.com/operator-framework/operator-
sdk/releases/download/v1.3.0/ansible-operator-v1.3.0-$(ARCHOPER)-$(OSOPER)

```

以下のように変更します。

```
https://github.com/operator-framework/operator-
sdk/releases/download/v1.8.0/ansible-operator_$(OS)_$(ARCH)
```

- Helm ベースの Operator プロジェクトの場合:

```
https://github.com/operator-framework/operator-sdk/releases/download/v1.3.0/helm-
operator-v1.3.0-$(ARCHOPPER)-$(OSOPER)
```

以下のように変更します。

```
https://github.com/operator-framework/operator-sdk/releases/download/v1.8.0/helm-
operator_$(OS)_$(ARCH)
```

- b. Ansible および Helm ベースの Operator プロジェクトの場合は、**Makefile** で **helm-operator**、**ansible-operator** および **kustomize** ルールを更新します。これらのルールはローカルのバイナリーをダウンロードしますが、グローバルバイナリーが存在する場合はローカルのバイナリーは使用されません。

例5.4 Ansible ベースの Operator プロジェクトの Makefile の差分

```
PATH := $(PATH):$(PWD)/bin
SHELL := env PATH=$(PATH) /bin/sh
-OS := $(shell uname -s | tr '[:upper:]' '[:lower:]')
-ARCH := $(shell uname -m | sed 's/x86_64/amd64/')
+OS = $(shell uname -s | tr '[:upper:]' '[:lower:]')
+ARCH = $(shell uname -m | sed 's/x86_64/amd64/')
+OSOPER = $(shell uname -s | tr '[:upper:]' '[:lower:]' | sed 's/darwin/apple-darwin/' |
sed 's/linux/linux-gnu/')
+ARCHOPPER = $(shell uname -m )

-# Download kustomize locally if necessary, preferring the $(pwd)/bin path over global
if both exist.
-.PHONY: kustomize
-KUSTOMIZE = $(shell pwd)/bin/kustomize
kustomize:
-ifeq (,$(wildcard $(KUSTOMIZE)))
-ifeq (,$(shell which kustomize 2>/dev/null))
+ifeq (,$(shell which kustomize 2>/dev/null))
@{ \
set -e ;\
-mkdir -p $(dir $(KUSTOMIZE)) ;\
-curl -sSLo - https://github.com/kubernetes-
sigs/kustomize/releases/download/kustomize/v3.5.4/kustomize_v3.5.4_$(OS)_$(ARCH).
tar.gz | \
-tar xzf - -C bin/ ;\
+mkdir -p bin ;\
+curl -sSLo - https://github.com/kubernetes-
sigs/kustomize/releases/download/kustomize/v3.5.4/kustomize_v3.5.4_$(OS)_$(ARCH).
tar.gz | tar xzf - -C bin/ ;\
}
+KUSTOMIZE=$(realpath ./bin/kustomize)
else
-KUSTOMIZE = $(shell which kustomize)
-endif
+KUSTOMIZE=$(shell which kustomize)
```

```

endif

-# Download ansible-operator locally if necessary, preferring the $(pwd)/bin path over
global if both exist.
-.PHONY: ansible-operator
-ANSIBLE_OPERATOR = $(shell pwd)/bin/ansible-operator
ansible-operator:
-ifeq (,$(wildcard $(ANSIBLE_OPERATOR)))
-ifeq (,$(shell which ansible-operator 2>/dev/null))
+ifeq (,$(shell which ansible-operator 2>/dev/null))
@{ \
set -e ;\
- mkdir -p $(dir $(ANSIBLE_OPERATOR)) ;\
- curl -sSLo $(ANSIBLE_OPERATOR) https://github.com/operator-
framework/operator-sdk/releases/download/v1.3.0/ansible-operator_$(OS)_$(ARCH)
;\
- chmod +x $(ANSIBLE_OPERATOR) ;\
+ mkdir -p bin ;\
+ curl -LO https://github.com/operator-framework/operator-
sdk/releases/download/v1.8.0/ansible-operator-v1.8.0-$(ARCHOPER)-$(OSOPER) ;\
+ mv ansible-operator-v1.8.0-$(ARCHOPER)-$(OSOPER) ./bin/ansible-operator ;\
+ chmod +x ./bin/ansible-operator ;\
}
+ANSIBLE_OPERATOR=$(realpath ./bin/ansible-operator)
else
-ANSIBLE_OPERATOR = $(shell which ansible-operator)
-endif
+ANSIBLE_OPERATOR=$(shell which ansible-operator)
endif

```

例5.5 Helm ベースの Operator プロジェクトの Makefile の差分

```

PATH := $(PATH):$(PWD)/bin
SHELL := env PATH=$(PATH) /bin/sh
-OS := $(shell uname -s | tr '[:upper:]' '[:lower:]')
-ARCH := $(shell uname -m | sed 's/x86_64/amd64/')
+OS = $(shell uname -s | tr '[:upper:]' '[:lower:]')
+ARCH = $(shell uname -m | sed 's/x86_64/amd64/')
+OSOPER = $(shell uname -s | tr '[:upper:]' '[:lower:]' | sed 's/darwin/apple-darwin/' |
sed 's/linux/linux-gnu/')
+ARCHOPER = $(shell uname -m )

-# Download kustomize locally if necessary, preferring the $(pwd)/bin path over global
if both exist.
-.PHONY: kustomize
-KUSTOMIZE = $(shell pwd)/bin/kustomize
kustomize:
-ifeq (,$(wildcard $(KUSTOMIZE)))
-ifeq (,$(shell which kustomize 2>/dev/null))
+ifeq (,$(shell which kustomize 2>/dev/null))
@{ \
set -e ;\
- mkdir -p $(dir $(KUSTOMIZE)) ;\
- curl -sSLo - https://github.com/kubernetes-

```



```

sigs/kustomize/releases/download/kustomize/v3.5.4/kustomize_v3.5.4_${OS}_${ARCH}.
tar.gz | \
- tar xzf - -C bin/ ;\
+ mkdir -p bin ;\
+ curl -sSLo - https://github.com/kubernetes-
sigs/kustomize/releases/download/kustomize/v3.5.4/kustomize_v3.5.4_${OS}_${ARCH}.
tar.gz | tar xzf - -C bin/ ;\
}
+KUSTOMIZE=$(realpath ./bin/kustomize)
else
-KUSTOMIZE = $(shell which kustomize)
-endif
+KUSTOMIZE=$(shell which kustomize)
endif

-# Download helm-operator locally if necessary, preferring the $(pwd)/bin path over
global if both exist.
-.PHONY: helm-operator
-HELM_OPERATOR = $(shell pwd)/bin/helm-operator
helm-operator:
-ifeq (,$(wildcard $(HELM_OPERATOR)))
-ifeq (,$(shell which helm-operator 2>/dev/null))
+ifeq (, $(shell which helm-operator 2>/dev/null))
@{ \
set -e ;\
- mkdir -p $(dir $(HELM_OPERATOR)) ;\
- curl -sSLo $(HELM_OPERATOR) https://github.com/operator-framework/operator-
sdk/releases/download/v1.3.0/helm-operator_${OS}_${ARCH} ;\
- chmod +x $(HELM_OPERATOR) ;\
+ mkdir -p bin ;\
+ curl -LO https://github.com/operator-framework/operator-
sdk/releases/download/v1.8.0/helm-operator-v1.8.0-${ARCHOPER}-${OSOPER} ;\
+ mv helm-operator-v1.8.0-${ARCHOPER}-${OSOPER} ./bin/helm-operator ;\
+ chmod +x ./bin/helm-operator ;\
}
+HELM_OPERATOR=$(realpath ./bin/helm-operator)
else
-HELM_OPERATOR = $(shell which helm-operator)
-endif
+HELM_OPERATOR=$(shell which helm-operator)
endif

```

- c. **docker-build** の **make** ターゲットで、位置ディレクトリー引数 `.` を移動します。
docker-build ターゲットのディレクトリー引数 `.` は、**podman** CLI が想定する内容に合わせ最後の位置引数に移動されるので、置換が整理されます。

以前のターゲット

```

docker-build:
    docker build . -t ${IMG}

```

新規ターゲット

```
docker-build:
  docker build -t ${IMG} .
```

以下のコマンドを実行して変更を加えます。

```
$ sed -i 's/docker build . -t ${IMG}/docker build -t ${IMG} ./' $(git grep -l 'docker.*build \.')
```

- d. Ansible および Helm ベースの Operator プロジェクトの場合は、**Makefile** に **ヘルプ** ターゲットを追加します。
- Ansible および Helm ベースのプロジェクトでは、**--help** フラグと同様に、デフォルトで **Makefile** で **ヘルプ** ターゲットを提供するようになりました。以下の行を使用して、このターゲットを **Makefile** に手動で追加できます。

例5.6 help ターゲット

```
##@ General
```

```
# The help target prints out all targets with their descriptions organized
# beneath their categories. The categories are represented by '##@' and the
# target descriptions by '##'. The awk commands is responsible for reading the
# entire set of makefiles included in this invocation, looking for lines of the
# file as xyz: ## something, and then pretty-format the target and help. Then,
# if there's a line with ##@ something, that gets pretty-printed as a category.
# More info on the usage of ANSI control characters for terminal formatting:
# https://en.wikipedia.org/wiki/ANSI_escape_code#SGR_parameters
# More info on the awk command:
# http://linuxcommand.org/lc3_adv_awk.php
```

```
help: ## Display this help.
```

```
@awk 'BEGIN {FS = ":.*##"; printf "\nUsage:\n  make \033[36m<target>\033[0m\n"}
/^[a-zA-Z_0-9-]+:.*?##/ { printf " \033[36m%-15s\033[0m %s\n", $1, $2 } /^##@/ {
printf "\n\033[1m%s\033[0m\n", substr($0, 5) } ' $(MAKEFILE_LIST)
```

- e. **opm** と **catalog-build** ターゲットを追加します。これらのターゲットを使用して Operator の独自のカタログを作成するか、Operator バンドルを既存のカタログに追加できます。

- i. 以下の行を追加して、**Makefile** にターゲットを追加します。

例5.7 opm および catalog-build ターゲット

```
.PHONY: opm
OPM = ./bin/opm
opm:
ifeq (,$(wildcard $(OPM)))
ifeq (,$(shell which opm 2>/dev/null))
@{ \
set -e ;\
mkdir -p $(dir $(OPM)) ;\
curl -sSL $(OPM) https://github.com/operator-framework/operator-
registry/releases/download/v1.15.1/$(OS)-$(ARCH)-opm ;\
chmod +x $(OPM) ;\
}
else
OPM = $(shell which opm)
```

```

endif
endif
BUNDLE_IMGS ?= $(BUNDLE_IMG)
CATALOG_IMG ?= $(IMAGE_TAG_BASE)-catalog:v$(VERSION) ifneq ($(origin
CATALOG_BASE_IMG), undefined) FROM_INDEX_OPT := --from-index
$(CATALOG_BASE_IMG) endif
.PHONY: catalog-build
catalog-build: opm
$(OPM) index add --container-tool docker --mode semver --tag
$(CATALOG_IMG) --bundles $(BUNDLE_IMGS) $(FROM_INDEX_OPT)

.PHONY: catalog-push
catalog-push: ## Push the catalog image.
$(MAKE) docker-push IMG=$(CATALOG_IMG)

```

- ii. Go ベースの Operator プロジェクトを更新する場合は、以下の **Makefile** 変数も追加します。

例5.8 Makefile 変数

```

OS = $(shell go env GOOS)
ARCH = $(shell go env GOARCH)

```

- f. Go ベースの Operator プロジェクトの場合は、**Makefile** の **SHELL** 変数をシステム **bash** バイナリーに設定します。

setup-envtest.sh スクリプトをインポートするには **bash** が必要であるため、**SHELL** 変数をエラーオプションで **bash** に設定する必要があります。

例5.9 Makefile の差分

```

else GOBIN=$(shell go env GOBIN)
endif
+## Setting SHELL to bash allows bash commands to be executed by recipes.
+## This is a requirement for 'setup-envtest.sh' in the test target.
+## Options are set to exit when a recipe line exits non-zero or a piped command fails.
+SHELL = /usr/bin/env bash -o pipefail
+.SHELLFLAGS = -ec
+ all: build

```

4. Go ベースの Operator プロジェクトの場合は、**go.mod** ファイルの以下のエントリーを変更して **controller-runtime** を v0.8.3 に、Kubernetes の依存関係を v0.20.2 にアップグレードし、プロジェクトを再ビルドします。

例5.10 go.mod ファイル

```

...
k8s.io/api v0.20.2
k8s.io/apimachinery v0.20.2
k8s.io/client-go v0.20.2
sigs.k8s.io/controller-runtime v0.8.3

```

5. **system:controller-manager** サービスアカウントをプロジェクトに追加します。デフォルト以外のサービスアカウント **controller-manager** が **operator-sdk init** コマンドで生成されるようになり、共有 namespace にインストールされている Operator のセキュリティが改善されます。このサービスアカウントを既存プロジェクトに追加するには、以下の手順に従います。

- a. ファイルに **ServiceAccount** 定義を作成します。

例5.11 config/rbac/service_account.yaml ファイル

