



OpenShift Container Platform 4.5

Operator

OpenShift Container Platform での Operator の使用

OpenShift Container Platform 4.5 Operator

OpenShift Container Platform での Operator の使用

Enter your first name here. Enter your surname here.

Enter your organisation's name here. Enter your organisational division here.

Enter your email address here.

法律上の通知

Copyright © 2022 | You need to change the HOLDER entity in the en-US/Operators.ent file |.

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 について | 9 |
| 1.1. OPERATOR について | 9 |
| 1.1.1. Operator を使用する理由 | 9 |
| 1.1.2. Operator Framework | 10 |
| 1.1.3. Operator 成熟度モデル | 10 |
| 1.2. OPERATOR FRAMEWORK の一般的な用語の用語集 | 11 |
| 1.2.1. Common Operator Framework の一般的な用語 | 11 |
| 1.2.1.1. バンドル | 11 |
| 1.2.1.2. バンドルイメージ | 11 |
| 1.2.1.3. カタログソース | 11 |
| 1.2.1.4. カタログイメージ | 11 |
| 1.2.1.5. チャンネル | 12 |
| 1.2.1.6. チャンネルヘッド | 12 |
| 1.2.1.7. クラスターサービスバージョン | 12 |
| 1.2.1.8. 依存関係 | 12 |
| 1.2.1.9. インデックスイメージ | 12 |
| 1.2.1.10. インストール計画 | 12 |
| 1.2.1.11. Operator グループ | 12 |
| 1.2.1.12. パッケージ | 12 |
| 1.2.1.13. レジストリー | 13 |
| 1.2.1.14. サブスクリプション | 13 |
| 1.2.1.15. 更新グラフ | 13 |
| 1.3. OPERATOR FRAMEWORK パッケージ形式 | 13 |
| 1.3.1. Package Manifest Format | 13 |
| 1.3.2. Bundle Format | 14 |
| 1.3.2.1. マニフェスト | 15 |
| オプションのオブジェクト | 15 |
| 1.3.2.2. アノテーション | 15 |
| 1.3.2.3. 依存関係 | 16 |
| 1.3.2.4. opm CLI | 17 |
| 1.4. OPERATOR LIFECYCLE MANAGER (OLM) | 17 |
| 1.4.1. Operator Lifecycle Manager の概念 | 17 |
| 1.4.1.1. Operator Lifecycle Manager について | 17 |
| 1.4.1.2. OLM リソース | 18 |
| 1.4.1.2.1. クラスターサービスバージョン | 18 |
| 1.4.1.2.2. カタログソース | 19 |
| 1.4.1.2.3. サブスクリプション | 20 |
| 1.4.1.2.4. インストール計画 | 20 |
| 1.4.1.2.5. Operator グループ | 20 |
| 1.4.2. Operator Lifecycle Manager アーキテクチャー | 21 |
| 1.4.2.1. コンポーネントのロール | 21 |
| 1.4.2.2. OLM Operator | 22 |
| 1.4.2.3. カタログ Operator | 22 |
| 1.4.2.4. カタログレジストリー | 23 |
| 1.4.3. Operator Lifecycle Manager ワークフロー | 23 |
| 1.4.3.1. 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 | 23 |
| 1.4.3.1.1. アップグレードパスの例 | 25 |
| 1.4.3.1.2. アップグレードの省略 | 25 |
| 1.4.3.1.3. 複数 Operator の置き換え | 27 |
| 1.4.3.1.4. z-stream サポート | 28 |
| 1.4.4. Operator Lifecycle Manager の依存関係の解決 | 29 |

| | |
|--|-----------|
| 1.4.4.1. 依存関係の解決 | 29 |
| 1.4.4.2. CRD のアップグレード | 29 |
| 1.4.4.2.1. 新規 CRD バージョンの追加 | 29 |
| 1.4.4.2.2. CRD バージョンの非推奨または削除 | 30 |
| 1.4.4.3. 依存関係解決のシナリオ例 | 31 |
| 例: 依存 API を非推奨にする | 31 |
| 例: バージョンのデッドロック | 31 |
| 1.4.5. Operator グループ | 32 |
| 1.4.5.1. Operator グループについて | 32 |
| 1.4.5.2. Operator グループメンバーシップ | 32 |
| 1.4.5.3. ターゲット namespace の選択 | 33 |
| 1.4.5.4. Operator グループの CSV アノテーション | 33 |
| 1.4.5.5. 提供される API アノテーション | 34 |
| 1.4.5.6. ロールベースのアクセス制御 | 34 |
| 1.4.5.7. コピーされる CSV | 38 |
| 1.4.5.8. 静的 Operator グループ | 38 |
| 1.4.5.9. Operator グループの交差部分 | 38 |
| 交差のルール | 39 |
| 1.4.5.10. Operator グループのトラブルシューティング | 40 |
| メンバーシップ | 40 |
| 1.4.6. Operator Lifecycle Manager メトリクス | 40 |
| 1.4.6.1. 公開されるメトリクス | 40 |
| 1.5. OPERATORHUB について | 41 |
| 1.5.1. OperatorHub について | 41 |
| 1.5.2. OperatorHub アーキテクチャー | 42 |
| 1.5.2.1. OperatorHub CRD | 42 |
| 1.5.2.2. OperatorSource CRD | 42 |
| 1.5.3. 追加リソース | 43 |
| 1.6. CRD | 43 |
| 1.6.1. カスタムリソース定義による Kubernetes API の拡張 | 43 |
| 1.6.1.1. カスタムリソース定義 | 43 |
| 1.6.1.2. カスタムリソース定義の作成 | 44 |
| 1.6.1.3. カスタムリソース定義のクラスターロールの作成 | 45 |
| 1.6.1.4. ファイルからのカスタムリソースの作成 | 47 |
| 1.6.1.5. カスタムリソースの検査 | 48 |
| 1.6.2. カスタムリソース定義からのリソースの管理 | 49 |
| 1.6.2.1. カスタムリソース定義 | 49 |
| 1.6.2.2. ファイルからのカスタムリソースの作成 | 49 |
| 1.6.2.3. カスタムリソースの検査 | 50 |
| 第2章 ユーザータスク | 52 |
| 2.1. インストールされた OPERATOR からのアプリケーションの作成 | 52 |
| 2.1.1. Operator を使用した etcd クラスターの作成 | 52 |
| 2.2. NAMESPACE への OPERATOR のインストール | 53 |
| 2.2.1. 前提条件 | 53 |
| 2.2.2. OperatorHub を使用した Operator のインストール | 53 |
| 2.2.3. Web コンソールを使用した OperatorHub からのインストール | 54 |
| 2.2.4. CLI を使用した OperatorHub からのインストール | 55 |
| 2.2.5. Operator の特定バージョンのインストール | 57 |
| 2.3. OPERATOR LIFECYCLE MANAGER での受付 WEBHOOK の管理 | 58 |
| 2.3.1. CSV での Webhook の定義 | 58 |
| 2.3.2. Webhook の考慮事項 | 59 |
| 認証局についての制約 | 59 |

| | |
|--|------------|
| 受付 Webhook ルールについての制約 | 59 |
| 2.3.3. 追加リソース | 60 |
| 第3章 管理者タスク | 61 |
| 3.1. OPERATOR のクラスターへの追加 | 61 |
| 3.1.1. OperatorHub を使用した Operator のインストール | 61 |
| 3.1.2. Web コンソールを使用した OperatorHub からのインストール | 61 |
| 3.1.3. CLI を使用した OperatorHub からのインストール | 63 |
| 3.1.4. Operator の特定バージョンのインストール | 65 |
| 3.2. インストールされた OPERATOR のアップグレード | 66 |
| 3.2.1. Operator の更新チャンネルの変更 | 66 |
| 3.2.2. 保留中の Operator アップグレードの手動による承認 | 67 |
| 3.3. クラスターからの OPERATOR の削除 | 67 |
| 3.3.1. Web コンソールの使用によるクラスターからの Operator の削除 | 68 |
| 3.3.2. CLI の使用によるクラスターからの Operator の削除 | 68 |
| 3.4. OPERATOR LIFECYCLE MANAGER でのプロキシサポートの設定 | 69 |
| 3.4.1. Operator のプロキシ設定の上書き | 69 |
| 3.4.2. カスタム CA 証明書の挿入 | 71 |
| 3.5. OPERATOR ステータスの表示 | 72 |
| 3.5.1. Operator サブスクリプションの状態のタイプ | 72 |
| 3.5.2. CLI を使用した Operator サブスクリプションステータスの表示 | 73 |
| 3.6. 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 | 74 |
| 3.6.1. Operator インストールポリシーについて | 74 |
| 3.6.1.1. インストールシナリオ | 74 |
| 3.6.1.2. インストールワークフロー | 75 |
| 3.6.2. Operator インストールのスコープ設定 | 75 |
| 3.6.2.1. 粒度の細かいパーミッション | 77 |
| 3.6.3. 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 | 78 |
| 3.7. カスタムカタログの管理 | 80 |
| 3.7.1. Package Manifest Format を使用したカスタムカタログ | 80 |
| 3.7.1.1. Operator カタログイメージについて | 80 |
| 3.7.1.2. Operator カタログイメージのビルド | 81 |
| 3.7.1.3. Operator カタログイメージのミラーリング | 83 |
| 3.7.1.4. Operator カタログイメージの更新 | 87 |
| 3.7.1.5. Operator カタログイメージのテスト | 90 |
| 3.7.2. Bundle Format を使用したカスタムカタログ | 92 |
| 3.7.2.1. opm CLI | 92 |
| 3.7.2.2. opm のインストール | 93 |
| 3.7.2.3. インデックスイメージの作成 | 93 |
| 3.7.2.4. インデックスイメージからのカタログの作成 | 94 |
| 3.7.2.5. インデックスイメージの更新 | 95 |
| 3.8. ネットワークが制限された環境での OPERATOR LIFECYCLE MANAGER の使用 | 96 |
| 3.8.1. Operator カタログイメージについて | 97 |
| 3.8.2. Operator カタログイメージのビルド | 97 |
| 3.8.3. ネットワークが制限された環境向けの OperatorHub の設定 | 100 |
| 3.8.4. Operator カタログイメージの更新 | 104 |
| 3.8.5. Operator カタログイメージのテスト | 107 |
| 第4章 OPERATOR の開発 | 110 |
| 4.1. OPERATOR SDK の使用を開始する | 110 |
| 4.1.1. Operator SDK のアーキテクチャー | 110 |
| 4.1.1.1. ワークフロー | 110 |
| 4.1.1.2. マネージャーファイル | 111 |

| | |
|--|-----|
| 4.1.1.3. Prometheus Operator のサポート | 111 |
| 4.1.2. Operator SDK CLI のインストール | 111 |
| 4.1.2.1. GitHub リリースからのインストール | 112 |
| 4.1.2.2. Homebrew からのインストール | 114 |
| 4.1.2.3. ソースを使用したコンパイルおよびインストール | 114 |
| 4.1.3. Operator SDK を使用した Go ベースの Operator のビルド | 115 |
| 4.1.4. Operator Lifecycle Manager を使用した Go ベースの Operator の管理 | 122 |
| 4.1.5. 追加リソース | 125 |
| 4.2. ANSIBLE ベース OPERATOR の作成 | 125 |
| 4.2.1. Operator SDK における Ansible サポート | 125 |
| 4.2.1.1. カスタムリソースファイル | 125 |
| 4.2.1.2. watches.yaml ファイル | 126 |
| 4.2.1.2.1. 高度なオプション | 127 |
| 4.2.1.3. Ansible に送信される追加変数 | 128 |
| 4.2.1.4. Ansible Runner ディレクトリー | 129 |
| 4.2.2. Operator SDK CLI のインストール | 129 |
| 4.2.2.1. GitHub リリースからのインストール | 129 |
| 4.2.2.2. Homebrew からのインストール | 131 |
| 4.2.2.3. ソースを使用したコンパイルおよびインストール | 132 |
| 4.2.3. Operator SDK を使用した Ansible ベースの Operator のビルド | 133 |
| 4.2.4. K8S Ansible モジュールの使用によるアプリケーションライフサイクルの管理 | 138 |
| 4.2.4.1. k8s Ansible モジュールのインストール | 139 |
| 4.2.4.2. k8s Ansible モジュールのローカルでのテスト | 139 |
| 4.2.4.3. Operator 内での k8s Ansible モジュールのテスト | 141 |
| 4.2.4.3.1. Ansible ベース Operator のローカルでのテスト | 142 |
| 4.2.4.3.2. Ansible ベース Operator のクラスター上でのテスト | 143 |
| 4.2.5. operator_sdk.util Ansible コレクションを使用したカスタムリソースのステータス管理 | 144 |
| 4.2.6. 追加リソース | 146 |
| 4.3. HELM ベース OPERATOR の作成 | 146 |
| 4.3.1. Operator SDK での Helm チャートのサポート | 146 |
| 4.3.2. Operator SDK CLI のインストール | 147 |
| 4.3.2.1. GitHub リリースからのインストール | 147 |
| 4.3.2.2. Homebrew からのインストール | 149 |
| 4.3.2.3. ソースを使用したコンパイルおよびインストール | 150 |
| 4.3.3. Operator SDK を使用した Helm ベースの Operator のビルド | 151 |
| 4.3.4. 追加リソース | 156 |
| 4.4. クラスターサービスバージョン (CSV) の生成 | 156 |
| 4.4.1. CSV 生成の仕組み | 157 |
| ワークフロー | 157 |
| 4.4.2. CSV 設定の設定 | 158 |
| 4.4.3. 手動で定義される CSV フィールド | 158 |
| 4.4.4. CSV の生成 | 160 |
| 4.4.5. ネットワークが制限された環境についての Operator の有効化 | 160 |
| 4.4.6. 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 | 162 |
| 4.4.6.1. 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 | 163 |
| 4.4.7. 推奨される namespace の設定 | 164 |
| 4.4.8. カスタムリソース定義 (CRD) について | 164 |
| 4.4.8.1. 所有 CRD (Owned CRD) | 165 |
| 4.4.8.2. 必須 CRD (Required CRD) | 167 |
| 4.4.8.3. CRD テンプレート | 168 |
| 4.4.8.4. 内部オブジェクトの非表示 | 168 |
| 4.4.9. API サービスについて | 169 |
| 4.4.9.1. 所有 API サービス | 169 |

| | |
|--|------------|
| 4.4.9.1.1. API サービスリソースの作成 | 170 |
| 4.4.9.1.2. API サービス提供証明書 | 170 |
| 4.4.9.2. 必要な API サービス | 171 |
| 4.5. バンドルイメージの使用 | 171 |
| 4.5.1. バンドルイメージのビルド | 171 |
| 4.5.2. 追加リソース | 172 |
| 4.6. スコアカードを使用した OPERATOR の検証 | 172 |
| 4.6.1. スコアカードツールについて | 173 |
| 4.6.2. スコアカードの設定 | 173 |
| 4.6.2.1. 設定ファイル | 173 |
| 4.6.2.2. コマンド引数 | 174 |
| 4.6.2.3. 設定ファイルのオプション | 174 |
| 4.6.2.3.1. 基本的なプラグインおよび OLM プラグイン | 175 |
| 4.6.3. 実行されるテスト | 176 |
| 4.6.3.1. 基本的なプラグイン | 176 |
| 4.6.3.2. OLM プラグイン | 177 |
| 4.6.4. スコアカードの実行 | 178 |
| 4.6.5. OLM 管理の Operator を使用したスコアカードの実行 | 179 |
| 4.7. PROMETHEUS による組み込みモニターリングの設定 | 183 |
| 4.7.1. Prometheus Operator のサポート | 183 |
| 4.7.2. メトリクスヘルパー | 183 |
| 4.7.2.1. メトリクスポートの変更 | 184 |
| 4.7.3. サービスモニター | 184 |
| 4.7.3.1. サービスモニターの作成 | 184 |
| 4.8. リーダー選択の設定 | 185 |
| 4.8.1. Leader-for-life 選択の使用 | 186 |
| 4.8.2. Leader-with-lease 選択の使用 | 186 |
| 4.9. OPERATOR SDK CLI リファレンス | 187 |
| 4.9.1. build | 187 |
| 4.9.2. completion | 188 |
| 4.9.3. print-deps | 188 |
| 4.9.4. generate | 189 |
| 4.9.4.1. crds | 189 |
| 4.9.4.2. csv | 190 |
| 4.9.4.3. k8s | 191 |
| 4.9.5. new | 191 |
| 4.9.6. add | 193 |
| 4.9.7. test | 195 |
| 4.9.7.1. local | 195 |
| 4.9.8. run | 196 |
| 4.9.8.1. --local | 196 |
| 4.10. 付録 | 197 |
| 4.10.1. Operator プロジェクトのスキヤフォールディングレイアウト | 197 |
| 4.10.1.1. Go ベースプロジェクト | 197 |
| 4.10.1.2. Helm ベースのプロジェクト | 198 |
| 第5章 RED HAT OPERATOR | 199 |
| 5.1. CLOUD CREDENTIAL OPERATOR | 199 |
| 目的 | 199 |
| プロジェクト | 199 |
| CRD | 199 |
| 設定オブジェクト | 199 |
| 注記 | 199 |

| | |
|---|-----|
| 5.2. クラスター認証 OPERATOR | 199 |
| 目的 | 199 |
| プロジェクト | 199 |
| 5.3. CLUSTER AUTOSCALER OPERATOR | 199 |
| 目的 | 199 |
| プロジェクト | 199 |
| CRD | 199 |
| 5.4. CLUSTER IMAGE REGISTRY OPERATOR | 200 |
| 目的 | 200 |
| プロジェクト | 200 |
| 5.5. クラスターモニタリング OPERATOR | 200 |
| 目的 | 200 |
| プロジェクト | 200 |
| CRD | 200 |
| 設定オブジェクト | 201 |
| 5.6. CLUSTER NETWORK OPERATOR | 201 |
| 目的 | 201 |
| 5.7. OPENSIFT CONTROLLER MANAGER OPERATOR | 201 |
| 目的 | 201 |
| プロジェクト | 201 |
| 5.8. CLUSTER SAMPLES OPERATOR | 201 |
| 目的 | 201 |
| プロジェクト | 202 |
| 5.9. CLUSTER STORAGE OPERATOR | 202 |
| 目的 | 202 |
| プロジェクト | 202 |
| 設定 | 202 |
| 注記 | 202 |
| 5.10. CLUSTER VERSION OPERATOR | 203 |
| 目的 | 203 |
| プロジェクト | 203 |
| 5.11. CONSOLE OPERATOR | 203 |
| 目的 | 203 |
| プロジェクト | 203 |
| 5.12. DNS OPERATOR | 203 |
| 目的 | 203 |
| プロジェクト | 203 |
| 5.13. ETCD CLUSTER OPERATOR | 203 |
| 目的 | 203 |
| プロジェクト | 203 |
| CRD | 203 |
| 設定オブジェクト | 204 |
| 5.14. INGRESS OPERATOR | 204 |
| 目的 | 204 |
| プロジェクト | 204 |
| CRD | 204 |
| 設定オブジェクト | 204 |
| 注記 | 204 |
| 5.15. KUBERNETES API SERVER OPERATOR | 205 |
| 目的 | 205 |
| プロジェクト | 205 |
| CRD | 205 |
| 設定オブジェクト | 205 |

| | |
|--|-----|
| 5.16. KUBERNETES CONTROLLER MANAGER OPERATOR | 205 |
| 目的 | 205 |
| プロジェクト | 205 |
| 5.17. KUBERNETES SCHEDULER OPERATOR | 205 |
| 目的 | 205 |
| プロジェクト | 206 |
| 設定 | 206 |
| 5.18. MACHINE API OPERATOR | 206 |
| 目的 | 206 |
| プロジェクト | 206 |
| CRD | 206 |
| 5.19. MACHINE CONFIG OPERATOR | 206 |
| 目的 | 206 |
| プロジェクト | 207 |
| 5.20. MARKETPLACE OPERATOR | 207 |
| 目的 | 207 |
| プロジェクト | 207 |
| 5.21. NODE TUNING OPERATOR | 207 |
| 目的 | 207 |
| プロジェクト | 207 |
| 5.22. OPERATOR LIFECYCLE MANAGER OPERATOR | 207 |
| 目的 | 207 |
| CRD | 208 |
| OLM Operator | 209 |
| カタログ Operator | 209 |
| カタログレジストリー | 210 |
| 関連情報 | 210 |
| 5.23. OPENSIFT API SERVER OPERATOR | 210 |
| 目的 | 210 |
| プロジェクト | 210 |
| CRD | 210 |
| 5.24. PROMETHEUS OPERATOR | 211 |
| 目的 | 211 |
| プロジェクト | 211 |

第1章 OPERATOR について

1.1. 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。

1.1.1. 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。

1.1.2. 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に Operator の作成者を支援します。

Operator Lifecycle Manager

Operator Lifecycle Manager 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 4.5 ではデフォルトでデプロイされます。