```
apiVersion: v1
kind: ServiceAccount
metadata:
  name: controller-manager
  namespace: system
```

- b. サービスアカウントを RBAC リソースの一覧に追加します。

```
$ echo "- service_account.yaml" >> config/rbac/kustomization.yaml
```

- c. Operator のサービスアカウントを参照する **RoleBinding** および **ClusterRoleBinding** オブジェクトをすべて更新します。

```
$ find config/rbac -name *_binding.yaml -exec sed -i -E 's/ name: default/ name: controller-manager/g' {} \;
```

- d. サービスアカウント名をマネージャーデプロイメントの **spec.template.spec.serviceAccountName** フィールドに追加します。

```
$ sed -i -E 's/([ ]+)(terminationGracePeriodSeconds:)/\1serviceAccountName: controller-manager\n\1\2/g' config/manager/manager.yaml
```

- e. 変更が以下の差分のようになっていることを確認します。

例5.12 config/manager/manager.yaml ファイルの差分

```
...
  requests:
    cpu: 100m
    memory: 20Mi
+ serviceAccountName: controller-manager
  terminationGracePeriodSeconds: 10
```

例5.13 config/rbac/auth_proxy_role_binding.yaml ファイルの差分

```
...
  name: proxy-role
  subjects:
    - kind: ServiceAccount
    - name: default
+ name: controller-manager
  namespace: system
```

例5.14 config/rbac/kustomization.yaml ファイルの差分

```
resources:
+- service_account.yaml
- role.yaml
- role_binding.yaml
- leader_election_role.yaml
```

例5.15 config/rbac/leader_election_role_binding.yaml ファイルの差分

```
...
name: leader-election-role
subjects:
- kind: ServiceAccount
- name: default
+ name: controller-manager
namespace: system
```

例5.16 config/rbac/role_binding.yaml ファイルの差分

```
...
name: manager-role
subjects:
- kind: ServiceAccount
- name: default
+ name: controller-manager
namespace: system
```

例5.17 config/rbac/service_account.yaml ファイルの差分

```
+apiVersion: v1
+kind: ServiceAccount
+metadata:
+ name: controller-manager
+ namespace: system
```

6. config/manifests/kustomization.yaml ファイルに以下の変更を加えます。

- a. [Kustomize](#) パッチを追加して、[cert-manager](#) ボリューム および **volumeMount** オブジェクトをクラスターサービスバージョン (CSV) から削除します。

Operator Lifecycle Manager (OLM) ではまだ [cert-manager](#) がサポートされないため、OLM が Operator 用の証明書を作成して管理できるように、このボリュームを削除してマウントするように、JSON パッチが追加されました。

config/manifests/kustomization.yaml ファイルに、以下の行を追加します。

例5.18 config/manifests/kustomization.yaml ファイル

```

patchesJson6902:
- target:
  group: apps
  version: v1
  kind: Deployment
  name: controller-manager
  namespace: system
  patch: |-
    # Remove the manager container's "cert" volumeMount, since OLM will create and
    # mount a set of certs.
    # Update the indices in this path if adding or removing containers/volumeMounts in
    # the manager's Deployment.
    - op: remove
      path: /spec/template/spec/containers/1/volumeMounts/0
    # Remove the "cert" volume, since OLM will create and mount a set of certs.
    # Update the indices in this path if adding or removing volumes in the manager's
    # Deployment.
    - op: remove
      path: /spec/template/spec/volumes/0

```

- b. オプション: Ansible および Helm ベースの Operator プロジェクトの場合は、コンポーネント設定で **ansible-operator** および **helm-operator** を設定します。このオプションを追加するには、以下の手順に従います。

- i. 以下のファイルを作成します。

例5.19 config/default/manager_config_patch.yaml ファイル

```

apiVersion: apps/v1
kind: Deployment
metadata:
  name: controller-manager
  namespace: system
spec:
  template:
    spec:
      containers:
      - name: manager
        args:
        - "--config=controller_manager_config.yaml"
        volumeMounts:
        - name: manager-config
          mountPath: /controller_manager_config.yaml
          subPath: controller_manager_config.yaml
      volumes:
      - name: manager-config
        configMap:
          name: manager-config

```

- ii. 以下のファイルを作成します。

例5.20 config/manager/controller_manager_config.yaml ファイル

```

apiVersion: controller-runtime.sigs.k8s.io/v1alpha1

```

```

kind: ControllerManagerConfig
health:
  healthProbeBindAddress: :6789
metrics:
  bindAddress: 127.0.0.1:8080

leaderElection:
  leaderElect: true
  resourceName: <resource_name>

```

- iii. 以下の変更を **resources** に適用して、**config/default/kustomization.yaml** ファイルを更新します。

例5.21 config/default/kustomization.yaml ファイル

```

resources:
...
- manager_config_patch.yaml

```

- iv. 以下の変更を適用して、**config/manager/kustomization.yaml** ファイルを更新します。

例5.22 config/manager/kustomization.yaml ファイル

```

generatorOptions:
  disableNameSuffixHash: true

configMapGenerator:
- files:
  - controller_manager_config.yaml
  name: manager-config
  apiVersion: kustomize.config.k8s.io/v1beta1
  kind: Kustomization
  images:
  - name: controller
    newName: quay.io/example/memcached-operator
    newTag: v0.0.1

```

- c. オプション: **config/default/kustomization.yaml** ファイルにマネージャー設定パッチを追加します。

[config file](#) のサポートが最初に追加されたタイミングで、生成された **--config** フラグが **ansible-operator** または **helm-operator** バイナリーのいずれかに追加されていなかったため、現在は機能しません。**--config** フラグは、ファイルによるバイナリーの設定をサポートします。この設定の方法は、Operator 全体としてではなく、基盤の [コントローラーマネージャー](#) にのみ適用されます。

オプションで設定ファイルで Operator のデプロイメントを設定するには、以下の差分に示されるように **config/default/kustomization.yaml** ファイルに変更を加えます。

例5.23 config/default/kustomization.yaml ファイルの差分

```

# If you want your controller-manager to expose the /metrics # endpoint w/o any

```

```

authn/z, please comment the following line.
\~ manager_auth_proxy_patch.yaml
+# Mount the controller config file for loading manager configurations
+# through a ComponentConfig type
+- manager_config_patch.yaml

```

フラグはそのまま使用したり、設定ファイルの値を上書きしたりできます。

7. Ansible および Helm ベースの Operator プロジェクトについては、**config/rbac/leader_election_role.yaml** ファイルに以下の変更を加え、リーダー選出のロールルールを追加します。

例5.24 config/rbac/leader_election_role.yaml ファイル

```

- apiGroups:
  - coordination.k8s.io
  resources:
  - leases
  verbs:
  - get
  - list
  - watch
  - create
  - update
  - patch
  - delete

```

8. Ansible ベースの Operator プロジェクトの場合は、Ansible コレクションを更新します。**requirements.yml** ファイルで **community.kubernetes** の バージョン フィールドを **1.2.1** に変更し、**operator_sdk.util** の バージョン フィールドを **0.2.0** に変更します。

9. **config/default/manager_auth_proxy_patch.yaml** ファイルに以下の変更を加えます。

- Ansible ベースの Operator プロジェクトの場合は、**--health-probe-bind-address=:6789** 引数を **config/default/manager_auth_proxy_patch.yaml** ファイルに追加します。

例5.25 config/default/manager_auth_proxy_patch.yaml ファイル

```

spec:
  template:
    spec:
      containers:
      - name: manager
        args:
        - "--health-probe-bind-address=:6789"
        ...

```

- Helm ベースの Operator のプロジェクト:
 - i. **--health-probe-bind-address=:8081** 引数を **config/default/manager_auth_proxy_patch.yaml** ファイルに追加します。

例5.26 config/default/manager_auth_proxy_patch.yaml ファイル

```

-

```



```
spec:
  template:
    spec:
      containers:
      - name: manager
        args:
        - "--health-probe-bind-address=:8081"
        ...
```

- ii. 非推奨のフラグ **--enable-leader-election** を **--leader-elect** に置き換え、非推奨のフラグ **--metrics-addr** を **--metrics-bind-address** に置き換えます。

10. **config/prometheus/monitor.yaml** ファイルに以下の変更を加えます。

- a. スキーム、トークン、および TLS 設定を Prometheus **ServiceMonitor** メトリクスエンドポイントに追加します。

マネージャー Pod で **https** ポートを指定しているにも拘らず、**tlsConfig** が設定されていないので、**/metrics** エンドポイントが HTTPS 経由でサービスを提供されるように実際には設定されていません。**kube-rbac-proxy** は、Pod にマウントされたサービスアカウントトークンを使用して、このエンドポイントのセキュリティをマネージャーサイドカーコンテナとして確保するので、デフォルトでこの問題が修正されます。

以下の差分に示されるように、**config/prometheus/monitor.yaml** ファイルに変更を適用します。

例5.27 config/prometheus/monitor.yaml ファイルの差分

```
spec:
  endpoints:
    - path: /metrics
      port: https
    + scheme: https
    + bearerTokenFile: /var/run/secrets/kubernetes.io/serviceaccount/token
    + tlsConfig:
    +   insecureSkipVerify: true
  selector:
    matchLabels:
      control-plane: controller-manager
```



注記

kube-rbac-proxy をプロジェクトから削除した場合には、適切な **TLS 設定** を使用して **/metrics** エンドポイントのセキュリティを確保するようにしてください。

- 11. 既存の依存リソースに所有者アノテーションがあることを確認します。

Ansible ベースの Operator プロジェクトの場合には、クラスタースコープの依存リソースおよび他の namespace の依存関係リソースで [所有者の参照アノテーション](#) が正しく適用されました。これらのアノテーションを手動で追加することが回避策でしたが、このバグが修正されたため、これは必要なくなりました。

- 12. パッケージマニフェストのサポートを非推奨にします。

[Operator Framework](#) では、今後のリリースで Operator パッケージマニフェスト形式のサポートが削除されます。非推奨のプロセスの一環として、**operator-sdk generate packagemanifests** および **operator-sdk run packagemanifests** コマンドが非推奨となりました。パッケージマニフェストをバンドルに移行するには、**operator-sdk pkgman-to-bundle** コマンドを使用できます。

operator-sdk pkgman-to-bundle --help コマンドを実行して、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を参照してください。

13. Operator のファイナライザー名を更新します。
[Kubernetes ドキュメント](#) が提案するファイナライザーの名前の形式は、以下のとおりです。

```
<qualified_group>/<finalizer_name>
```

以前の Operator SDK の記述形式は次のとおりです。

```
<finalizer_name>.<qualified_group>
```

Operator が、名前の形式が不適切なファイナライザーを使用する場合は、公式の形式に一致するようにこれらのファイナライザーを変更します。たとえば、**finalizer.cache.example.com** を **cache.example.com/finalizer** に変更する必要があります。

Operator プロジェクトは Operator SDK v1.8.0 との互換性が確保されました。

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](#)

5.4. GO ベースの OPERATOR

5.4.1. Go ベースの Operator の Operator SDK の使用を開始する

Operator SDK によって提供されるツールおよびライブラリーを使用して Go ベースの Operator をセットアップし、実行することに関連した基本内容を示すには、Operator 開発者は Go ベースの Memcached の Operator のサンプル、分散キー/値のストアをビルドして、クラスターへデプロイすることができます。

5.4.1.1. 前提条件

- [Operator SDK CLI](#) がインストールされていること。
- [OpenShift CLI \(oc\)](#) v4.8+ がインストールされていること。
- **cluster-admin** パーミッションを持つアカウントを使用して、**oc** で OpenShift Container Platform 4.8 クラスターにログインしていること。
- クラスターがイメージをプルできるようにするには、イメージをプッシュするリポジトリを public として設定するか、またはイメージプルシークレットを設定する必要があります。

5.4.1.2. Go ベースの Operator の作成およびデプロイ

Operator SDK を使用して Memcached の単純な Go ベースの Operator をビルドし、デプロイできます。

手順

1. プロジェクトを作成します。

- a. プロジェクトディレクトリーを作成します。

```
$ mkdir memcached-operator
```

- b. プロジェクトディレクトリーに移動します。

```
$ cd memcached-operator
```

- c. **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
```

このコマンドは、デフォルトで Go プラグインを使用します。

2. API を作成します。

単純な Memcached API を作成します。

```
$ operator-sdk create api \
  --resource=true \
  --controller=true \
  --group cache \
  --version v1 \
  --kind Memcached
```

3. Operator イメージをビルドし、プッシュします。

デフォルトの **Makefile** ターゲットを使用して Operator をビルドし、プッシュします。プッシュ先となるレジストリーを使用するイメージのプル仕様を使用して **IMG** を設定します。

```
$ make docker-build docker-push IMG=<registry>/<user>/<image_name>:<tag>
```

4. Operator を実行します。

- a. CRD をインストールします。

```
$ make install
```

- b. プロジェクトをクラスターにデプロイします。 **IMG** をプッシュしたイメージに設定します。

```
$ make deploy IMG=<registry>/<user>/<image_name>:<tag>
```