Operator レジストリー

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

OperatorHub

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

Operator Metering

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

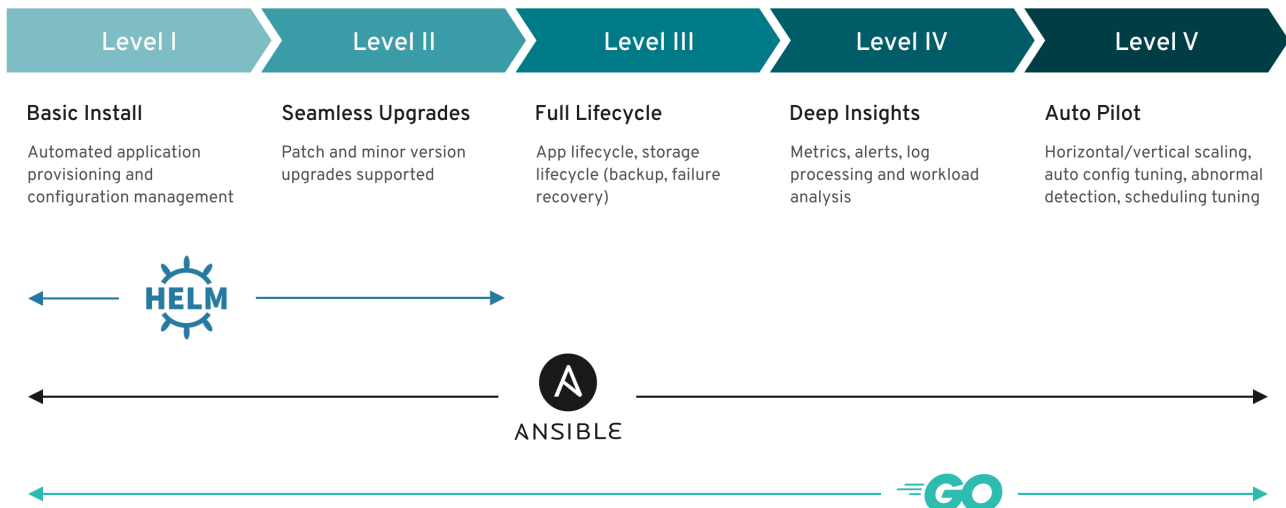
これらのツールは組み立て可能なツールとして設計されているため、役に立つと思われるツールを使用できます。

1.1.3. 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1.1 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。

1.2. 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 (Package Manifest Format および Bundle Format の両方) 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。

1.2.1. Common Operator Framework の一般的な用語

1.2.1.1. 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。

1.2.1.2. 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。

1.2.1.3. 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。

1.2.1.4. カタログイメージ

Package Manifest Format で、**カタログイメージ** は、Operator メタデータのセットを記述し、OLM を使用してクラスターにインストールできるメタデータを更新するコンテナ化されたデータストアです。

1.2.1.5. 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へと編成されます。

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。

1.2.1.6. 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。

1.2.1.7. 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。

1.2.1.8. 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。

1.2.1.9. 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。

1.2.1.10. 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。

1.2.1.11. Operator グループ

Operator グループ は、**OperatorGroup** オブジェクトと同じ namespace にデプロイされたすべての Operator を、namespace の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。

1.2.1.12. パッケージ

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。

1.2.1.13. レジストリー

レジストリー は、Operator のバンドルイメージを保存するデータベースで、それぞれにすべてのチャン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。

1.2.1.14. サブスクリプション

サブスクリプション は、パッケージのチャンネルを追跡して CSV を最新の状態に保ちます。

1.2.1.15. 更新グラフ

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

1.3. 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。

1.3.1. Package Manifest Format

Operator の **Package Manifest Format** は、Operator Framework で導入されたレガシーパッケージ形式です。この形式は OpenShift Container Platform 4.5 で非推奨となっていますが、これは引き続きサポートされ、現在 Red Hat が提供する Operator はこの方法を使用して提供されています。

この形式では、Operator のバージョンは単一のクラスターサービスバージョン (CSV) で表され、通常は追加のオブジェクトが含まれる可能性はありますが CSV の所有される API を定義するカスタムリソース定義 (CRD) で表されます。

Operator のすべてのバージョンは単一ディレクトリーにネストされます。

Package Manifest Format のレイアウトの例

```

etcd
├── 0.6.1
│   ├── etcdcluster.crd.yaml
│   └── etcdoperator.clusterserviceversion.yaml
├── 0.9.0
│   ├── etcdbackup.crd.yaml
│   ├── etcdcluster.crd.yaml
│   ├── etcdoperator.v0.9.0.clusterserviceversion.yaml
│   └── etcdrestore.crd.yaml
├── 0.9.2
│   ├── etcdbackup.crd.yaml
│   ├── etcdcluster.crd.yaml
│   ├── etcdoperator.v0.9.2.clusterserviceversion.yaml
│   └── etcdrestore.crd.yaml
└── etcd.package.yaml
  
```

また、パッケージ名およびチャネルの詳細を定義する **package manifest** である **<name>.package.yaml** ファイルも含まれます。

パッケージマニフェストの例

```
packageName: etcd
channels:
- name: alpha
  currentCSV: etcdoperator.v0.9.2
- name: beta
  currentCSV: etcdoperator.v0.9.0
- name: stable
  currentCSV: etcdoperator.v0.9.2
defaultChannel: alpha
```

パッケ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。

- すべてのパッケージには最低でも1つのチャネルがある。
- パッケージのチャネルによって参照されるすべての CSV が存在する。
- Operator のすべてのバージョンに1つの CSV がある。
- CSV が CRD を所有する場合、その CRD は Operator バージョンのディレクトリーに存在する必要がある。
- CSV が別の CSV を置き換える場合、新旧両方の CSV がパッケージに存在する必要がある。

1.3.2. Bundle Format

Operator の **Bundle Format** は、Operator Framework によって導入される新しい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。

1.3.2.1. 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

```

オプションのオブジェクト

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

- **Secrets**
- **ConfigMaps**

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。

1.3.2.2. 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。
- ❻ 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。

1.3.2.3. 依存関係

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

依存関係の一覧には、依存関係の内容を指定するために各項目の **type** フィールドが含まれます。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**: パッケージタイプは、これが特定の 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 ベースの使用法と同様に 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
```

1.3.2.4. opm CLI

新規の **opm** CLI ツールが新規の Bundle Format と共に導入されます。このツールを使用して、リポジトリに相当する **インデックス** 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、OLM はインデックスイメージを CatalogSource 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。

1.4. OPERATOR LIFECYCLE MANAGER (OLM)

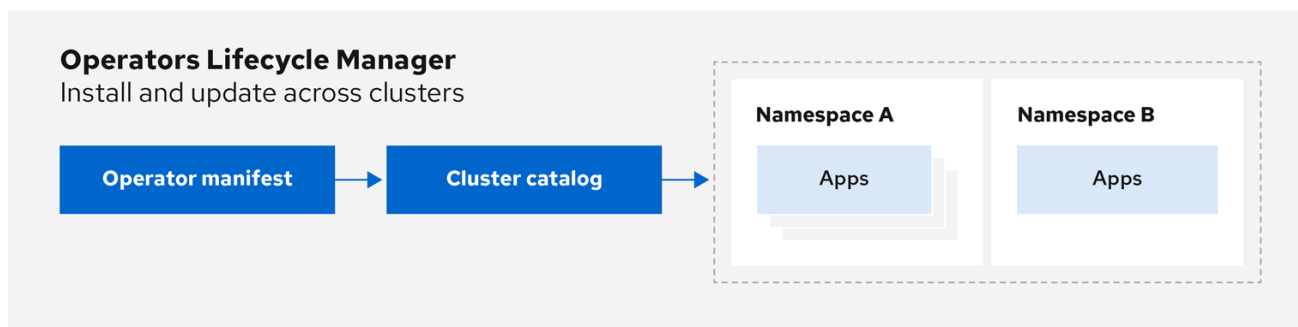
1.4.1. Operator Lifecycle Manager の概念

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。

1.4.1.1. 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1.2 Operator Lifecycle Manager ワークフロー



OpenShift_43_1019

OLM は OpenShift Container Platform 4.5 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 コンソールは、クラスター管理者が Operator をインストールしたり、クラスターで利用可能な Operator のカタログを使用できるように特定のプロジェクトアクセスを付与したりするのに使用する管理画面を提供します。

開発者の場合は、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。

1.4.1.2. 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1.1 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。 |
| 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。 |

1.4.1.2.1. 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。

1.4.1.2.2. カタログソース

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

CatalogSource オブジェクトには、3 つの主な **sourceTypes** があります。

- **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。
- **internal** または **configmap**: OLM は ConfigMap データを解析し、gRPC API を提供できる Pod を実行します。

以下の例では、OperatorHub.io コンテンツのカタログソースを定義します。

CatalogSource オブジェクトの例

```
apiVersion: operators.coreos.com/v1alpha1
kind: CatalogSource
metadata:
  name: operatorhubio-catalog
  namespace: olm
spec:
  sourceType: grpc
  image: quay.io/operator-framework/upstream-community-operators:latest
  displayName: Community Operators
  publisher: OperatorHub.io
```

CatalogSource オブジェクトの **name** は、サブスクリプションへの入力として使用され、OLM に対して要求された Operator を検索する場所を指示します。

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: my-operator
  namespace: olm
spec:
  channel: stable
  name: my-operator
  source: operatorhubio-catalog
```

1.4.1.2.3. サブスクリプション

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: my-operator
  namespace: operators
spec:
  channel: stable
  name: my-operator
  source: my-catalog
  sourceNamespace: operators
```

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。

1.4.1.2.4. 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、クラスターサービスバージョン (CSV) によって定義される特定バージョンの 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されるリソースのセットを記述します。

1.4.1.2.5. 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、それらのデプロイメントに展開されます。

詳細は、[Operator グループ](#) についてのガイドを参照してください。

1.4.2. 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。

1.4.2.1. 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1.2 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 | catalog | 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1.3 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 | |

1.4.2.2. 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。

1.4.2.3. 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. カタログソースおよびサブスクリプションの有無を確認し、それらに基づいてインストール計画を作成します。

1.4.2.4. 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)。

1.4.3. 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。

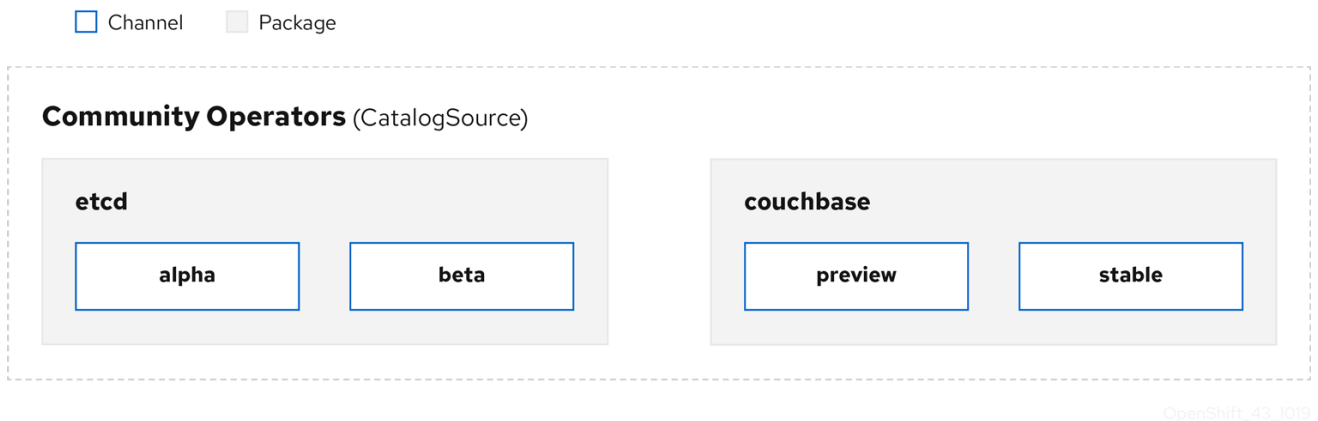
1.4.3.1. OLM での Operator のインストールおよびアップグレードのワークフロー