5. サンプルカスタムリソース (CR) を作成します。

- a. サンプル CR を作成します。

```
$ oc apply -f config/samples/cache_v1_memcached.yaml \
  -n memcached-operator-system
```

b. Operator を調整する CR を確認します。

```
$ oc logs deployment.apps/memcached-operator-controller-manager \
  -c manager \
  -n memcached-operator-system
```

6. クリーンアップします。

以下のコマンドを実行して、この手順の一部として作成されたリソースをクリーンアップします。

```
$ make undeploy
```

5.4.1.3. 次のステップ

- Go ベースの Operator のビルドに関する詳細な手順は、[Go ベースの Operator の Operator SDK チュートリアル](#) を参照してください。

5.4.2. 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、[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。

5.4.2.1. 前提条件

- [Operator SDK CLI がインストールされていること。](#)
- [OpenShift CLI \(oc\) v4.8+ がインストールされていること。](#)
- cluster-admin** パーミッションを持つアカウントを使用して、**oc** で OpenShift Container Platform 4.8 クラスターにログインしていること。
- クラスターがイメージをプルできるようにするには、イメージをプッシュするリポジトリを public として設定するか、またはイメージプルシークレットを設定する必要があります。

5.4.2.2. 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。

5.4.2.2.1. 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: {}
```

5.4.2.2.2. 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 ❶
mgr, err := ctrl.NewManager(cfg, manager.Options{ ❷
    NewCache: cache.MultiNamespacedCacheBuilder(namespaces),
})
```

❶ namespace の一覧

❷ **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。

5.4.2.2.3. 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。

5.4.2.3. 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。

5.4.2.3.1. 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。

5.4.2.3.2. 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。

5.4.2.3.2.1. 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 のプロパティを検証できます。

API の検証を設定するには、マーカまたはアノテーションを使用できます。これらのマーカには、**+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。

5.4.2.4. 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5.28 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"

    cachev1alpha1 "github.com/example/memcached-operator/api/v1alpha1"
)

// 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;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;

// 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)

    // Fetch the Memcached instance
    memcached := &cachev1alpha1.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
*cachev1alpha1.Memcached) *apps1.Deployment {
    ls := labelsForMemcached(m.Name)
    replicas := m.Spec.Size

    dep := &apps1.Deployment{
        ObjectMeta: metav1.ObjectMeta{
            Name:      m.Name,
            Namespace: m.Namespace,
        },
        Spec: apps1.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(&cachev1alpha1.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。

5.4.2.4.1. 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。

5.4.2.4.2. 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ーリングします。
- `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。

5.4.2.4.3. 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。

5.4.2.4.4. 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;update;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** ユーティリティを使用して、以前のマーカーから生成されます。

5.4.2.5. Operator の実行

Operator SDK CLI を使用して Operator をビルドし、実行する方法は 3 つあります。

- 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。
- Operator をバンドルし、Operator Lifecycle Manager (OLM) を使用してクラスター上にデプロイします。



注記

Go ベースの Operator を OpenShift Container Platform でのデプロイメントとして、または OLM を使用するバンドルとして実行する前に、プロジェクトがサポートされているイメージを使用するように更新されていることを確認します。

5.4.2.5.1. クラスター外でローカルに実行する。

Operator プロジェクトをクラスター外の Go プログラムとして実行できます。これは、デプロイメントとテストを迅速化するという開発目的において便利です。

手順

- 以下のコマンドを実行して、**~/.kube/config** ファイルに設定されたクラスターにカスタムリソース定義 (CRD) をインストールし、Operator をローカルで実行します。

```
$ make install run
```

出力例

```
...
2021-01-10T21:09:29.016-0700 INFO controller-runtime.metrics metrics server is starting to
listen {"addr": ":8080"}
2021-01-10T21:09:29.017-0700 INFO setup starting manager
```



```

2021-01-10T21:09:29.017-0700 INFO controller-runtime.manager starting metrics server
{"path": "/metrics"}
2021-01-10T21:09:29.018-0700 INFO controller-runtime.manager.controller.memcached
Starting EventSource {"reconciler group": "cache.example.com", "reconciler kind":
"Memcached", "source": "kind source: /, Kind="}
2021-01-10T21:09:29.218-0700 INFO controller-runtime.manager.controller.memcached
Starting Controller {"reconciler group": "cache.example.com", "reconciler kind":
"Memcached"}
2021-01-10T21:09:29.218-0700 INFO controller-runtime.manager.controller.memcached
Starting workers {"reconciler group": "cache.example.com", "reconciler kind": "Memcached",
"worker count": 1}

```

5.4.2.5.2. クラスター上でのデプロイメントとしての実行

Operator プロジェクトは、クラスター上でのデプロイメントとして実行することができます。

前提条件

- プロジェクトを更新してサポートされるイメージを使用することで、OpenShift Container Platform で実行する Go ベースの Operator が準備済みである。

手順

- 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を参照します。Quay.io などのリポジトリサイトにコンテナを保存するためのアカウントを取得できます。

- 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。

- イメージをリポジトリにプッシュします。

```
$ 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。

- 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. 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
```

5.4.2.5.3. Operator のバンドルおよび Operator Lifecycle Manager を使用したデプロイ

5.4.2.5.3.1. 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こと。
- OpenShift CLI (**oc**) v4.8+ がインストールされていること。
- Operator プロジェクトが Operator SDK を使用して初期化されていること。
- Operator が Go ベースの場合、プロジェクトを更新して OpenShift Container Platform での実行をサポートするイメージを使用する必要がある。

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を参照します。Quay.io などのリポジトリサイトにコンテナ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>
```

5.4.2.5.3.2. 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 Container Platform にインストールされ、Kubernetes 拡張として実行されるため、追加のツールなしにすべての Operator のライフサイクル管理機能に Web コンソールおよび 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こと。
- ビルドされ、レジストリーにプッシュされる Operator バンドルイメージ。

- (OpenShift Container Platform 4.8 など、**apiextensions.k8s.io/v1** CRD を使用する場合は v1.16.0 以降の) Kubernetes ベースのクラスターに OLM がインストールされていること。
- **cluster-admin** パーミッションのあるアカウントを使用して **oc** でクラスターへログインしていること。
- Operator が Go ベースの場合、プロジェクトを更新して OpenShift Container Platform での実行をサポートするイメージを使用する必要がある。

手順

1. 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 \
  [-n <namespace>] \1
  <registry>/<user>/<bundle_image_name>:<tag>
```

- 1 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。

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オブジェクトすべてを作成して、Operator をクラスターにデプロイします。

5.4.2.6. 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 **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-6fd7c98d8-7dqd8	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. 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>
```

5.4.2.7. 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、[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。

5.4.3. 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。

5.4.3.1. 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。
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。

5.5. ANSIBLE ベース OPERATOR

5.5.1. Ansible ベースの Operator の Operator SDK の使用を開始する

Operator プロジェクトを生成するための Operator SDK には、Go コードを作成せずに Kubernetes リソースを統一されたアプリケーションとしてデプロイするために、既存の Ansible Playbook およびモジュールを使用するオプションがあります。

Operator SDK によって提供されるツールおよびライブラリーを使用して [Ansible](#) ベースの Operator をセットアップし、実行するための基本を示すには、Operator 開発者は 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、クラスターへデプロイすることができます。

5.5.1.1. 前提条件

- [Operator SDK CLI](#) がインストールされていること。
- [OpenShift CLI \(oc\)](#) v4.8+ がインストールされていること。
- [Ansible](#) バージョン v2.9.0
- [Ansible Runner](#) バージョン v1.1.0+
- [Ansible Runner HTTP Event Emitter プラグイン](#) バージョン v1.0.0+
- [OpenShift Python クライアント](#) バージョン v0.11.2+

- **cluster-admin** パーミッションを持つアカウントを使用して、**oc** で OpenShift Container Platform 4.8 クラスターにログインしていること。
- クラスターがイメージをプルできるようにするには、イメージをプッシュするリポジトリを **public** として設定するか、またはイメージプルシークレットを設定する必要があります。

5.5.1.2. Ansible ベース Operator の作成およびデプロイ

Operator SDK を使用して、Memcached の単純な Ansible ベースの Operator をビルドし、デプロイできます。

手順

1. プロジェクトを作成します。

- プロジェクトディレクトリーを作成します。

```
$ mkdir memcached-operator
```

- プロジェクトディレクトリーに移動します。

```
$ cd memcached-operator
```

- 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
```

2. API を作成します。

単純な Memcached API を作成します。

```
$ operator-sdk create api \
  --group cache \
  --version v1 \
  --kind Memcached \
  --generate-role 1
```

- API の Ansible ロールを生成します。

3. Operator イメージをビルドし、プッシュします。

デフォルトの **Makefile** ターゲットを使用して Operator をビルドし、プッシュします。プッシュ先となるレジストリーを使用するイメージのプル仕様を使用して **IMG** を設定します。

```
$ make docker-build docker-push IMG=<registry>/<user>/<image_name>:<tag>
```

4. Operator を実行します。

- CRD をインストールします。

```
$ make install
```

- b. プロジェクトをクラスターにデプロイします。 **IMG** をプッシュしたイメージに設定します。

```
$ make deploy IMG=<registry>/<user>/<image_name>:<tag>
```

5. サンプルカスタムリソース (CR) を作成します。

- a. サンプル CR を作成します。

```
$ oc apply -f config/samples/cache_v1_memcached.yaml \
-n memcached-operator-system
```

- b. Operator を調整する CR を確認します。

```
$ oc logs deployment.apps/memcached-operator-controller-manager \
-c manager \
-n memcached-operator-system
```

出力例

```
...
I0205 17:48:45.881666    7 leaderelection.go:253] successfully acquired lease
memcached-operator-system/memcached-operator
{"level":"info","ts":1612547325.8819902,"logger":"controller-
runtime.manager.controller.memcached-controller","msg":"Starting
EventSource","source":"kind source: cache.example.com/v1, Kind=Memcached"}
{"level":"info","ts":1612547325.98242,"logger":"controller-
runtime.manager.controller.memcached-controller","msg":"Starting Controller"}
{"level":"info","ts":1612547325.9824686,"logger":"controller-
runtime.manager.controller.memcached-controller","msg":"Starting workers","worker
count":4}
{"level":"info","ts":1612547348.8311093,"logger":"runner","msg":"Ansible-runner exited
successfully","job":"4037200794235010051","name":"memcached-
sample","namespace":"memcached-operator-system"}
```

6. クリーンアップします。

以下のコマンドを実行して、この手順の一部として作成されたリソースをクリーンアップします。

```
$ make undeploy
```

5.5.1.3. 次のステップ

- Ansible ベースの Operator のビルドに関する詳細な手順は、[Ansible ベース Operator の Operator SDK チュートリアル](#) を参照してください。

5.5.2. 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、[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。

5.5.2.1. 前提条件

- [Operator SDK CLI](#) がインストールされていること。
- [OpenShift CLI \(oc\)](#) v4.8+ がインストールされていること。
- [Ansible](#) バージョン v2.9.0
- [Ansible Runner](#) バージョン v1.1.0+
- [Ansible Runner HTTP Event Emitter プラグイン](#) バージョン v1.0.0+
- [OpenShift Python クライアント](#) バージョン v0.11.2+
- **cluster-admin** パーミッションを持つアカウントを使用して、**oc** で OpenShift Container Platform 4.8 クラスターにログインしていること。
- クラスターがイメージをプルできるようにするには、イメージをプッシュするリポジトリを public として設定するか、またはイメージプルシークレットを設定する必要があります。

5.5.2.2. 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
```

5.5.2.2.1. 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
projectName: memcached-operator
version: 3
```

5.5.2.3. 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

5.5.2.4. 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
  community.kubernetes.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:
  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。

5.5.2.5. Operator の実行

Operator SDK CLI を使用して Operator をビルドし、実行する方法は 3 つあります。

- 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。
- Operator をバンドルし、Operator Lifecycle Manager (OLM) を使用してクラスター上にデプロイします。

5.5.2.5.1. クラスター外でローカルに実行する。

Operator プロジェクトをクラスター外の Go プログラムとして実行できます。これは、デプロイメントとテストを迅速化するという開発目的において便利です。

手順

- 以下のコマンドを実行して、`~/kube/config` ファイルに設定されたクラスターにカスタムリソース定義 (CRD) をインストールし、Operator をローカルで実行します。

```
$ make install run
```

出力例

```
...
{"level":"info","ts":1612589622.7888272,"logger":"ansible-controller","msg":"Watching resource","Options.Group":"cache.example.com","Options.Version":"v1","Options.Kind":"Memcached"}
{"level":"info","ts":1612589622.7897573,"logger":"proxy","msg":"Starting to serve","Address":"127.0.0.1:8888"}
{"level":"info","ts":1612589622.789971,"logger":"controller-runtime.manager","msg":"starting metrics server","path":"/metrics"}
{"level":"info","ts":1612589622.7899997,"logger":"controller-runtime.manager.controller.memcached-controller","msg":"Starting EventSource","source":"kind source: cache.example.com/v1, Kind=Memcached"}
{"level":"info","ts":1612589622.8904517,"logger":"controller-runtime.manager.controller.memcached-controller","msg":"Starting Controller"}
{"level":"info","ts":1612589622.8905244,"logger":"controller-runtime.manager.controller.memcached-controller","msg":"Starting workers","worker count":8}
```

5.5.2.5.2. クラスター上でのデプロイメントとしての実行

Operator プロジェクトは、クラスター上でのデプロイメントとして実行することができます。

手順

- 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を参照します。Quay.io などのリポジトリサイトにコンテナを保存するためのアカウントを取得できます。

- 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。

- イメージをリポジトリにプッシュします。

```
$ 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。

- 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。

- 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
```

5.5.2.5.3. Operator のバンドルおよび Operator Lifecycle Manager を使用したデプロイ

5.5.2.5.3.1. 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こと。
- OpenShift CLI (**oc**) v4.8+ がインストールされていること。
- Operator プロジェクトが Operator SDK を使用して初期化されていること。

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を参照します。Quay.io などのリポジトリサイトにコンテナ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>
```

5.5.2.5.3.2. 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 Container Platform にインストールされ、Kubernetes 拡張として実行されるため、追加のツールなしにすべての Operator のライフサイクル管理機能に Web コンソールおよび 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こと。
- ビルドされ、レジストリーにプッシュされる Operator バンドルイメージ。
- (OpenShift Container Platform 4.8 など、**apiextensions.k8s.io/v1** CRD を使用する場合は v1.16.0 以降の) Kubernetes ベースのクラスターに OLM がインストールされていること。
- **cluster-admin** パーミッションのあるアカウントを使用して **oc** でクラスターへログインしていること。

手順

1. 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 \
  [-n <namespace>] ❶
  <registry>/<user>/<bundle_image_name>:<tag>
```

- ❶ 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。

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。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オブジェクトすべてを作成して、Operator をクラスターにデプロイします。

5.5.2.6. 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 **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. 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>
```

5.5.2.7. 関連情報

- Operator SDK によって作成されるディレクトリ構造の詳細は、[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。

5.5.3. 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。

5.5.3.1. 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	目的
Dockerfile	Operator のコンテナ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のロールとロールバインディング。

ファイルまたはディレクトリー	目的
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。
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。

5.5.4. Operator SDK における Ansible サポート

5.5.4.1. 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一覧。このフィールドは、デフォルトでは空です。
status	オブジェクトの現在の状態の概要を示します。Ansible ベースの Operator の場合、 status サブリソース はデフォルトで CRD について有効にされ、 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一覧は 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"
```

5.5.4.2. 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 へのマッピングの一覧が含まれます。Operator はこのマッピングファイルが事前に定義された場所の **/opt/ansible/watches.yaml** にあることを予想します。

表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。

フィールド	説明
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 ❶
  group: test1.example.com
  kind: Test1
  role: /opt/ansible/roles/Test1

- version: v1alpha1 ❷
  group: test2.example.com
  kind: Test2
  playbook: /opt/ansible/playbook.yml

- version: v1alpha1 ❸
  group: test3.example.com
  kind: Test3
  playbook: /opt/ansible/test3.yml
  reconcilePeriod: 0
  manageStatus: false

```

- ❶ **Test1** の **test1** ロールへの単純なマッピングの例。
- ❷ **Test2** の Playbook への単純なマッピングの例。
- ❸ **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。

5.5.4.2.1. 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
----	---------	----	-------------	--------

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
```

5.5.4.3. 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 }}"

```

5.5.4.4. Ansible Runner ディレクトリー

Ansible Runner はコンテナに 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。

5.5.5. 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、またはネイティブの Ansible でライフサイクル管理を表現することができます。

Ansible を既存の Kubernetes リソースファイルと併用する最大の利点の1つに、Ansible のいくつかを変数のみを使う単純な方法でのリソースのカスタマイズを可能にする Jinja テンプレートを使用できる点があります。

このセクションでは、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。

5.5.5.1. 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 2.9+ をインストールします。

```
$ sudo dnf install ansible
```

2. [OpenShift python クライアント](#) パッケージをインストールします。

```
$ pip3 install openshift
```

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
```

5.5.5.2. 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: default ❶
state: "{{ state }}"
ignore_errors: true ❷
```

- ❶ **default** から別の namespace に設定マップを作成する場合には、この値を変更します。
- ❷ **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
```

5.5.5.3. 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。

5.5.6. Operator 内での Ansible の使用

[Kubernetes Collection for Ansible をローカルで使用する](#)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。

5.5.6.1. 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一覧。このフィールドは、デフォルトでは空です。
status	オブジェクトの現在の状態の概要を示します。Ansible ベースの Operator の場合、 status サブリソース はデフォルトで CRD について有効にされ、 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一覧は 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"
```

5.5.6.2. 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、または **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 バージョン v1.1.0+](#)
- [Ansible Runner HTTP Event Emitter プラグイン バージョン 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":"v1.8.0","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":false}
{"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
```

5.5.6.3. 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でテストした後に、OpenShift Container Platform クラスターの Pod 内で Operator をテストすることができます。これは実稼働環境での使用が推奨されます。

Operator プロジェクトは、クラスター上でのデプロイメントとして実行することができます。

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を参照します。Quay.io などのリポジトリサイトにコンテナ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. 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
```

5.5.6.4. 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。

5.5.6.4.1. 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 \ ❶
  -n <namespace> ❷
```

❶ **manager** コンテナのログを表示します。

❷ **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": "v1.8.0",
  "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": false
}
{
  "level": "info",
  "ts": 1612732107.7688901,
  "logger": "ansible-controller",
  "msg": "Watching
```

```
resource","Options.Group":"cache.example.com","Options.Version":"v1","Options.Kind":"Memcached"}
{"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}
```

5.5.6.4.2. 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"
```

5.5.6.4.3. 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
```


5.5.7. カスタムリソースのステータス管理

5.5.7.1. 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/値のペアを使って Ansible から **status** を更新できます。

デフォルトでは、Ansible ベースの Operator には、上記のように常に汎用的な Ansible 実行出力が含まれます。アプリケーションのステータスが Ansible 出力で更新 **されない** ようにする必要がある場合は、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。

5.5.7.2. 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

- 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
```

5.6. HELM ベースの OPERATOR

5.6.1. Helm ベースの Operator の Operator SDK の使用を開始する

Operator プロジェクトを生成するための Operator SDK には、Go コードを作成せずに Kubernetes リソースを統一されたアプリケーションとしてデプロイするために、既存の [Helm](#) チャートを使用するオプションがあります。

Operator SDK によって提供されるツールおよびライブラリーを使用して [Helm](#) ベースの Operator をセットアップし、実行するための基本を示すには、Operator 開発者は Helm ベースの Nginx Operator のサンプルをビルドし、これをクラスターヘデプロイすることができます。

5.6.1.1. 前提条件

- [Operator SDK CLI](#) がインストールされていること。
- [OpenShift CLI \(oc\)](#) v4.8+ がインストールされていること。
- **cluster-admin** パーミッションを持つアカウントを使用して、**oc** で OpenShift Container Platform 4.8 クラスターにログインしていること。
- クラスターがイメージをプルできるようにするには、イメージをプッシュするリポジトリを public として設定するか、またはイメージプルシークレットを設定する必要があります。

5.6.1.2. Helm ベースの Operator の作成とデプロイ

Operator SDK を使用して Nginx の単純な Helm ベースの Operator をビルドし、デプロイできます。

手順

1. プロジェクトを作成します。

- a. プロジェクトディレクトリーを作成します。

```
$ mkdir nginx-operator
```

- b. プロジェクトディレクトリーに移動します。

```
$ cd nginx-operator
```

- c. **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
```

2. API を作成します。

単純な Nginx API を作成します。

```
$ operator-sdk create api \
  --group demo \
  --version v1 \
  --kind Nginx
```

この API は、**helm create** コマンドでビルトインの Helm チャートボイラープレートを使用します。

3. Operator イメージをビルドし、プッシュします。

デフォルトの **Makefile** ターゲットを使用して Operator をビルドし、プッシュします。プッシュ先となるレジストリーを使用するイメージのプル仕様を使用して **IMG** を設定します。

```
$ make docker-build docker-push IMG=<registry>/<user>/<image_name>:<tag>
```

4. Operator を実行します。

- a. CRD をインストールします。

```
$ make install
```

- b. プロジェクトをクラスターにデプロイします。 **IMG** をプッシュしたイメージに設定します。

```
$ make deploy IMG=<registry>/<user>/<image_name>:<tag>
```

5. SCC(Security Context Constraints) を追加します。

Nginx サービスアカウントには、OpenShift Container Platform で実行する特権アクセスが必要です。以下の SCC 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
```

6. サンプルカスタムリソース (CR) を作成します。

- a. サンプル CR を作成します。

```
$ oc apply -f config/samples/demo_v1_nginx.yaml \
-n nginx-operator-system
```

- b. Operator を調整する CR を確認します。

```
$ oc logs deployment.apps/nginx-operator-controller-manager \
-c manager \
-n nginx-operator-system
```

7. クリーンアップします。

以下のコマンドを実行して、この手順の一部として作成されたリソースをクリーンアップします。

```
$ make undeploy
```

5.6.1.3. 次のステップ

- Helm ベースの Operator のビルドに関する詳細な手順は、[Helm ベースの Operator の Operator SDK チュートリアル](#) を参照してください。

5.6.2. 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。

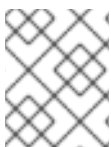
このプロセスは、Operator Framework の 2 つの重要な設定要素を使用して実行されます。

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、[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。

5.6.2.1. 前提条件

- [Operator SDK CLI がインストールされていること。](#)

- OpenShift CLI (**oc**) v4.8+ がインストールされていること。
- **cluster-admin** パーミッションを持つアカウントを使用して、**oc** で OpenShift Container Platform 4.8 クラスターにログインしていること。
- クラスターがイメージをプルできるようにするには、イメージをプッシュするリポジトリを **public** として設定するか、またはイメージプルシークレットを設定する必要があります。

5.6.2.2. 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。

5.6.2.2.1. 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を指定する場合、チャートは検証され、プロジェクトに展開されるかコピーされます。そうでない場合は、Operator SDK はリモトリポジトリ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
<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
```

5.6.2.2.2. 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
projectName: helm-operator
resources:
- group: demo
  kind: Nginx
  version: v1
version: 3
```

5.6.2.3. 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
```

5.6.2.3.1. 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。

5.6.2.3.2. 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。

5.6.2.4. Operator の実行

Operator SDK CLI を使用して Operator をビルドし、実行する方法は 3 つあります。

- 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。
- Operator をバンドルし、Operator Lifecycle Manager (OLM) を使用してクラスター上にデプロイします。

5.6.2.4.1. クラスター外でローカルに実行する。

Operator プロジェクトをクラスター外の Go プログラムとして実行できます。これは、デプロイメントとテストを迅速化するという開発目的において便利です。

手順

- 以下のコマンドを実行して、`~/kube/config` ファイルに設定されたクラスターにカスタムリソース定義 (CRD) をインストールし、Operator をローカルで実行します。

```
$ make install run
```

出力例

```
...
{"level":"info","ts":1612652419.9289865,"logger":"controller-runtime.metrics","msg":"metrics
server is starting to listen","addr":":8080"}
{"level":"info","ts":1612652419.9296563,"logger":"helm.controller","msg":"Watching
resource","apiVersion":"demo.example.com/v1","kind":"Nginx","namespace":"","reconcilePeriod
":"1m0s"}
{"level":"info","ts":1612652419.929983,"logger":"controller-runtime.manager","msg":"starting
metrics server","path":"/metrics"}
{"level":"info","ts":1612652419.930015,"logger":"controller-runtime.manager.controller.nginx-
controller","msg":"Starting EventSource","source":"kind source: demo.example.com/v1,
Kind=Nginx"}
{"level":"info","ts":1612652420.2307851,"logger":"controller-runtime.manager.controller.nginx-
controller","msg":"Starting Controller"}
{"level":"info","ts":1612652420.2309358,"logger":"controller-runtime.manager.controller.nginx-
controller","msg":"Starting workers","worker count":8}
```

5.6.2.4.2. クラスター上でのデプロイメントとしての実行

Operator プロジェクトは、クラスター上でのデプロイメントとして実行することができます。

手順

- 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を参照します。Quay.io などのリポジトリサイトにコンテナを保存するためのアカウントを取得できます。

- 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。

- イメージをリポジトリにプッシュします。

```
$ 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。

- 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。

- 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

5.6.2.4.3. Operator のバンドルおよび Operator Lifecycle Manager を使用したデプロイ

5.6.2.4.3.1. 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こと。
- OpenShift CLI (**oc**) v4.8+ がインストールされていること。
- Operator プロジェクトが Operator SDK を使用して初期化されていること。