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**

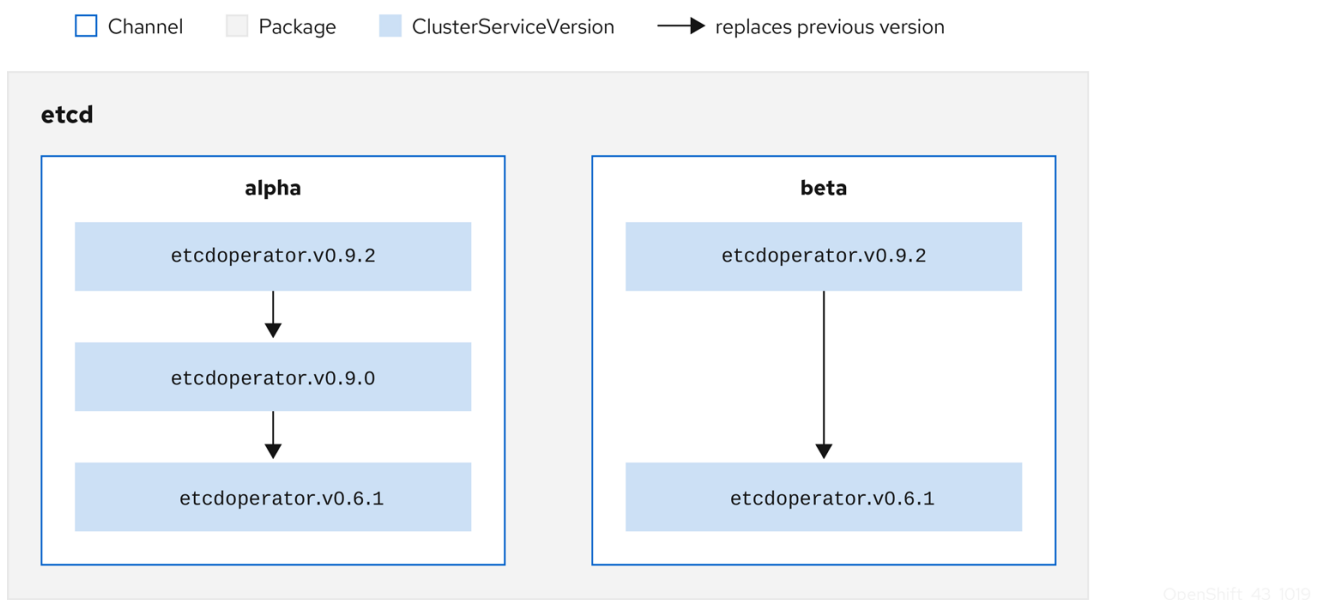
CSV で定義される Operator メタデータは、カタログソースというコレクションに保存できます。OLM はカタログソースを使用します。これは [Operator Registry API](#) を使用して利用可能な Operator やインストールされた Operator のアップグレードについてクエリーします。

図1.3 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1.4 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。

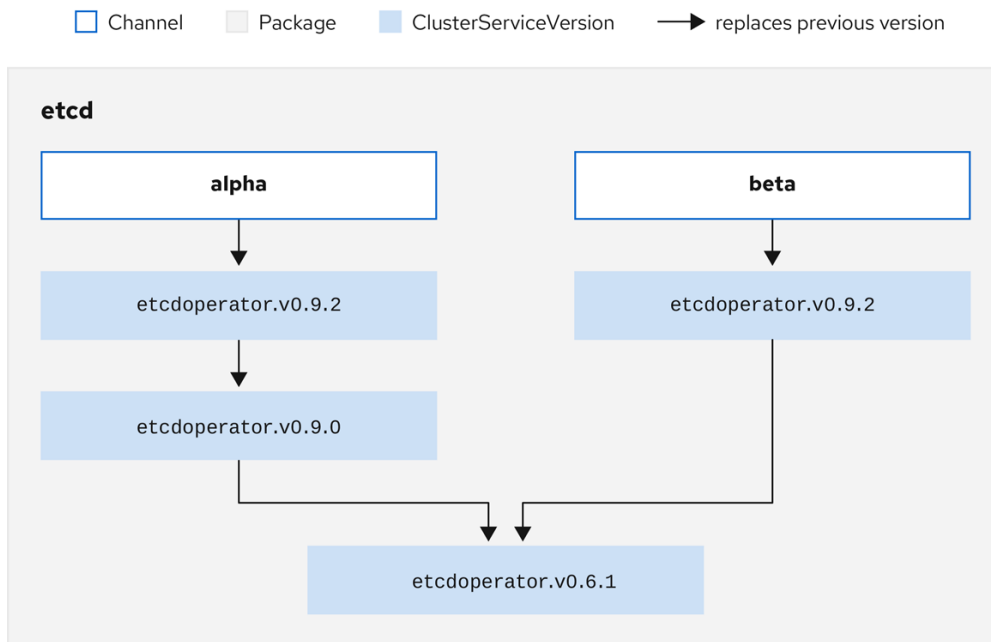


注記

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1.5 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。

1.4.3.1.1. 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** についてサブスクリ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。

1.4.3.1.2. 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。

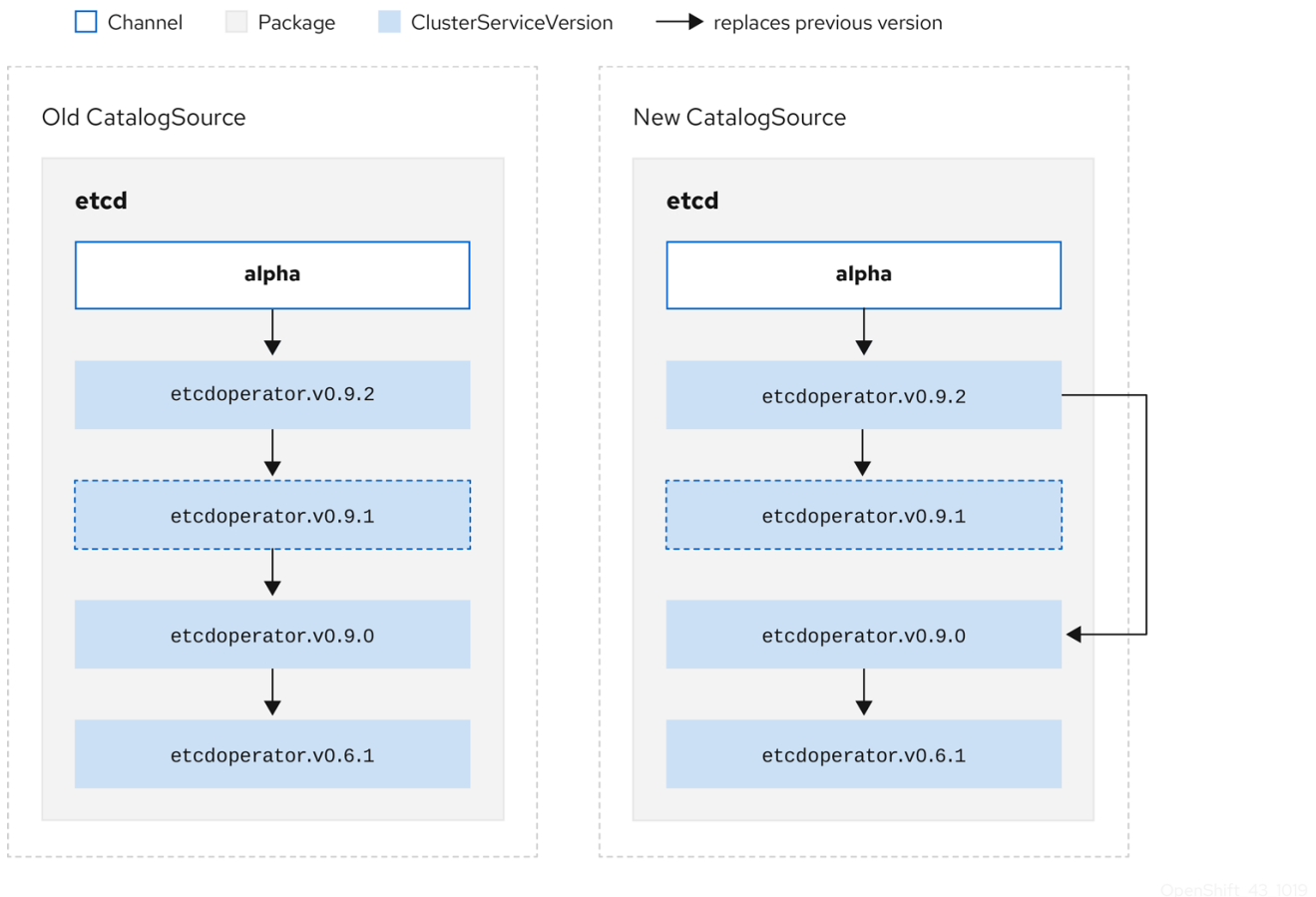
OLM は、新規カタログを送り、**省略された**リリースを追加することで、クラスターの状態や問題のある更新が発見されたかどうかにかかわらず、単一の固有の更新を常に取得することができます。

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1.6 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。