手順

- 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を参照します。Quay.io などのリポジトリサイトにコンテナを保存するためのアカウントを取得できます。
 - 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>
```

5.6.2.4.3.2. 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 Container Platform にインストールされ、Kubernetes 拡張として実行されるため、追加のツールなしにすべての Operator のライフサイクル管理機能に Web コンソールおよび 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こと。
- ビルドされ、レジストリーにプッシュされる Operator バンドルイメージ。
- (OpenShift Container Platform 4.8 など、**apiextensions.k8s.io/v1** CRD を使用する場合は v1.16.0 以降の) Kubernetes ベースのクラスターに OLM がインストールされていること。
- **cluster-admin** パーミッションのあるアカウントを使用して **oc** でクラスターへログインしていること。

手順

1. 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 \
  [-n <namespace>] \ 1
  <registry>/<user>/<bundle_image_name>:<tag>
```

- 1** 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。

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。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オブジェクトすべてを作成して、Operator をクラスターにデプロイします。

5.6.2.5. 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には、OpenShift Container Platform で実行する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-6fd7c98d8-7dqr	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. 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>
```

5.6.2.6. 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。

5.6.3. 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。

5.6.3.1. 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。

5.6.4. Operator SDK での Helm サポート

5.6.4.1. 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一覧です。Helm CLI を使用してフラグ付きの値を設定する代わりに (例: **helm install -f values.yaml**)、これらをカスタムリソース (CR) 内で表現することができます。これにより、ネイティブ Kubernetes オブジェクトとして、適用される RBAC および監査証跡の利点を活用できます。

Tomcat という単純な CR の例:

```
apiVersion: apache.org/v1alpha1
kind: Tomcat
metadata:
  name: example-app
spec:
  replicaCount: 2
```


この場合の **replicaCount** 値、**2** は以下が使用されるチャートのテンプレートに伝播されます。

```
{{ .Values.replicaCount }}
```

Operator のビルドおよびデプロイ後に、CR の新規インスタンスを作成してアプリケーションの新規インスタンスをデプロイしたり、**oc** コマンドを使用してすべての環境で実行される異なるインスタンスを一覧表示したりすることができます。

```
$ oc get Tomcats --all-namespaces
```

Helm CLI を使用したり、Tiller をインストールしたりする必要はありません。Helm ベースの Operator はコードを Helm プロジェクトからインポートします。Operator のインスタンスを実行状態にし、カスタムリソース定義 (CRD) 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。

5.7. 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 機能进行处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しています。

5.7.1. 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については、[Operator のバンドル](#) を参照してください。

5.7.1.1. 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。

5.7.1.2. 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。

5.7.2. 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

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)。
spec.labels	Operator 内部で使われるキー/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 マニフェスト仕様がない複数のフィールドでは、ユーザーの入力が必要です。 <ul style="list-style-type: none"> ● 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

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)。

フィールド	説明
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](#)


5.7.2.1. 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) のメタデータで特定のアノテーションを手動で定義し、OperatorHub 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。

以下の表は、**metadata.annotations** フィールドを使用して、手動で定義できる Operator メタデータアノテーションを一覧表示しています。

表5.9 アノテーション

フィールド	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。

フィールド	説明
operatorframework.io/suggested-namespace	Operator をデプロイする必要がある推奨 namespace を設定します。
operators.openshift.io/infrastructure-features	<p>Operator によってサポートされるインフラストラクチャー機能。ユーザーは、Web コンソールで OperatorHub を使用して Operator を検出する際に、これらの機能で表示してフィルターを実行できます。有効で、大文字と小文字が区別される値は以下のとおりです。</p> <ul style="list-style-type: none"> ● 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 は基礎となるプラットフォームの FIPS モードを受け入れ、FIPS モードで起動されるノードで機能します。 <div>  <div> <p>重要</p> <p>FIPS 検証済み/進行中のモジュール (Modules in Process) 暗号ライブラリーの使用は、x86_64 アーキテクチャーの OpenShift Container Platform デプロイメントでのみサポートされています。</p> </div> </div> <ul style="list-style-type: none"> ● proxy-aware: Operator は、プロキシの背後にある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。

フィールド	説明
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一覧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。

使用例

Operator は非接続およびプロキシ対応をサポートします

```
operators.openshift.io/infrastructure-features: ['"disconnected", "proxy-aware"]'
```

Operator には OpenShift Container Platform ライセンスが必要です。

```
operators.openshift.io/valid-subscription: ['"OpenShift Container Platform"]'
```

Operator には 3scale ライセンスが必要です

```
operators.openshift.io/valid-subscription: ['"3Scale Commercial License", "Red Hat Managed Integration"]'
```

Operator は非接続およびプロキシ対応をサポートします。また、OpenShift Container Platform ライセンスが必要です。

```
operators.openshift.io/infrastructure-features: ['"disconnected", "proxy-aware"]'
operators.openshift.io/valid-subscription: ['"OpenShift Container Platform"]'
```

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](#)
- [ネットワークが制限された環境についての Operator の有効化\(非接続モード\)](#)
- [内部オブジェクトの非表示](#)
- [FIPS 暗号のサポート](#)

5.7.3. 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

- 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を一覧表示します。
 - 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。

CSV の要件については、Operator の作成者は以下の変更を加えることができます。

前提条件

- CSV を含む Operator プロジェクト

手順

1. Operator の CSV の 2 つの場所で関連するイメージへの SHA 参照を使用します。

- a. **spec.relatedImages** を更新します。

```
...
spec:
  relatedImages: ❶
    - name: etcd-operator ❷
      image: quay.io/etcd-
operator/operator@sha256:d134a9865524c29fcf75bbc4469013bc38d8a15cb5f41acfddeb6
b9e492f556e4 ❸
    - name: etcd-image
      image: quay.io/etcd-
operator/etcd@sha256:13348c15263bd8838ec1d5fc4550ede9860fcb0f843e48cbccec07
810eebb68
...
```

❶ **relatedImages** セクションを作成し、関連するイメージの一覧を設定します。

❷ イメージの一意の識別子を指定します。

❸ 各イメージを、イメージタグでなく、ダイジェスト (SHA) で指定します。

- b. Operator が使用する必要のあるイメージを挿入する環境変数を宣言する際に、デプロイメントの **env** セクションを更新します。

```
spec:
  install:
    spec:
      deployments:
        - name: etcd-operator-v3.1.1
          spec:
            replicas: 1
            selector:
              matchLabels:
                name: etcd-operator
```

```

strategy:
  type: Recreate
template:
  metadata:
    labels:
      name: etcd-operator
  spec:
    containers:
      - args:
        - /opt/etcd/bin/etcd_operator_run.sh
        env:
          - name: WATCH_NAMESPACE
            valueFrom:
              fieldRef:
                fieldPath: metadata.annotations['olm.targetNamespaces']
          - name: ETCD_OPERATOR_DEFAULT_ETCD_IMAGE ❶
            value: quay.io/etcd-
operator/etcd@sha256:13348c15263bd8838ec1d5fc4550ede9860fcbb0f843e48cbccec07
810eebb68 ❷
          - name: ETCD_LOG_LEVEL
            value: INFO
            image: quay.io/etcd-
operator/operator@sha256:d134a9865524c29fcf75bbc4469013bc38d8a15cb5f41acfddb6
b9e492f556e4 ❸
            imagePullPolicy: IfNotPresent
          livenessProbe:
            httpGet:
              path: /healthy
              port: 8080
              initialDelaySeconds: 10
              periodSeconds: 30
            name: etcd-operator
          readinessProbe:
            httpGet:
              path: /ready
              port: 8080
              initialDelaySeconds: 10
              periodSeconds: 30
          resources: {}
          serviceAccountName: etcd-operator
strategy: deployment

```

- ❶ 環境変数を使用して Operator によって参照されるイメージを挿入します。
- ❷ 各イメージを、イメージタグでなく、ダイジェスト (SHA) で指定します。
- ❸ また、イメージタグではなく、ダイジェスト (SHA) で Operator コンテナイメージを参照します。



注記

プローブの設定時に、**timeoutSeconds** 値は **periodSeconds** の値よりも低い値である必要があります。**timeoutSeconds** のデフォルト値は **1** です。**periodSeconds** のデフォルト値は **10** です。

2. **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。

5.7.4. 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) では、すべての Operator が Linux ホストで実行されることを前提としています。ただし、Operator の作成者は、ワーカーノードが OpenShift Container Platform クラスタ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 以外のバリエーションをサポートする場合、サポートされるバリエーションを一覧表示するために 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 ❶
  operatorframework.io/os.<os>: supported ❷
```

❶ **<arch>** をサポートされる文字列に設定します。

❷ **<os>** をサポートされる文字列に設定します。



注記

デフォルトチャンネルのチャン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ンネルで 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一覧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一覧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 ❶
  operatorframework.io/arch.amd64: supported ❷
```

- ❶ ❷ 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** バリエーションも明示的に組み込む必要があります。

関連情報

- マニフェストの一覧についての詳細は、[Image Manifest V 2, Schema 2](#) 仕様を参照してください。

5.7.4.1. 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 Container Platform の Operator Lifecycle Manager (OLM) でサポートされます。

表5.10 OpenShift Container Platform 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
64 ビット PowerPC little-endian	ppc64le
IBM Z	s390x

表5.11 OpenShift Container Platform 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

OpenShift Container Platform およびその他の Kubernetes ベースのディストリビューションの異なるバージョンは、アーキテクチャーおよびオペレーティングシステムの異なるセットをサポートする可能性があります。

5.7.5. 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、または特定の 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 Operator をクラスターに追加する場合、Web コンソールはインストールプロセス時にクラスター管理者に提案される 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。

5.7.6. 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ンネルを 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 がない場合、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 v1.8.0 は **operator-lib** v0.3.0 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 1.15

require (
    k8s.io/apimachinery v0.19.2
    k8s.io/client-go v0.19.2
    sigs.k8s.io/controller-runtime v0.7.0
    operator-framework/operator-lib v0.3.0
)
```

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](#)

5.7.7. 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 ❶
        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** プロパティは **false** または **nil** に設定される必要があります。

関連情報

- [Webhook 受付プラグインのタイプ](#)
- Kubernetes ドキュメント:
 - [検証用の受付 Webhook](#)
 - [変更用の受付 Webhook](#)
 - [変換 Webhook](#)

5.7.7.1. 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。

5.7.8. 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。

5.7.8.1. 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 ファイルに一覧表示される必要があります。

表5.12 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

フィールド	説明	必須/オプション
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
Resources	<p>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 セクションに一覧表示されます。</p> <p>この場合、オーケストレーションするすべての一覧ではなく、重要なオブジェクトのみを一覧表示することが推奨されます。たとえば、ユーザーが変更できない内部状態を保存する設定マップを一覧表示しないでください。</p>	オプション

フィールド	説明	必須/オプション
SpecDescriptors 、 StatusDescriptors 、および ActionDescriptors	<p>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が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。</p> <p>記述子には、3 つの種類があります。</p> <ul style="list-style-type: none"> ● 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。 <p>すべての記述子は以下のフィールドを受け入れます。</p> <ul style="list-style-type: none"> ● 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 Container Platform の正規の React UI X-Descriptor の一覧 については、openshift/console プロジェクトを参照してください。 <p>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。</p>	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.

```

5.7.8.2. 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ったら、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5.13 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

フィールド	説明	必須/オプション
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.
```

5.7.8.3. 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。

5.7.8.3.1. 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 ❶
      kind: cluster
      displayName: Cluster
```

- ❶ **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。

5.7.8.3.2. 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、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 ❶
    storage: true
```

- ❶ **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 ❶
  - name: v1beta1
    served: true
    storage: true ❷
```

- ❶ ❷ **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。

5.7.8.4. 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こと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 の一覧で設定されます (例: CRD 名および Kubernetes オブジェクトの対応する **metadata** および **spec**)。

以下の詳細の例では、**EtcdCluster**、**EtcdBackup** および **EtcdRestore** のテンプレートを示しています。

```
metadata:
  annotations:
    alm-examples: >-
      [{"apiVersion":"etcd.database.coreos.com/v1beta2","kind":"EtcdCluster","metadata":
{"name":"example","namespace":"default"},"spec":{"size":3,"version":"3.2.13"}},
{"apiVersion":"etcd.database.coreos.com/v1beta2","kind":"EtcdRestore","metadata":
{"name":"example-etcd-cluster"},"spec":{"etcdCluster":{"name":"example-etcd-
cluster"},"backupStorageType":"S3","s3":{"path":"<full-s3-path>","awsSecret":"<aws-secret>"}},
{"apiVersion":"etcd.database.coreos.com/v1beta2","kind":"EtcdBackup","metadata":
{"name":"example-etcd-cluster-backup"},"spec":{"etcdEndpoints":["<etcd-cluster-
endpoints>"],"storageType":"S3","s3":{"path":"<full-s3-path>","awsSecret":"<aws-secret>"}}]}
```

5.7.8.5. 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 Operator 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"] ❶
  ...
```

- ❶ 内部 CRD を文字列の配列として設定します。

5.7.8.6. 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、OpenShift Container Platform Web コンソールから Operator をインストールすると、ユーザーには 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ことを求めるプロンプトが出されます。