1.4.3.1.3. 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

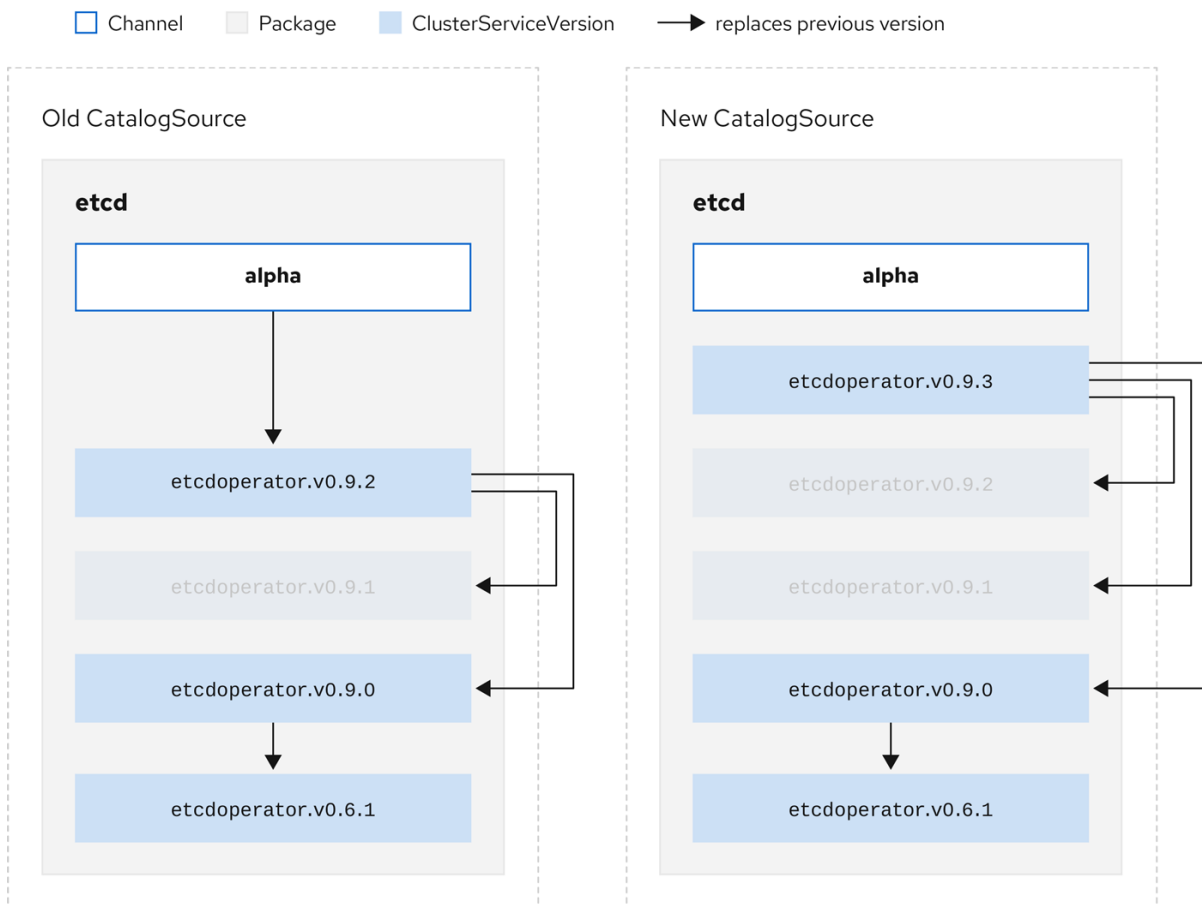
```
apiVersion: operators.coreos.com/v1alpha1
kind: ClusterServiceVersion
metadata:
  name: elasticsearch-operator.v4.1.2
  namespace: <namespace>
  annotations:
    olm.skipRange: '>=4.1.0 <4.1.2'
```

1.4.3.1.4. 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1.7 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。

1.4.4. 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。

1.4.4.1. 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 には通常同様のシステムには1つの制約があります。それは、Operator は常に実行中であるため、OLM は相互に機能しない Operator のセットの共存を防ごうとする点です。

つまり、これは OLM が以下を実行しないことを意味します。

- 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

1.4.4.2. 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。

1.4.4.2.1. 新規 CRD バージョンの追加

手順

CRD の新規バージョンを追加するには、以下を実行します。

1. **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 ❶
  served: true
  storage: true
```

❶ 新規エントリー。

- 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。

- 更新された CRD および CSV をバンドルにプッシュします。

1.4.4.2.2. 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。

- 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。

- 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 ②
```

① ② **storage** フィールドを適宜更新します。



注記

CRD から **storage** バージョンであるか、このバージョンであった特定のバージョンを削除するために、そのバージョンが CRD のステータスの **storedVersion** から削除される必要があります。OLM は、保存されたバージョンが新しい CRD に存在しないことを検知した場合に、この実行を試行します。

- 上記の変更内容で CRD をアップグレードします。
- 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
```

- 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。

1.4.4.3. 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。

1.4.5. 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。

1.4.5.1. 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。

1.4.5.2. 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1.4 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。 |
| 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。

1.4.5.3. 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。

1.4.5.4. 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。

1.4.5.5. 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
```

1.4.5.6. ロールベースのアクセス制御

Operator グループの作成時に、3 つのクラスタール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1.5 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 |
|--|---|
| <kind>.<group>-<version>-edit | <p><kind> の動詞</p> <ul style="list-style-type: none"> ● create ● update ● patch ● delete <p>集計ラベル:</p> <ul style="list-style-type: none"> ● rbac.authorization.k8s.io/aggregate-to-edit: true ● olm.opgroup.permissions/aggregate-to-edit: <operatorgroup_name> |
| <kind>.<group>-<version>-view | <p><kind> の動詞</p> <ul style="list-style-type: none"> ● get ● list ● watch <p>集計ラベル:</p> <ul style="list-style-type: none"> ● rbac.authorization.k8s.io/aggregate-to-view: true ● olm.opgroup.permissions/aggregate-to-view: <operatorgroup_name> |
| <kind>.<group>-<version>-view-crdview | <p>Verbs on apiextensions.k8s.io customresourcedefinitions <crd-name>:</p> <ul style="list-style-type: none"> ● get <p>集計ラベル:</p> <ul style="list-style-type: none"> ● rbac.authorization.k8s.io/aggregate-to-view: true ● olm.opgroup.permissions/aggregate-to-view: <operatorgroup_name> |

表1.6 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。

1.4.5.7. 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。

1.4.5.8. 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"
```

1.4.5.9. 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。



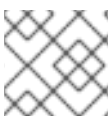
注記

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。

1.4.5.10. Operator グループのトラブルシューティング

メンバーシップ

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

1.4.6. Operator Lifecycle Manager メトリクス

1.4.6.1. 公開されるメトリクス

Operator Lifecycle Manager (OLM) は、Prometheus ベースの OpenShift Container Platform クラスターモニター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1.7 OLM によって公開されるメトリクス

| 名前 | 説明 |
|-----------------------------|--|
| catalog_source_count | カタログソースの数。 |
| csv_abnormal | クラスターサービスバージョン (CSV) を調整する際に、(インストールされていない場合など) CSV バージョンが Succeeded 以外の状態にあることを表します。 name 、 namespace 、 phase 、 reason 、および version ラベルが含まれます。Prometheus アラートは、このメトリクスが存在する場合に作成されます。 |
| csv_count | 正常に登録された CSV の数。 |
| csv_succeeded | CSV を調整する際に、CSV バージョンが Succeeded 状態 (値 1) にあるか、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。 |

| 名前 | 説明 |
|--------------------------------|--|
| 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。 |

1.5. OPERATORHUB について

1.5.1. 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、以下のカテゴリにグループ化された Operator ソースから選択することができます。

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



注記

OperatorHub コンテンツは 60 分ごとに自動的に更新されます。

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、Deployment、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 の ISV パートナーポータル (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。

1.5.2. 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。

Marketplace Operator は **OperatorHub** および **OperatorSource** カスタムリソース定義 (CRD) を管理します。



注記

一部の **OperatorSource** オブジェクト情報は OperatorHub UI 公開されますが、それは独自の Operator を作成するユーザーによってのみ直接使用されます。

1.5.2.1. OperatorHub CRD

OperatorHub CRD を使用して、クラスター上で OperatorHub で提供されているデフォルト **OperatorSource** オブジェクトの状態を enabled と disabled 間で切り替えることができます。この機能は、OpenShift Container Platform をネットワークが制限された環境で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の **OperatorSource** オブジェクト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、デフォルトのソース を個別に無効にします。

1.5.2.2. OperatorSource CRD

それぞれの Operator について、**OperatorSource** CRD は Operator バンドルを保存するために使用される外部データストアを定義するために使用されます。

OperatorSource カスタムリソースの例

```
apiVersion: operators.coreos.com/v1
```

```

kind: OperatorSource
metadata:
  name: community-operators
  namespace: marketplace
spec:
  type: appregistry ❶
  endpoint: https://quay.io/cnr ❷
  registryNamespace: community-operators ❸
  displayName: "Community Operators" ❹
  publisher: "Red Hat" ❺

```

- ❶ データストアをアプリケーションレジストリーとして識別するために、**type** は **appregistry** に設定されます。
- ❷ 現時点で、Quay は OperatorHub によって使用される外部データストアであるため、エンドポイントは Quay.io **appregistry** について **https://quay.io/cnr** に設定されます。
- ❸ コミュニティ Operator の場合、**registryNamespace** は **community-operator** に設定されます。
- ❹ オプションで、**displayName** を、OperatorHub UI の Operator の表示される名前に設定します。
- ❺ オプションで、**publisher** を、OperatorHub UI に表示される Operator を公開する人または組織に設定します。

1.5.3. 追加リソース

- [カタログソース](#)
- [Operator SDK の使用を開始する](#)
- [ClusterServiceVersion \(CSV\) の生成](#)
- [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](#)

1.6. CRD

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

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

1.6.1.1. カスタムリソース定義

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

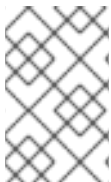
カスタムリソース定義 (CRD) オブジェクトは、クラスター内に新規の固有オブジェクト **kind** を定義し、Kubernetes API サーバーにそのライフサイクル全体を処理させます。

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。

1.6.1.2. カスタムリソース定義の作成

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

前提条件

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

手順

CRD を作成するには、以下を実行します。

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

CRD の YAML ファイルの例

```
apiVersion: apiextensions.k8s.io/v1beta1 ❶
kind: CustomResourceDefinition
metadata:
  name: crontabs.stable.example.com ❷
spec:
  group: stable.example.com ❸
  version: v1 ❹
  scope: Namespaced ❺
  names:
    plural: crontabs ❻
```



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

- 1 **apiextensions.k8s.io/v1beta1** API を使用します。
- 2 定義の名前を指定します。これは **group** および **plural** フィールドの値を使用する **<plural-name>.<group>** 形式である必要があります。
- 3 API のグループ名を指定します。API グループは、論理的に関連付けられるオブジェクトのコレクションです。たとえば、**Job** または **ScheduledJob** などのすべてのバッチオブジェクトはバッチ API グループ (**batch.api.example.com** など) である可能性があります。組織の完全修飾ドメイン名 (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。

1.6.1.3. カスタムリソース定義のクラスターロールの作成

クラスター管理者は、既存のクラスタースコープのカスタムリソース定義 (CRD) にパーミッションを付与できます。**admin**、**edit**、および **view** のデフォルトクラスターロールを使用する場合、これらのルールに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 1
metadata:
  name: aggregate-cron-tabs-admin-edit 2
  labels:
    rbac.authorization.k8s.io/aggregate-to-admin: "true" 3
    rbac.authorization.k8s.io/aggregate-to-edit: "true" 4
rules:
- apiGroups: ["stable.example.com"] 5
  resources: ["crontabs"] 6
  verbs: ["get", "list", "watch", "create", "update", "patch", "delete", "deletecollection"] 7
---
kind: ClusterRole
apiVersion: rbac.authorization.k8s.io/v1
metadata:
  name: aggregate-cron-tabs-view 8
  labels:
    # Add these permissions to the "view" default role.
    rbac.authorization.k8s.io/aggregate-to-view: "true" 9
    rbac.authorization.k8s.io/aggregate-to-cluster-reader: "true" 10
rules:
- apiGroups: ["stable.example.com"] 11
  resources: ["crontabs"] 12
  verbs: ["get", "list", "watch"] 13
```