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": {}
      }
    ]
  }
}
...

```

5.7.9. 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。

5.7.9.1. 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一覧表示できます。

表5.14 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
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一覧表示されます。 この場合、オーケストレーションするすべての一覧ではなく、重要なオブジェクトのみを一覧表示することが推奨されます。たとえば、ユーザーが変更できない内部状態を保存する設定マップを一覧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

5.7.9.1.1. 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、または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。

5.7.9.1.2. 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。

5.7.9.2. 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の種類に依存できます。

表5.15 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

5.8. 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。

5.8.1. 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こと。
- OpenShift CLI (**oc**) v4.8+ がインストールされていること。
- Operator プロジェクトが Operator SDK を使用して初期化されていること。
- Operator が Go ベースの場合、プロジェクトを更新して OpenShift Container Platform での実行をサポートするイメージを使用する必要がある。

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を参照します。Quay.io などのリポジトリサイトにコンテナ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。

- 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。

- 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>
```

- バンドルイメージをプッシュします。

```
$ docker push <registry>/<user>/<bundle_image_name>:<tag>
```

5.8.2. 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 Container Platform にインストールされ、Kubernetes 拡張として実行されるため、追加のツールなしにすべての Operator のライフサイクル管理機能に Web コンソールおよび 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こと。
- ビルドされ、レジストリーにプッシュされる Operator バンドルイメージ。
- (OpenShift Container Platform 4.8 など、**apiextensions.k8s.io/v1** CRD を使用する場合は v1.16.0 以降の) Kubernetes ベースのクラスターに OLM がインストールされていること。
- **cluster-admin** パーミッションのあるアカウントを使用して **oc** でクラスターへログインしていること。
- Operator が Go ベースの場合、プロジェクトを更新して OpenShift Container Platform での実行をサポートするイメージを使用する必要がある。

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 \
  [-n <namespace>] \ 1
  <registry>/<user>/<bundle_image_name>:<tag>
```

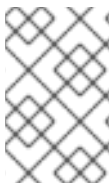
- 1 デフォルトで、このコマンドは `~/kube/config` ファイルの現在アクティブなプロジェクトに Operator をインストールします。`-n` フラグを追加して、インストールに異なる namespace スコープを設定できます。

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。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オブジェクトすべてを作成して、Operator をクラスターにデプロイします。

5.8.3. 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一覧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こと。
- ビルドされ、レジストリーにプッシュされる Operator バンドルイメージ。
- (OpenShift Container Platform 4.8 など、**apiextensions.k8s.io/v1** CRD を使用する場合は v1.16.0 以降の) Kubernetes ベースのクラスターに OLM がインストールされていること。
- **cluster-admin** パーミッションのあるアカウントを使用して **oc** でクラスターへログインしていること。

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を参照します。Quay.io などのリポジトリサイトにコンテナ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: default
spec:
  displayName: My Test
  publisher: Company
  sourceType: grpc
  image: quay.io/example/memcached-catalog:v0.0.1 ❶
  updateStrategy:
    registryPoll:
      interval: 10m
```

- ❶ **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: default
spec:
  targetNamespaces:
    - default
```

- 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: default
spec:
  channel: "alpha"
  installPlanApproval: Manual
  name: catalog
  source: cs-memcached
  sourceNamespace: default
  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。

5.8.4. 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、または従来の 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

- 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[0009] Successfully created registry pod: quay-io-demo-memcached-operator-v0-0-1
INFO[0009] Created CatalogSource: memcached-operator-catalog
INFO[0010] OperatorGroup "operator-sdk-og" created
INFO[0010] Created Subscription: memcached-operator-v0-0-1-sub
INFO[0013] Approved InstallPlan install-bqggr for the Subscription: memcached-operator-v0-0-1-sub
INFO[0013] Waiting for ClusterServiceVersion "my-project/memcached-operator.v0.0.1" to reach 'Succeeded' phase
INFO[0013] Waiting for ClusterServiceVersion "my-project/memcached-operator.v0.0.1" to appear
INFO[0019] Found ClusterServiceVersion "my-project/memcached-operator.v0.0.1" phase: Succeeded
```

- 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[0009] Successfully 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[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"
```

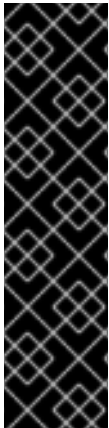
- インストールされた Operator のクリーンアップ

```
$ 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](#)

5.8.5. OpenShift Container Platform バージョンとの Operator 互換性の制御



重要

Kubernetes は、今後のリリースで削除される特定の API を定期的に非推奨にします。Operator が非推奨の API を使用している場合、OpenShift Container Platform 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した後に機能しない可能性があります。

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。理想的には、OpenShift Container Platform の今後のバージョンでは Operator の互換性が失われるので今後のバージョンがリリースされる前に Operator を更新することをお勧めします。

API が OpenShift Container Platform バージョンから削除されると、削除された API を依然として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 が適切に機能しなくなります。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

OpenShift Container Platform 4.8 以降で実行されている Operator のイベントアラートをチェックして、現在使用中の API に関する警告があるかどうかを確認できます。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

今後の OpenShift Container Platform リリースで削除される API。

APIRemovedInNextEUSReleaseInUse

次の OpenShift Container Platform [Extended Update Support](#) (EUS) リリースで削除される API。

クラスター管理者が Operator をインストールしている場合に、OpenShift Container Platform の次のバージョンにアップグレードする前に、そのクラスターのバージョンと互換性がある 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。Operator プロジェクトを更新して非推奨または削除済みの API を使用しないようにすることが推奨されますが、OpenShift Container Platform の以前のバージョンを引き続き使用して削除済みの API で Operator バンドルを公開する必要がある場合には、バンドルが正しく設定されていることを確認します。

以下の手順では、管理者が互換性のないバージョンの OpenShift Container Platform に Operator をインストールできないようにするのに役立ちます。これらの手順では、管理者が、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 OpenShift Container Platform のバージョンにアップグレードできないようにします。

この手順は、Operator の現行バージョンが、何らかの理由で特定の OpenShift Container Platform バージョンで適切に機能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atalog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 は、クラスター管理者が API がサポートされなくなった OpenShift Container Platform の将来のバージョンにアップグレードする際に、重大なワークロードに悪影響を及ぼす可能性があります。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 の特定のバンドルは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 Container Platform で正常に機能しない場合は、Operator と互換性のある OpenShift Container Platform の最大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、クラスター管理者は 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 Container Platform の最大クラスターバージョンを指定します。たとえば、**value** を **4.8** 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、クラスターが OpenShift Container Platform 4.8 より後のバージョンにアップグレードされなくなります。

2. バンドルが Red Hat 提供の Operator カタログでのディストリビューション向けの場合には、以下のプロパティを設定して、Operator の OpenShift Container Platform を互換性のあるバージョンに設定します。この設定では、Operator は互換性のある OpenShift Container Platform のバージョンを対象とするカタログにだけ含まれます。



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: "v4.6-v4.8" 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 Container Platform に引き継がれないようにするには、Operator 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: **v4.6-v4.8**)。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](#)

5.8.6. 関連情報

- Bundle Format 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。

5.9. 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 Container Platform などの設定済みの Kubernetes クラスターへのアクセスと共に実行されることを前提とします。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. プロキシコンテナを 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:v1.8.0
    entrypoint:
    - scorecard-test
    - basic-check-spec
    labels:
      suite: basic
      test: basic-check-spec-test
  - image: quay.io/operator-framework/scorecard-test:v1.8.0
    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5.16 基本的なテストスイート

テスト	説明	短縮名
-----	----	-----

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5.17 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一覧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一覧表示します。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 に一覧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 に一覧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5.29 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:v1.8.0",
        "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5.30 テキスト出力スニペットの例

```
-----
Image:    quay.io/operator-framework/scorecard-test:v1.8.0
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、テストの一覧と設定可能な **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 ❶
  tests:
  - entrypoint:
    - scorecard-test
    - basic-check-spec
    image: quay.io/operator-framework/scorecard-test:v1.8.0
    labels:
      suite: basic
      test: basic-check-spec-test
  - entrypoint:
    - scorecard-test
    - olm-bundle-validation
    image: quay.io/operator-framework/scorecard-test:v1.8.0
    labels:
      suite: olm
      test: olm-bundle-validation-test
```

- ❶ 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5.31 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. PROMETHEUS による組み込みモニタリングの設定

以下では、Prometheus Operator を使用して Operator SDK によって提供されるビルトインされたモニタリングサポートについて説明し、Operator 作成者がどのように使用できるかについて詳しく説明します。

5.10.1. Prometheus Operator のサポート

Prometheus はオープンソースのシステムモニタリングおよびアラートツールキットです。Prometheus Operator は、OpenShift Container Platform などの Kubernetes ベースのクラスターで実行される Prometheus クラスターを作成し、設定し、管理します。

ヘルパー関数は、デフォルトで Operator SDK に存在し、Prometheus Operator がデプロイされているクラスターで使用できるように生成された Go ベースの Operator にメトリクスを自動的にセットアップします。

5.10.2. メトリクスヘルパー

Operator SDK を使用して生成される Go ベース Operator では、以下の関数が実行中のプログラムについての一般的なメトリクスを公開します。

```
func ExposeMetricsPort(ctx context.Context, port int32) (*v1.Service, error)
```

これらのメトリクスは **controller-runtime** ライブラリー API から継承されます。メトリクスはデフォルトで **0.0.0.0:8383/metrics** で提供されます。

Service オブジェクトは、メトリクスポートが公開された状態で作成されます。これはその後 Prometheus によってアクセスされます。**Service** オブジェクトは、リーダー Pod の **root** オーナーが削除されるとガベージコレクションの対象になります。

以下のサンプルは、Operator SDK を使用して生成されるすべての Operator の **cmd/manager/main.go** ファイルにあります。

```
import(
    "github.com/operator-framework/operator-sdk/pkg/metrics"
    "machine.openshift.io/controller-runtime/pkg/manager"
)

var (
    // Change the below variables to serve metrics on a different host or port.
    metricsHost    = "0.0.0.0" ❶
    metricsPort int32 = 8383 ❷
)
...
func main() {
    ...
    // Pass metrics address to controller-runtime manager
    mgr, err := manager.New(cfg, manager.Options{
        Namespace:      namespace,
        MetricsBindAddress: fmt.Sprintf("%s:%d", metricsHost, metricsPort),
    })

    ...
    // Create Service object to expose the metrics port.
    _, err = metrics.ExposeMetricsPort(ctx, metricsPort)
    if err != nil {
        // handle error
        log.Info(err.Error())
    }
    ...
}
```

❶ メトリクスの公開に使用されるホスト。

❷ メトリクスの公開に使用されるポート。

5.10.2.1. メトリクスポートの変更

Operator の作成者は、メトリクスが公開されるポートを変更できます。

前提条件

- Operator SDK を使用して生成される Go ベースの Operator
- Prometheus Operator がデプロイされた Kubernetes ベースのクラスター

手順

- 生成された Operator の **cmd/manager/main.go** ファイルで、以下の行の **metricsPort** の値を変更します。

```
var metricsPort int32 = 8383
```

5.10.3. サービスモニター

ServiceMonitor は、Prometheus Operator によって提供されるカスタマリソースであり、**Service** オブジェクトで **Endpoints** を検出し、Prometheus がこれらの Pod を監視するように設定します。

Operator SDK を使用して生成される Go ベースの Operator では、**GenerateServiceMonitor()** ヘルパー関数は **Service** オブジェクトを取り、これに基づいて **ServiceMonitor** オブジェクトを生成することができます。

関連情報

- **ServiceMonitor** カスタムリソース定義 (CRD) についての詳細は、[Prometheus Operator ドキュメント](#) を参照してください。

5.10.3.1. サービスモニターの作成

Operator の作成者は、新規に作成されるサービスを受け入れる **metrics.CreateServiceMonitor()** ヘルパー関数を使用して、作成されたモニターリングサービスのサービスターゲット検出を追加できます。

前提条件

- Operator SDK を使用して生成される Go ベースの Operator
- Prometheus Operator がデプロイされた Kubernetes ベースのクラスター

手順

- **metrics.CreateServiceMonitor()** ヘルパー関数を Operator コードに追加します。

```
import(
    "k8s.io/api/core/v1"
    "github.com/operator-framework/operator-sdk/pkg/metrics"
    "machine.openshift.io/controller-runtime/pkg/client/config"
)
func main() {
    ...
    // Populate below with the Service(s) for which you want to create ServiceMonitors.
    services := []*v1.Service{}
    // Create one ServiceMonitor per application per namespace.
    // Change the below value to name of the Namespace you want the ServiceMonitor to be
    // created in.
    ns := "default"
```

```
// restConfig is used for talking to the Kubernetes apiserver
restConfig := config.GetConfig()

// Pass the Service(s) to the helper function, which in turn returns the array of
// ServiceMonitor objects.
serviceMonitors, err := metrics.CreateServiceMonitors(restConfig, ns, services)
if err != nil {
    // Handle errors here.
}
...
}
```

5.11. 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にある場合、**pod-eviction-timeout** はリーダー Pod がノードから削除され、リーダーシップを中止するまでの時間を判別します (デフォルトは **5m**)。詳細は、[Leader-for-life Go ドキュメント](#)を参照してください。

Leader-with-lease

リーダー Pod は定期的にリーダーリースを更新し、リースを更新できない場合にリーダーシップを放棄します。この実装により、既存リーダーが分離される場合に新規リーダーへの迅速な移行が可能になりますが、スプリットブレインが **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。