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

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

2. クラスターロールを作成します。

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

1.6.1.4. ファイルからのカスタムリソースの作成

カスタムリソース定義 (CRD) がクラスターに追加された後に、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
```

- 1 CRD からグループ名および API バージョン (name/version) を指定します。
- 2 CRD にタイプを指定します。

- 3 オブジェクトの名前を指定します。
- 4 オブジェクトの **ファイナライザー** を指定します (ある場合)。ファイナライザーは、コントローラーがオブジェクトの削除前に完了する必要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
```

1.6.1.5. カスタムリソースの検査

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

前提条件

- CR オブジェクトがアクセスできる namespace にあること。

手順

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

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

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。

1.6.2. カスタムリソース定義からのリソースの管理

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

1.6.2.1. カスタムリソース定義

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

1.6.2.2. ファイルからのカスタムリソースの作成

カスタムリソース定義 (CRD) がクラスターに追加された後に、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
```

1.6.2.3. カスタムリソースの検査

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

前提条件

- CR オブジェクトがアクセスできる namespace にあること。

手順

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

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

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 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2章 ユーザータスク

2.1. 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。

2.1.1. 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 4.5 クラスターへのアクセス
- 管理者によってクラスターにすでにインストールされている etcd Operator

手順

1. この手順を実行するために OpenShift Container Platform Web コンソールで新規プロジェクトを作成します。この例では、**my-etcd** というプロジェクトを使用します。
2. **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、**Copied** をクリックしてから、etcd Operator をクリックして詳細情報および選択可能なアクションを表示します。
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 New** をクリックします。
 - b. 次の画面では、クラスターのサイズなど **EtcdCluster** オブジェクトのテンプレートを起動する最小条件への変更を加えることができます。ここでは **Create** をクリックして確定します。これにより、Operator がトリガーされ、Pod、サービス、および新規 etcd クラスターの他のコンポーネントが起動します。
5. **Resources** タブをクリックして、プロジェクトに Operator によって自動的に作成され、設定された数多くのリソースが含まれることを確認します。

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

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

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

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

2.2. 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。

2.2.1. 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。

2.2.2. OperatorHub を使用した Operator のインストール

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) と連携して機能し、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](#)

2.2.3. 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 Operator を選択すると、Red Hat がコミュニティ 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。

- 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。

- 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。

2.2.4. 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。

- 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
```

- 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。

- 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>
```

- OperatorGroup** オブジェクトを作成します。

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

- 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
```

- 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。

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](#)

2.2.5. 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 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 アップグレードの手動による承認](#)

2.3. OPERATOR LIFECYCLE MANAGER での受付 WEBHOOK の管理

検証用および変更用の受付 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のこれらの Webhook のライフサイクルを管理できます。

2.3.1. CSV での Webhook の定義

ClusterServiceVersion (CSV) リソースには、Operator に同梱される検証用および変更用の Webhook を定義するための **webhookdefinitions** セクションが含まれます。以下は例になります。

検証用の受付 Webhook を含む CSV

```

apiVersion: operators.coreos.com/v1alpha1
kind: ClusterServiceVersion
metadata:
  annotations:
    description: |-
      An example CSV that contains a webhook
  name: example-webhook.v1.0.0
  namespace: placeholder
spec:

```

```

webhookdefinitions:
- generateName: example.webhook.com
  type: ValidatingAdmissionWebhook
  deploymentName: "example-webhook-deployment"
  containerPort: 443
  sideEffects: "None"
  failurePolicy: "Ignore"
  admissionReviewVersions:
  - "v1"
  - "v1beta1"
  rules:
  - operations:
    - "CREATE"
    apiGroups:
    - ""
    apiVersions:
    - "v1"
    resources:
    - "configmaps"
  objectSelector:
    foo: bar
  webhookPath: "/validate"
...

```

Operator Lifecycle Manager (OLM) では、以下を定義する必要があります。

- **type** フィールドは **ValidatingAdmissionWebhook** または **MutatingAdmissionWebhook** 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 Deployment 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。

2.3.2. Webhook の考慮事項

Operator Lifecycle Manager (OLM) によって管理される受付 Webhook を開発する場合、以下の制約を考慮してください。

認証局についての制約

OLM は、各デプロイメントに単一の認証局 (CA) を提供するように設定されます。CA を生成してデプロイメントにマウントするロジックは、元々 API サービスのライフサイクルロジックで使用されていました。結果は、以下のようになります。

- TLS 証明書ファイルは、**/apiserver.local.config/certificates/apiserver.crt** にあるデプロイメントにマウントされます。
- TLS キーファイルは、**/apiserver.local.config/certificates/apiserver.key** にあるデプロイメントにマウントされます。

受付 Webhook ルールについての制約

Operator がクラスターをリカバリー不可能な状態に設定しないようにするため、OLM は受付 Webhook に定義されたルールが以下の要求のいずれかをインターセプトする場合に、失敗フェーズに CSV を配置します。

- すべてのグループをターゲットとする要求
- **operators.coreos.com** グループをターゲットとする要求
- **ValidatingWebhookConfigurations** または **MutatingWebhookConfigurations** リソースをターゲットとする要求

2.3.3. 追加リソース

- [Webhook 受付プラグインのタイプ](#)

第3章 管理者タスク

3.1. 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。

3.1.1. OperatorHub を使用した Operator のインストール

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) と連携して機能し、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](#)

3.1.2. 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。

3.1.3. 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

- 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。

- 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
```

- ❶ **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。

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](#)

3.1.4. 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 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 アップグレードの手動による承認](#)

3.2. インストールされた OPERATOR のアップグレード

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 をアップグレードできます。

3.2.1. Operator の更新チャネルの変更

インストールされた Operator の Subscription は、Operator の更新を追跡し、受信するために使用される更新チャネルを指定します。Operator をアップグレードして新規チャネルからの更新の追跡および受信を開始するために、Subscription で更新チャネルを変更できます。

Subscription の更新チャネルの名前は Operator 間で異なる可能性があります。指定された Operator 内の一般的な規則に従う必要があります。命名スキームには以下が含まれます。

- **4.3、4.4、4.5**
- **stable-4.3、stable-4.4、stable-4.5**
- **alpha、beta、stable、latest**

または、チャネル名は Operator によって提供されるアプリケーションのバージョン番号に従っている可能性があります。



注記

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

サブスクリプションの承認戦略が **Automatic** に設定されている場合、アップグレードプロセスは、選択したチャネルで新規 Operator バージョンが利用可能になるとすぐに開始します。承認戦略が **Manual** に設定されている場合、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. 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 Subscription の場合、アップグレードは自動的に開始します。**Operators → Installed Operators** ページに戻り、アップグレード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 Subscription の場合、Subscription タブからアップグレードを手動で承認できます。

3.2.2. 保留中の Operator アップグレードの手動による承認

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。アップグレードする Operator の名前をクリックします。
3. **Subscription** タブをクリックします。アップグレードの承認を必要とするアップグレードは、**Upgrade Status** の横に表示されます。たとえば、**1 requires approval**が表示される可能性があります。
4. **1 requires approval** をクリックしてから、**Preview Install Plan** をクリックします。
5. アップグレード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、アップグレード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。

3.3. 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。

3.3.1. 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** ドロップダウンメニューから **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。

Operator、Operator デプロイメントおよび Pod はこのアクションで削除されます。CRD および CR を含む Operator によって管理される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。

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

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


出力例

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

- 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
```

- 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
```

3.4. OPERATOR LIFECYCLE MANAGER でのプロキシサポートの設定

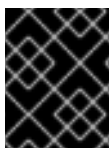
グローバルプロキシが OpenShift Container Platform クラスターで設定されている場合、Operator Lifecycle Manager (OLM) はクラスター全体のプロキシで管理する Operator を自動的に設定します。ただし、インストールされた Operator をグローバルプロキシを上書きするか、またはカスタム CA 証明書を挿入するように設定することもできます。

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)

3.4.1. 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** 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. 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リブされた 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@sha256:66a37fd61a06a43969854ee6d3e21088a98b93838e284a6086b13917f96b0
d9c
...
```

3.4.2. 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。

```
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。
- ❺ **tls-ca-bundle.pem** は設定マップパスとして必要になります。
- ❻ **trusted-ca** ボリュームマウントを作成します。

3.5. 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。

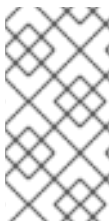
3.5.1. Operator サブスクリプションの状態のタイプ

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3.1 サブスクリプションの状態のタイプ

| 状態 | 説明 |
|----|----|
|----|----|

| 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。

3.5.2. 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** オブジェクトはありません。

3.6. 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 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。

クラスター管理者は、Operator がクラスタースコープの権限を実行できず、ユーザーが OLM を使用して権限をエスカレートできないようにするよう対策を取る必要があります。これを制限する方法として、クラスター管理者は Operator をクラスターに追加される前に監査する必要があります。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。

Operator グループ を、その権限のセットが付与されたサービスアカウントセットに関連付けることにより、クラスター管理者は Operator にポリシーを設定して、それらが RBAC ルールを使用して事前に決定された境界内でのみ動作するようにできます。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。

クラスター管理者以外のユーザーによるこの自己完結型の、スコープが制限された Operator のインストールによって、より多くのユーザーがさらに多くの Operator Framework ツールを利用でき、Operator によるアプリケーションのビルドのエクスペリエンスが強化されます。

3.6.1. Operator インストールポリシーについて

Operator Lifecycle Manager (OLM) を使用すると、クラスター管理者は Operator グループに関連付けられたすべての Operator がデプロイされ、サービスアカウントに付与される権限に基づいてデプロイされ、実行されるように Operator グループのサービスアカウントを指定できます。

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。

指定したサービスアカウントがインストールまたはアップグレードされる Operator についての適切なパーミッションを持たない場合、便利なコンテキスト情報がそれぞれのリソースのステータスに追加されます。これにより、管理者が問題のトラブルシューティングおよび解決が容易になります。

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、インストールは適切なエラーを出して失敗します。

3.6.1.1. 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。

3.6.1.2. 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。

3.6.2. 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 ですべてのことに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。

3.6.2.1. 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て、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"] ①
  resources: ["deployments"]
  verbs: ["list", "watch", "get", "create", "update", "patch", "delete"]
- apiGroups: [""] ②
  resources: ["pods"]
  verbs: ["list", "watch", "get", "create", "update", "patch", "delete"]
```