5.11.1. 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。

5.11.1.1. 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。

5.11.1.2. 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。

5.12. 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 Container Platform 4.8 以降で削除されます。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 Container Platform 4.6 以降の Operator Lifecycle Manager (OLM) の推奨されるパッケージ形式です。

5.12.1. 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を取得し、入力ディレクトリ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](#)

5.12.2. 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こと。
- 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> \ ❶
[--output-dir <directory>] \ ❷
--image-tag-base <image_name_base> ❸
```

- ❶ **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。
- ❷ 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。
- ❸ **--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"
```

5.13. 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 Container Platform など) へのクラスター管理者のアクセスのある Operator の作成者は、Operator SDK CLI を使用して Go、Ansible、または Helm をベースに独自の 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。

5.13.1. 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。

5.13.1.1. validate

bundle validate サブコマンドは Operator バンドルを検証します。

表5.18 **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展開するためのツール。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一覧表示します。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一覧表示します。

5.13.2. 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5.19 **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。

フラグ	説明
--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。

5.13.3. 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5.20 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5.21 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
```

5.13.4. create

operator-sdk create コマンドは、Kubernetes API の作成または **スキャフオールディング** に使用されます。

5.13.4.1. api

create api サブコマンドは Kubernetes API をスキャフオ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5.22 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。

5.13.5. 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。

5.13.5.1. 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する入力された [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5.23 generate bundle フラグ

フラグ	説明
--channels (文字列)	バンドルが属するチャン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)	CustomResourceDefinition 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ンネル。
--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。

フラグ	説明
--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。
--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、または 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。

5.13.5.2. 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。

5.13.5.2.1. 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5.24 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。
-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。
--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。

フラグ	説明
--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。

5.13.6. 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 メトリック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5.25 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。

5.13.7. 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。

5.13.7.1. 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5.26 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。
-h, --help	run bundle サブコマンドのヘルプ出力。

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。

5.13.7.2. 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5.27 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。
-h, --help	run bundle サブコマンドのヘルプ出力。

5.13.8. 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5.28 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一覧表示します。
-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。
-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。

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。

第6章 クラスター OPERATOR のリファレンス

このリファレンスガイドは、OpenShift Container Platform のアーキテクチャー基盤として機能する、Red Hat が出荷する **クラスター Operator** のインデックスを作成します。クラスター Operator は、特に明記されていない限り、デフォルトでインストールされ、Cluster Version Operator (CVO) により管理されます。コントロールプレーンアーキテクチャーの詳細は[OpenShift Container Platform の Operator](#) を参照してください。

クラスター管理者は、OpenShift Container Platform Web コンソールの **Administration → Cluster Settings** ページからクラスター Operator を表示できます。



注記

クラスター Operator は、Operator Lifecycle Manager (OLM) および Operator Hub では管理されていません。OLM と Operator Hub は、[Operator Framework](#) の一部で、オプションの[アドオン Operator](#) のインストールおよび実行時に OpenShift Container Platform で使用されます。

6.1. CLOUD CREDENTIAL OPERATOR

目的

Cloud Credential Operator (CCO) は、クラウドプロバイダーの認証情報を Kubernetes カスタムリソース定義 (CRD) として管理します。CCO は **CredentialsRequest** カスタムリソース (CR) で同期し、OpenShift Container Platform コンポーネントが、クラスターの実行に必要な特定のパーミッションと共にクラウドプロバイダーの認証情報を要求できるようにします。

install-config.yaml ファイルで **credentialsMode** パラメーターに異なる値を設定すると、CCO は複数の異なるモードで動作するように設定できます。モードが指定されていない場合や、**credentialsMode** パラメーターが空の文字列 ("") に設定されている場合は、CCO はデフォルトモードで動作します。

プロジェクト

[openshift-cloud-credential-operator](#)

CRD

- **credentialsrequests.cloudcredential.openshift.io**
 - スコープ: Namespaced
 - CR: **CredentialsRequest**
 - 検証: Yes

設定オブジェクト

必要な設定はありません。

関連情報

- [CredentialsRequest カスタムリソース](#)
- [Cloud Credential Operator について](#)

6.2. CLUSTER AUTHENTICATION OPERATOR

目的

Cluster Authentication Operator は、クラスター内に **Authentication** カスタムリソースをインストールし、維持します。これは、以下を使用して表示できます。

```
$ oc get clusteroperator authentication -o yaml
```

プロジェクト

[cluster-authentication-operator](#)

6.3. CLUSTER AUTOSCALER OPERATOR

目的

Cluster Autoscaler Operator は **cluster-api** プロバイダーを使用して OpenShift Cluster Autoscaler のデプロイメントを管理します。

プロジェクト

[cluster-autoscaler-operator](#)

CRD

- **ClusterAutoscaler**: これは、クラスターの Autoscaler インスタンスの設定を制御するシングルトリソースです。Operator は、管理された namespace の **default** という名前の **ClusterAutoscaler** リソース (**WATCH_NAMESPACE** 環境変数の値) のみに応答します。
- **MachineAutoscaler**: このリソースはノードグループを対象にし、アノテーションを管理してグループの自動スケーリングを有効にし、設定します (**min** および **max** サイズ)。現時点では、**MachineSet** オブジェクトのみをターゲットにすることができます。

6.4. CLUSTER CONFIG OPERATOR

目的

Cluster Config Operator は、**config.openshift.io** に関連する以下のタスクを実行します。

- CRD を作成する。
- 最初のカスタムリソースをレンダリングする。
- 移行を処理する。

プロジェクト

[cluster-config-operator](#)

6.5. CLUSTER CSI SNAPSHOT CONTROLLER OPERATOR

目的

Cluster CSI Snapshot Controller Operator は、CSI Snapshot Controller をインストールし、維持します。CSI Snapshot Controller は **VolumeSnapshot** CRD オブジェクトを監視し、ボリュームスナップショットの作成および削除のライフサイクルを管理します。

プロジェクト

[cluster-csi-snapshot-controller-operator](#)

6.6. CLUSTER IMAGE REGISTRY OPERATOR

目的

Cluster Image Registry Operator は、OpenShift Container Platform レジストリーのシングルトンインスタンスを管理します。ストレージの作成を含む、レジストリーのすべての設定を管理します。

初回起動時に、Operator はクラスターで検出される設定に基づいてデフォルトの **image-registry** リソースインスタンスを作成します。これは、クラウドプロバイダーに基づいて使用するクラウドストレージのタイプを示します。

完全な **image-registry** リソースを定義するのに利用できる十分な情報がない場合、その不完全なリソースが定義され、Operator は足りない情報を示す情報を使ってリソースのステータスを更新します。

Cluster Image Registry Operator は **openshift-image-registry** namespace で実行され、その場所のレジストリーインスタンスも管理します。レジストリーのすべての設定およびワークロードリソースはその namespace に置かれます。

プロジェクト

[cluster-image-registry-operator](#)

6.7. CLUSTER MACHINE APPROVER OPERATOR

目的

Cluster Machine Approver Operator は、クラスターのインストール後に、新規ワーカーノードに要求された CSR を自動承認します。



注記

コントロールプレーンノードの場合に、ブートストラップノードの **approve-csr** サービスは、クラスターのブートストラップフェーズ時にすべての CSR を自動的に承認します。

プロジェクト

[cluster-machine-approver-operator](#)

6.8. クラスターモニタリング OPERATOR

目的

Cluster Monitoring Operator は、OpenShift Container Platform の上部にデプロイされた Prometheus ベースのクラスターモニタリングスタックを管理し、更新します。

プロジェクト

[openshift-monitoring](#)

CRD

- **alertmanagers.monitoring.coreos.com**
 - スコープ: Namespaced
 - CR: **alertmanager**
 - 検証: Yes
- **prometheuses.monitoring.coreos.com**
 - スコープ: Namespaced

- CR: **prometheus**
- 検証: Yes
- **prometheusrules.monitoring.coreos.com**
 - スコープ: Namespaced
 - CR: **prometheusrule**
 - 検証: Yes
- **servicemonitors.monitoring.coreos.com**
 - スコープ: Namespaced
 - CR: **servicemonitor**
 - 検証: Yes

設定オブジェクト

```
$ oc -n openshift-monitoring edit cm cluster-monitoring-config
```

6.9. CLUSTER NETWORK OPERATOR

目的

Cluster Network Operator は、OpenShift Container Platform クラスターでネットワークコンポーネントをインストールし、アップグレードします。

6.10. OPENSIFT CONTROLLER MANAGER OPERATOR

目的

OpenShift Controller Manager Operator は **OpenShiftControllerManager** カスタムリソースをクラスターにインストールし、これを維持します。これは、以下で表示できます。

```
$ oc get clusteroperator openshift-controller-manager -o yaml
```

カスタムリソース定義 (CRD) **openshiftcontrollermanagers.operator.openshift.io** は以下を使用してクラスターで確認できます。

```
$ oc get crd openshiftcontrollermanagers.operator.openshift.io -o yaml
```

プロジェクト

[cluster-openshift-controller-manager-operator](#)

6.11. CLUSTER SAMPLES OPERATOR

目的

Cluster Samples Operator は、**openshift** namespace に保存されるサンプルイメージストリームおよびテンプレートを管理します。

初回起動時に、Operator はデフォルトのサンプル設定リソースを作成し、イメージストリームおよびテンプレートの作成を開始します。設定オブジェクトは、キーが **cluster** で、タイプが **configs.samples** のクラスタースコープのオブジェクトです。

イメージストリームは、**registry.redhat.io** のイメージを参照する Red Hat Enterprise Linux CoreOS (RHCOS) ベースの OpenShift Container Platform イメージストリームです。同様に、テンプレートは OpenShift Container Platform テンプレートとして分類されます。

Cluster Samples Operator デプロイメントは **openshift-cluster-samples-operator** namespace 内に含まれます。起動時に、インストールプルシークレットは内部レジストリーおよび API サーバーのイメージストリームのインポートロジックによって使用され、**registry.redhat.io** で認証されます。管理者は、サンプルイメージストリームに使用されるレジストリーを変更する場合、追加のシークレットを **openshift** namespace に作成できます。これらのシークレットが作成される場合、これらには、イメージのインポートを容易にするために必要な **docker** の **config.json** のコンテンツが含まれます。

Cluster Samples Operator のイメージには、関連付けられた OpenShift Container Platform リリースのイメージストリームおよびテンプレートの定義が含まれます。Cluster Samples Operator がサンプルを作成した後に、互換性のある OpenShift Container Platform バージョンを示すアノテーションを追加します。Operator はこのアノテーションを使用して、各サンプルを互換性のあるリリースバージョンに一致させるようにします。このインベントリーの外にあるサンプルは省略されるサンプルであるために無視されます。

Operator によって管理されるサンプルへの変更は、バージョンのアノテーションが変更または削除されない限り許可されます。ただし、アップグレード時に、バージョンアノテーションが変更されると、サンプルが新しいバージョンで更新されるため、これらの変更は置き換えられる可能性があります。jenkins イメージはインストールからのイメージペイロードの一部であり、イメージストリームに直接タグ付けされます。

Samples Operator 設定リソースには、削除時に以下を消去するファイナライザーが含まれます。

- Operator 管理のイメージストリーム
- Operator 管理のテンプレート
- Operator が生成する設定リソース
- クラスターステータスのリソース

サンプルリソースの削除時に、Cluster Samples Operator はデフォルト設定を使用してリソースを再作成します。

プロジェクト

[cluster-samples-operator](#)

6.12. CLUSTER STORAGE OPERATOR

目的

Cluster Storage Operator は OpenShift Container Platform のクラスター全体のストレージのデフォルト値を設定します。これにより、OpenShift Container Platform クラスターのデフォルトのストレージクラスの存在を確認できます。

プロジェクト

[cluster-storage-operator](#)

設定

必要な設定はありません。

注記

- Cluster Storage Operator は Amazon Web Services (AWS) および Red Hat OpenStack Platform (RHOSP) をサポートします。
- 作成されたストレージクラスは、そのアノテーションを編集してデフォルト以外にすることができますが、ストレージクラスは Operator が実行される限り削除できません。

6.13. CLUSTER VERSION OPERATOR

目的

Cluster Operator は、クラスター機能の特定の領域を管理します。Cluster Version Operator (CVO) はクラスター Operator のライフサイクルを管理し、その多くはデフォルトで OpenShift Container Platform にインストールされます。

また、CVO は OpenShift Update Service をチェックして、現在のコンポーネントのバージョンとグラフの情報に基づいて、有効な更新と更新パスを確認します。

プロジェクト

[cluster-version-operator](#)

関連情報

- [OpenShift Container Platform の Operator](#)

6.14. CONSOLE OPERATOR

目的

Console Operator は OpenShift Container Platform Web コンソールをクラスターにインストールし、維持します。

プロジェクト

[console-operator](#)

6.15. DNS OPERATOR

目的

DNS Operator は、Pod に対して名前解決サービスを提供するために CoreDNS をデプロイし、これを管理し、OpenShift Container Platform での DNS ベースの Kubernetes サービス検出を可能にします。

Operator は、クラスターの設定に基づいて作業用のデフォルトデプロイメントを作成します。

- デフォルトのクラスタードメインは **cluster.local** です。
- CoreDNS Corefile または Kubernetes プラグインの設定はサポートされていません。

DNS Operator は、静的 IP を持つサービスとして公開される Kubernetes デモンセットとして CoreDNS を管理します。CoreDNS は、クラスター内のすべてのノードで実行されます。

プロジェクト

[cluster-dns-operator](#)

6.16. ETCD CLUSTER OPERATOR

目的

etcd cluster Operator は etcd クラスターのスケーリングを自動化し、etcd モニタリングおよびメトリックを有効にし、障害復旧手順を単純化します。

プロジェクト

[cluster-etcd-operator](#)

CRD

- **etcds.operator.openshift.io**
 - スcope: Cluster
 - CR: **etcd**
 - 検証: Yes

設定オブジェクト