① ② 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 ①
rules:
- apiGroups: [""]
  resources: ["secrets"]
  verbs: ["get"]
---
kind: Role
rules:
- apiGroups: [""]
  resources: ["secrets"]
  verbs: ["create", "update", "patch"]
```

① シークレットを OLM namespace から取得するために必要です。

3.6.3. 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。

3.7. カスタムカタログの管理

以下では、OpenShift Container Platform で Operator Lifecycle Manager (OLM) の Package Manifest Format または Bundle Format のいずれかを使用してパッケージ化されたカスタムカタログを使用する方法について説明します。

3.7.1. Package Manifest Format を使用したカスタムカタログ

3.7.1.1. Operator カタログイメージについて

Operator Lifecycle Manager (OLM) は常に Operator カタログの最新バージョンから Operator をインストールします。OpenShift Container Platform 4.5 では、Red Hat が提供する Operator は、quay.io から Quay App Registry カタログ経由で配布されます。

表3.2 Red Hat が提供する App Registry カタログ

| カタログ | 説明 |
|----------------------------|--|
| redhat-operators | Red Hat によってパッケージ化され、出荷される Red Hat 製品のパブリックカタログ。Red Hat によってサポートされます。 |
| certified-operators | 大手独立系ソフトウェアベンダー (ISV) の製品のパブリックカタログ。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。 |
| community-operators | operator-framework/community-operators GitHub リポジトリに関連するエンティティによってメンテナンスされる、オプションで表示可能になるソフトウェアのパブリックカタログ。正式なサポートはありません。 |

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。この動作により、再現可能なインストールを維持することが徐々に難しくな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、quay.io からカタログに直接アクセスすることはできません。

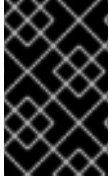
oc adm catalog build コマンドを使用して、クラスター管理者は **Operator カタログイメージ** を作成できます。以下は Operator カタログイメージの説明です。

- App Registry タイプカタログのコンテンツの特定の時点のエクスポート。
- App Registry カタログをコンテナイメージタイプカタログに変換した結果。
- イミュータブルなアーティファクト。

Operator カタログイメージを作成する方法は、前述の問題を引き起こさずにこのコンテンツを使用できる簡単な方法です。

3.7.1.2. Operator カタログイメージのビルド

クラスター管理者は、Operator Lifecycle Manager (OLM) によって使用される Package Manifest Format に基づいてカスタム Operator カタログイメージをビルドできます。カタログイメージは、[Docker v2-2](#) をサポートするコンテナイメージレジストリーにプッシュできます。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。

以下の例では、お使いのネットワークとインターネットの両方にアクセスできるミラーレジストリーを使用することを前提としています。



注記

Windows および macOS のバージョンは **oc adm catalog build** コマンドを提供しないため、この手順では **oc** クライアントの Linux バージョンのみを使用できます。

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
- **oc** バージョン 4.3.5+ Linux クライアント
- **podman** version 1.4.4+
- [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
- 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、以下ようになります。

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

- [quay.io](#) アカウントがアクセスできるプライベート namespace を使用している場合、Quay 認証トークンを設定する必要があります。[quay.io](#) 認証情報を使用してログイン API に対して要求を行うことにより、**--auth-token** フラグで利用できる **AUTH_TOKEN** 環境変数を設定します。

```
$ AUTH_TOKEN=$(curl -sH "Content-Type: application/json" \
  -XPOST https://quay.io/cnr/api/v1/users/login -d '
  {
    "user": {
      "username": "<quay_username>",
      "password": "<quay_password>"
    }
  }' | jq -r '.token')
```

手順

1. 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、ターゲットミラーレジストリーを使用して認証を行います。

```
$ podman login <registry_host_name>
```

また、ビルド時にベースイメージをプルできるように、**registry.redhat.io** で認証します。

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

2. Quay.io から **redhat-operators** カタログをベースにカタログイメージをビルドし、そのイメージにタグを付け、ミラーレジストリーにプッシュします。

```
$ oc adm catalog build \
  --appregistry-org redhat-operators \ ❶
  --from=registry.redhat.io/openshift4/ose-operator-registry:v4.5 \ ❷
  --filter-by-os="linux/amd64" \ ❸
  --to=<registry_host_name>:<port>/olm/redhat-operators:v1 \ ❹
  [-a ${REG_CREDS}] \ ❺
  [--insecure] \ ❻
  [--auth-token "${AUTH_TOKEN}"] ❼
```

- ❶ App Registry インスタンスからのプルに使用する組織 (namespace)。
- ❷ 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して、**--from** を **ose-operator-registry** ベースイメージに設定します。
- ❸ **--filter-by-os** を、ターゲットの OpenShift Container Platform クラスターと一致する必要のある、ベースイメージに使用するオペレーティングシステムおよびアーキテクチャーに設定します。使用できる値は、**linux/amd64**、**linux/ppc64le**、および **linux/s390x** です。
- ❹ カatalogイメージに名前を付け、**v1** などのタグを追加します。
- ❺ 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。
- ❻ 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。
- ❼ オプション: 公開されていない他のアプリケーションレジストリーカタログが使用されている場合、Quay 認証トークンを指定します。

出力例

```
INFO[0013] loading Bundles
dir=/var/folders/st/9cskxqs53ll3wdn434vw4cd80000gn/T/300666084/manifests-829192605
...
Pushed sha256:f73d42950021f9240389f99ddc5b0c7f1b533c054ba344654ff1edaf6bf827e3
to example_registry:5000/olm/redhat-operators:v1
```

無効なマニフェストが Red Hat のカタログに誤って導入されることがあります。これが実際に生じる場合には、以下のようなエラーが表示される可能性があります。

エラーのある出力の例

```
...
INFO[0014] directory
dir=/var/folders/st/9cskxqs53ll3wdn434vw4cd80000gn/T/300666084/manifests-829192605
file=4.2 load=package
W1114 19:42:37.876180 34665 builder.go:141] error building database: error loading
package into db: fuse-camel-k-operator.v7.5.0 specifies replacement that couldn't be found
Uploading ... 244.9kB/s
```

通常、これらのエラーは致命的なエラーではなく、該当する Operator パッケージにインストールする予定の Operator やその依存関係が含まれない場合、それらを見捨てるすることができます。

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](#)

3.7.1.3. Operator カタログイメージのミラーリング

クラスター管理者はそれぞれのカタログの内容をレジストリーにミラーリングし、CatalogSource を使用してコンテンツを OpenShift Container Platform クラスターに読み込むことができます。この例では、以前にビルドされ、サポートされているレジストリーにプッシュされたカスタム **redhat-operators** カタログイメージを使用します。

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
- サポートされているレジストリーにプッシュされるカスタム Operator カタログイメージ
- **oc** version 4.3.5+
- **podman** version 1.4.4+
- [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
- 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、以下ようになります。

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

手順

1. **oc adm catalog mirror** コマンドは、カスタム Operator カタログ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。以下のいずれかを選択できます。
 - コマンドのデフォルト動作で、マニフェストの生成後にすべてのイメージコンテンツをミラーレジストリーに自動的にミラーリングできるようにします。または、
 - **--manifests-only** フラグを追加して、ミラーリングに必要なマニフェストのみを生成しますが、これにより、イメージコンテンツがレジストリーに自動的にミラーリングされる訳ではありません。これは、ミラーリングする内容を確認するのに役立ちます。また、コンテンツ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、そのファイルを **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。

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ます。

```
$ oc adm catalog mirror \
  <registry_host_name>:<port>/olm/redhat-operators:v1 \ ❶
  <registry_host_name>:<port> \
  [-a ${REG_CREDS}] \ ❷
  [--insecure] \ ❸
  --filter-by-os='*' \ ❹
  [--manifests-only] ❺
```

- ❶ Operator カタログイメージを指定します。
- ❷ 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。
- ❸ 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。
- ❹ このフラグは、現時点で複数のアーキテクチャーのサポートに関して問題が存在するため必要になります。
- ❺ 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。



警告

--filter-by-os フラグが設定されていない状態か、または `*` 以外の値に設定されている場合、コマンドが複数の異なるアーキテクチャーをフィルタし、マニフェスト一覧のダイジェスト (**multi-arch image** イメージとしても知られる) が変更されます。ダイジェストが間違っていると、それらのイメージおよび Operator の非接続クラスターでのデプロイメントに失敗します。詳細は、[BZ#1890951](#) を参照してください。

出力例

```
using database path mapping: /tmp/190214037
wrote database to /tmp/190214037
using database at: /tmp/190214037/bundles.db ❶
...
```

- ❶ コマンドで生成される一時的なデータベース。

コマンドの実行後に、**<image_name>-manifests/** ディレクトリーが現在のディレクトリーに作成され、以下のファイルが生成されます。

- 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。

- **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。
2. 直前の手順で **--manifests-only** フラグを使用して、コンテンツのサブセットのみをミラーリングする場合は、以下を実行します。
 - a. **mapping.txt** ファイルのイメージの一覧を仕様に変更します。ミラーリングするイメージのサブセットの名前とバージョンが不明な場合は、以下の手順で確認します。
 - i. **oc adm catalog mirror** コマンドで生成された一時的なデータベースに対して **sqlite3** ツールを実行し、一般的な検索クエリーに一致するイメージの一覧を取得します。出力は、後に **mapping.txt** ファイルを編集する方法を通知するのに役立ちます。たとえば、**clusterlogging.4.3** の文字列のようなイメージの一覧を取得するには、以下を実行します。

```
$ echo "select * from related_image \
      where operatorbundle_name like 'clusterlogging.4.3%';" \
      | sqlite3 -line /tmp/190214037/bundles.db 1
```

- 1 **oc adm catalog mirror** コマンドの直前の出力を参照し、データベースファイルのパスを見つけます。

出力例

```
image = registry.redhat.io/openshift4/ose-logging-
kibana5@sha256:aa4a8b2a00836d0e28aa6497ad90a3c116f135f382d8211e3c55f34f
b36dfe61
operatorbundle_name = clusterlogging.4.3.33-202008111029.p0

image = registry.redhat.io/openshift4/ose-oauth-
proxy@sha256:6b4db07f6e6c962fc96473d86c44532c93b146bbefe311d0c348117bf75
9c506
operatorbundle_name = clusterlogging.4.3.33-202008111029.p0
...
```

- ii. 直前の手順で取得した結果を使用して **mapping.txt** ファイルを編集し、ミラーリングする必要のあるイメージのサブセットのみを追加します。たとえば、前述の出力例の **image** 値を使用して、**mapping.txt** ファイルに以下の一致する行が存在することを確認できます。

mapping.txt の一致するイメージマッピング。

```
registry.redhat.io/openshift4/ose-logging-
kibana5@sha256:aa4a8b2a00836d0e28aa6497ad90a3c116f135f382d8211e3c55f34f
b36dfe61=<registry_host_name>:<port>/openshift4-ose-logging-kibana5:a767c8f0
registry.redhat.io/openshift4/ose-oauth-
proxy@sha256:6b4db07f6e6c962fc96473d86c44532c93b146bbefe311d0c348117bf75
9c506=<registry_host_name>:<port>/openshift4-ose-oauth-proxy:3754ea2b
```

この例では、これらのイメージのみをミラーリングする場合に、**mapping.txt** ファイルの他のすべてのエントリーを削除し、上記の 2 行のみを残します。

- b. ネットワークアクセスが無制限のワークステーション上で、変更した **mapping.txt** ファイルを使用し、**oc image mirror** コマンドを使用してイメージをレジストリーにミラーリングします。