```
$ oc edit etcd cluster
```

6.17. INGRESS OPERATOR

目的

Ingress Operator は OpenShift Container Platform ルーターを設定し、管理します。

プロジェクト

[openshift-ingress-operator](#)

CRD

- **clusteringresses.ingress.openshift.io**
 - スcope: Namespaced
 - CR: **clusteringresses**
 - 検証: No

設定オブジェクト

- クラスター設定
 - タイプ名: **clusteringresses.ingress.openshift.io**
 - インスタンス名: **default**
 - コマンドの表示:

```
$ oc get clusteringresses.ingress.openshift.io -n openshift-ingress-operator default -o yaml
```

注記

Ingress Operator はルーターを **openshift-ingress** プロジェクトに設定し、ルーターのデプロイメントを作成します。


```
$ oc get deployment -n openshift-ingress
```

Ingress Operator は、**network/cluster** ステータスの **clusterNetwork[].cidr** を使用して、管理 Ingress コントローラー (ルーター) が動作するモード (IPv4、IPv6、またはデュアルスタック) を判別します。たとえば、**clusterNetwork** に v6 **cidr** のみが含まれる場合、Ingress コントローラーは v6 専用モードで動作します。

以下の例では、Ingress Operator によって管理される Ingress コントローラーは、1つのクラスターネットワークのみが存在し、ネットワークが IPv4 **cidr** であるために IPv4 専用モードで実行されます。

```
$ oc get network/cluster -o jsonpath='{.status.clusterNetwork[*]}'
```

出力例

```
map[cidr:10.128.0.0/14 hostPrefix:23]
```

6.18. INSIGHTS OPERATOR

目的

Insights Operator は OpenShift Container Platform 設定データを収集し、これを Red Hat に送信します。このデータは、クラスターで発生する可能性のある問題について、今後を見据えた上で、事前に対応できる内容に関して推奨事項を生み出します。これらの今後の対応案については、console.redhat.com の Insights Advisor を介してクラスター管理者に伝達されます。

プロジェクト

[insights-operator](#)

設定

必要な設定はありません。

注記

Insights Operator は、OpenShift Container Platform Telemetry を補完します。

関連情報

- Insights Operator と Telemetry の詳細は、[About remote health monitoring](#) を参照してください。

6.19. KUBERNETES API SERVER OPERATOR

目的

Kubernetes API Server Operator は、OpenShift Container Platform の上部にデプロイされた Kubernetes API サーバーを管理し、更新します。Operator は OpenShift Container Platform の **library-go** フレームワークをベースとしており、Cluster Version Operator (CVO) でインストールされます。

プロジェクト

[openshift-kube-apiserver-operator](#)

CRD

- kubeapiservers.operator.openshift.io**
 - スコープ: Cluster

- CR: **kubeapiserver**
- 検証: Yes

設定オブジェクト

```
$ oc edit kubeapiserver
```

6.20. KUBERNETES CONTROLLER MANAGER OPERATOR

目的

Kubernetes Controller Manager Operator は、OpenShift Container Platform にデプロイされた Kubernetes Controller Manager を管理し、更新します。Operator は OpenShift Container Platform の **library-go** フレームワークをベースとしており、Cluster Version Operator (CVO) でインストールされます。

これには、以下のコンポーネントが含まれます。

- Operator
- ブートストラップマニフェストレンダラー
- 静的 Pod をベースとするインストーラー
- 設定オブザーバー

デフォルトで、Operator は **metrics** サービス経由で Prometheus メトリックを公開します。

プロジェクト

[cluster-kube-controller-manager-operator](#)

6.21. KUBERNETES SCHEDULER OPERATOR

目的

Kubernetes Scheduler Operator は、OpenShift Container Platform の上部にデプロイされる Kubernetes スケジューラーを管理し、更新します。Operator は OpenShift Container Platform の **library-go** フレームワークをベースとしており、Cluster Version Operator (CVO) でインストールされます。

Kubernetes Scheduler Operator には以下のコンポーネントが含まれます。

- Operator
- ブートストラップマニフェストレンダラー
- 静的 Pod をベースとするインストーラー
- 設定オブザーバー

デフォルトで、Operator はメトリックサービス経由で Prometheus メトリックを公開します。

プロジェクト

[cluster-kube-scheduler-operator](#)

設定

Kubernetes Scheduler の設定はマージの結果になります。

- デフォルト設定。
- 仕様 [schedulers.config.openshift.io](https://kubernetes.io/docs/reference/config-api/schedulers.config.openshift.io/) からの観察される設定。

これらはすべてスパースな設定であり、最後に有効な設定を形成するためにマージされる無効にされた JSON スニペットです。

6.22. KUBERNETES STORAGE VERSION MIGRATOR OPERATOR

目的

Kubernetes Storage Version Migrator Operator はデフォルトのストレージバージョンの変更を検出し、ストレージバージョンの変更時にリソースタイプの移行要求を作成し、移行要求を処理します。

プロジェクト

[cluster-kube-storage-version-migrator-operator](#)

6.23. MACHINE API OPERATOR

目的

Machine API Operator は、Kubernetes API を拡張する特定の目的のカスタムリソース定義 (CRD)、コントローラー、および RBAC オブジェクトのライフサイクルを管理します。これにより、クラスター内のマシンの必要な状態が宣言されます。

プロジェクト

[machine-api-operator](#)

CRD

- **MachineSet**
- **Machine**
- **MachineHealthCheck**

6.24. MACHINE CONFIG OPERATOR

目的

Machine Config Operator は、カーネルと kubelet 間のすべてのものを含め、ベースオペレーティングシステムおよびコンテナランタイムの設定および更新を管理し、適用します。

以下の 4 つのコンポーネントがあります。

- **machine-config-server**: クラスターに参加する新規マシンに Ignition 設定を提供します。
- **machine-config-controller**: マシンのアップグレードを **MachineConfig** オブジェクトで定義される必要な設定に調整します。マシンセットのアップグレードを個別に制御するオプションが提供されます。
- **machine-config-daemon**: 更新時に新規のマシン設定を適用します。マシンの状態を要求されたマシン設定に対して検証し、確認します。
- **machine-config**: インストール時のマシン設定の完全なソース、初回の起動、およびマシンの更新を提供します。

プロジェクト

[openshift-machine-config-operator](#)

6.25. MARKETPLACE OPERATOR

目的

Marketplace Operator はクラスター外の Operator をクラスターに入れるための経路です。

プロジェクト

[operator-marketplace](#)

6.26. NODE TUNING OPERATOR

目的

Node Tuning Operator は、TuneD デーモンのオーケストレーションによるノードレベルのチューニングの管理に役立ちます。ほとんどの高パフォーマンスアプリケーションでは、一定レベルのカーネルのチューニングが必要です。Node Tuning Operator は、ノードレベルの `sysctl` の統一された管理インターフェイスをユーザーに提供し、ユーザーが指定するカスタムチューニングを追加できるよう柔軟性を提供します。

Operator は、コンテナ化された OpenShift Container Platform の TuneD デーモンを Kubernetes デーモンセットとして管理します。これにより、カスタムチューニング仕様が、デーモンが認識する形式でクラスターで実行されるすべてのコンテナ化された TuneD デーモンに渡されます。デーモンは、ノードごとに1つずつ、クラスターのすべてのノードで実行されます。

コンテナ化された TuneD デーモンによって適用されるノードレベルの設定は、プロファイルの変更をトリガーするイベントで、または終了シグナルの受信および処理によってコンテナ化された TuneD デーモンが正常に終了する際にロールバックされます。

Node Tuning Operator は、バージョン 4.1 以降における標準的な OpenShift Container Platform インストールの一部となっています。

プロジェクト

[cluster-node-tuning-operator](#)

6.27. OPENSIFT API SERVER OPERATOR

目的

OpenShift API Server Operator は、クラスターに **openshift-apiserver** をインストールし、維持します。

プロジェクト

[openshift-apiserver-operator](#)

CRD

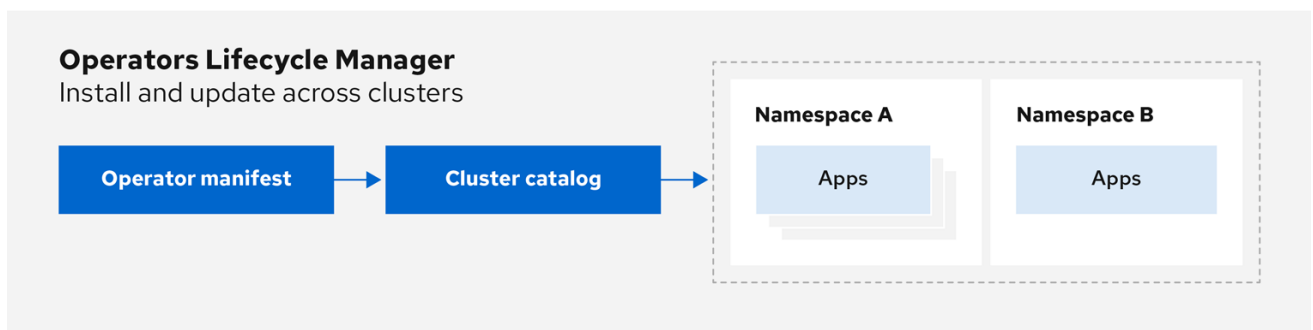
- **openshiftapiservers.operator.openshift.io**
 - スcope: Cluster
 - CR: **openshiftapiserver**
 - 検証: Yes

6.28. OPERATOR LIFECYCLE MANAGER OPERATOR

目的

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 Container Platform 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6.1 Operator Lifecycle Manager ワークフロー



OpenShift_43_1019

OLM は OpenShift Container Platform 4.8 でデフォルトで実行されます。これは、クラスター管理者がクラスターで実行されている Operator をインストールし、アップグレードし、アクセスをこれに付与するのに役立ちます。OpenShift Container Platform Web コンソールでは、クラスター管理者が Operator をインストールし、特定のプロジェクトアクセスを付与して、クラスターで利用可能な Operator のカタログを使用するための管理画面を利用できます。

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、またモニターリング、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。

CRD

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6.1 OLM およびカタログ Operator で管理される CRD

リソース	短縮名	所有する Operator	説明
ClusterServiceVersion (CSV)	csv	OLM	アプリケーションのメタデータ: 名前、バージョン、アイコン、必須リソース、インストールなど。
InstallPlan	ip	カタログ	CSV を自動的にインストールするか、またはアップグレードするために作成されるリソースの計算された一覧。

リソース	短縮名	所有する Operator	説明
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。
サブスクリプション	sub	カタログ	パッケージのチャンネルを追跡して 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 の一覧またはクラスター全体でカスタムリソース (CR) を監視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6.2 OLM およびカタログ Operator によって作成されるリソース

リソース	所有する Operator
Deployments	OLM
ServiceAccounts	
(Cluster)Role	
(Cluster)RoleBinding	
CustomResourceDefinitions (CRDs)	カタログ
ClusterServiceVersions	

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。

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ン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。

チャン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、チャンネル、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。更新が見つかったら、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. 管理対象または必須の CRD のそれぞれについて、これを解決済みリソースとして追加します。
 - 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atalog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。

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ン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、チャンネルは特定の 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 をチャン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)。

関連情報

- [Operator Lifecycle Manager \(OLM\) について](#)

6.29. OPENSIFT SERVICE CA OPERATOR

目的

OpenShift Service CA Operator は、Kubernetes サービスへの証明書を作成し、提供を管理します。

プロジェクト

[openshift-service-ca-operator](#)

6.30. VSPHERE PROBLEM DETECTOR OPERATOR

目的

vSphere Problem Detector Operator は、一般的なインストールおよびストレージに関連する正しくない設定の問題について vSphere にデプロイされたクラスターをチェックします。



注記

vSphere でクラスターがデプロイされていることが、Cluster Storage Operator で検出された場合にのみ、Cluster Storage Operator により vSphere Problem Detector Operator が起動されます。

設定

必要な設定はありません。

注記

- Operator は、vSphere での OpenShift Container Platform のインストールをサポートします。
- Operator は **vsphere-cloud-credentials** を使用して vSphere と通信します。
- Operator はストレージに関連するチェックを実行します。

関連情報

- [vSphere Problem Detector Operator の使用](#)