```
$ oc image mirror \
  [-a ${REG_CREDS}] \
  --filter-by-os='.*' \
  -f ./redhat-operators-manifests/mapping.txt
```



警告

--filter-by-os フラグが設定されていない状態か、または **.*** 以外の値に設定されている場合、コマンドが複数の異なるアーキテクチャーをフィルターし、マニフェスト一覧のダイジェスト (**multi-arch image** イメージとしても知られる) が変更されます。ダイジェストが間違っていると、それらのイメージおよび Operator の非接続クラスターでのデプロイメントに失敗します。

3. **ImageContentSourcePolicy** オブジェクトを適用します。

```
$ oc apply -f ./redhat-operators-manifests/imageContentSourcePolicy.yaml
```

4. カタログ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
spec:
  sourceType: grpc
  image: <registry_host_name>:<port>/olm/redhat-operators:v1 1
  displayName: My Operator Catalog
  publisher: grpc
```

- 1** Operator カタログイメージを指定します。

- 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 create -f catalogsource.yaml
```

5. 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 |
|------|---------------------|-----|
| etcd | My Operator Catalog | 34s |

ネットワークが制限された環境の OpenShift Container Platform クラスター Web コンソールで、**OperatorHub** ページから Operator をインストールできます。

追加リソース

- [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](#)

3.7.1.4. Operator カタログイメージの更新

クラスター管理者がカスタム Operator カタログイメージを使用するように OperatorHub を設定した後、管理者は Red Hat の App Registry カタログに追加された更新をキャプチャーして、OpenShift Container Platform クラスターを最新の Operator と共に最新の状態に保つことができます。これは、新規 Operator カタログイメージをビルドし、プッシュしてから、既存の **CatalogSource** オブジェクトの **spec.image** パラメーターを新規イメージダイジェストに置き換えることによって実行されます。

この例では、カスタムの **redhat-operators** カタログイメージが OperatorHub と使用するように設定されていることを前提としています。



注記

Windows および macOS のバージョンは **oc adm catalog build** コマンドを提供しないため、この手順では **oc** クライアントの Linux バージョンのみを使用できます。

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
- **oc** バージョン 4.3.5+ Linux クライアント

- **podman** version 1.4.4+
- [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
- カスタムカタログイメージを使用するように設定されている OperatorHub
- 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、以下ようになります。

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

- [quay.io](#) アカウントがアクセスできるプライベート namespace を使用している場合、Quay 認証トークンを設定する必要があります。[quay.io](#) 認証情報を使用してログイン API に対して要求を行うことにより、**--auth-token** フラグで使用する **AUTH_TOKEN** 環境変数を設定します。

```
$ AUTH_TOKEN=$(curl -sH "Content-Type: application/json" \
  -XPOST https://quay.io/cnr/api/v1/users/login -d '
  {
    "user": {
      "username": "<quay_username>",
      "password": "<quay_password>"
    }
  }' | jq -r '.token')
```

手順

1. 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、ターゲットミラーレジストリーを使用して認証を行います。

```
$ podman login <registry_host_name>
```

また、ビルド時にベースイメージをプルできるように、**registry.redhat.io** で認証します。

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

2. Quay.io から **redhat-operators** カタログをベースに新規カタログイメージをビルドし、そのイメージにタグを付け、ミラーレジストリーにプッシュします。

```
$ oc adm catalog build \
  --appregistry-org redhat-operators \ ❶
  --from=registry.redhat.io/openshift4/ose-operator-registry:v4.5 \ ❷
  --filter-by-os="linux/amd64" \ ❸
  --to=<registry_host_name>:<port>/olm/redhat-operators:v2 \ ❹
  [-a ${REG_CREDS}] \ ❺
  [--insecure] \ ❻
  [--auth-token "${AUTH_TOKEN}"] ❼
```

- ❶ App Registry インスタンスからのプルに使用する組織 (namespace)。
- ❷ 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して、**--from** を **ose-operator-registry** ベースイメージに設定します。

- ③ **--filter-by-os** を、ターゲットの OpenShift Container Platform クラスターと一致する必要のある、ベースイメージに使用するオペレーティングシステムおよびアーキテクチャーに
- ④ カタログイメージに名前を付け、タグを追加します (更新済みのカタログの場合は **v2** などのタグ)。
- ⑤ 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。
- ⑥ 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。
- ⑦ オプション: 公開されていない他のアプリケーションレジストリーカタログが使用されている場合、Quay 認証トークンを指定します。

出力例

```
INFO[0013] loading Bundles
dir=/var/folders/st/9cskxqs53ll3wdn434vw4cd80000gn/T/300666084/manifests-829192605
...
Pushed sha256:f73d42950021f9240389f99ddc5b0c7f1b533c054ba344654ff1edaf6bf827e3
to example_registry:5000/olm/redhat-operators:v2
```

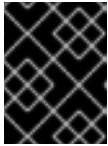
3. カタログのコンテンツをターゲットレジストリーに対してミラーリングします。以下の **oc adm catalog mirror** コマンドは、カスタム Operator カタログイメージのコンテンツを抽出し、ミラーリングに必要なマニフェストを生成し、イメージをレジストリーにミラーリングします。

```
$ oc adm catalog mirror \
  <registry_host_name>:<port>/olm/redhat-operators:v2 ①
  <registry_host_name>:<port> \
  [-a ${REG_CREDS}] ②
  [--insecure] ③
  --filter-by-os='.*' ④
```

- ① 新規の Operator カタログイメージを指定します。
- ② 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。
- ③ 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。
- ④ このフラグは、現時点で複数のアーキテクチャーのサポートに関して問題が存在するため必要になります。**--filter-by-os** フラグが設定されていない状態か、または **.*** 以外の値に設定されている場合、コマンドが複数の異なるアーキテクチャーをフィルターし、マニフェスト一覧のダイジェスト (**multi-arch image** イメージとしても知られる) が変更されます。ダイジェストが間違っていると、それらのイメージおよび Operator の非接続クラスターでのデプロイメントに失敗します。詳細は、[BZ#1890951](#) を参照してください。

4. 新たに生成されたマニフェストを適用します。

```
$ oc apply -f ./redhat-operators-manifests
```

**重要**

imageContentSourcePolicy.yaml マニフェストを適用する必要がある場合があります。ファイルの **diff** を完了して、変更が必要かどうかを判断します。

5. カタログイメージを参照する **CatalogSource** オブジェクトを更新します。

a. この **CatalogSource** オブジェクトの元の **catalogsource.yaml** ファイルがある場合:

i. **catalogsource.yaml** ファイルを編集し、**spec.image** フィールドで新規カタログイメージを参照できるようにします。

```
apiVersion: operators.coreos.com/v1alpha1
kind: CatalogSource
metadata:
  name: my-operator-catalog
  namespace: openshift-marketplace
spec:
  sourceType: grpc
  image: <registry_host_name>:<port>/olm/redhat-operators:v2 1
  displayName: My Operator Catalog
  publisher: grpc
```

1 新規の Operator カタログイメージを指定します。

ii. 更新されたファイルを使用して **CatalogSource** オブジェクトを置き換えます。

```
$ oc replace -f catalogsource.yaml
```

b. または、以下のコマンドを使用してカタログソースを編集し、**spec.image** パラメーターで新規カタログイメージを参照します。

```
$ oc edit catalogsource <catalog_source_name> -n openshift-marketplace
```

更新された Operator は、OpenShift Container Platform クラスターの **OperatorHub** ページから利用できるようになりました。

関連情報

- [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](#)

3.7.1.5. Operator カタログイメージのテスト

Operator カタログイメージのコンテンツは、これをコンテナとして実行し、gRPC API をクエリーして検証できます。イメージをさらにテストするには、**CatalogSource** オブジェクトでイメージを参照して OLM サブスクリプションを解決できます。この例では、以前にビルドされ、サポートされているレジストリーにプッシュされたカスタム **redhat-operators** カタログイメージを使用します。

前提条件

- サポートされているレジストリーにプッシュされるカスタム Operator カタログイメージ
- **podman** version 1.4.4+

- **oc** version 4.3.5+
- [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
- [grpcurl](#)

手順

1. Operator カタログイメージをプルします。

```
$ podman pull <registry_host_name>:<port>/olm/redhat-operators:v1
```

2. イメージを実行します。

```
$ podman run -p 50051:50051 \
  -it <registry_host_name>:<port>/olm/redhat-operators:v1
```

3. **grpcurl** を使用して利用可能なパッケージの実行中のイメージをクエリーします。

```
$ grpcurl -plaintext localhost:50051 api.Registry/ListPackages
```

出力例

```
{
  "name": "3scale-operator"
}
{
  "name": "amq-broker"
}
{
  "name": "amq-online"
}
```

4. チャネルの最新の Operator バンドルを取得します。

```
$ grpcurl -plaintext -d '{"pkgName":"kiali-oss","channelName":"stable"}' localhost:50051
api.Registry/GetBundleForChannel
```

出力例

```
{
  "csvName": "kiali-operator.v1.0.7",
  "packageName": "kiali-oss",
  "channelName": "stable",
  ...
}
```

5. イメージのダイジェストを取得します。

```
$ podman inspect \
  --format='{{index .RepoDigests 0}}' \
  <registry_host_name>:<port>/olm/redhat-operators:v1
```

出力例


```
example_registry:5000/olm/redhat-
operators@sha256:f73d42950021f9240389f99ddc5b0c7f1b533c054ba344654ff1edaf6bf827e3
```

- Operator グループが Operator とその依存関係をサポートする namespace **my-ns** にあることを前提とし、イメージダイジェストを使用して **CatalogSource** オブジェクトを作成します。以下に例を示します。

```
apiVersion: operators.coreos.com/v1alpha1
kind: CatalogSource
metadata:
  name: custom-redhat-operators
  namespace: my-ns
spec:
  sourceType: grpc
  image: example_registry:5000/olm/redhat-
operators@sha256:f73d42950021f9240389f99ddc5b0c7f1b533c054ba344654ff1edaf6bf827e3

  displayName: Red Hat Operators
```

- カタログイメージから、利用可能な最新の **servicemeshoperator** およびその依存関係を解決するサブスクリプションを作成します。

```
apiVersion: operators.coreos.com/v1alpha1
kind: Subscription
metadata:
  name: servicemeshoperator
  namespace: my-ns
spec:
  source: custom-redhat-operators
  sourceNamespace: my-ns
  name: servicemeshoperator
  channel: "1.0"
```

3.7.2. Bundle Format を使用したカスタムカタログ

3.7.2.1. opm CLI

新規の **opm** CLI ツールが新規の Bundle Format と共に導入されます。このツールを使用して、リポジトリに相当する **インデックス** 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、その後には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、OLM はインデックスイメージを CatalogSource 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。

追加リソース

- Operator SDK を使用してバンドルイメージを作成するには、[バンドルイメージの使用](#) について参照してください。

3.7.2.2. opm のインストール

opm CLI ツールはワークステーションにインストールできます。

前提条件

- **podman** version 1.4.4+

手順

1. 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
```

2. **registry.redhat.io** で認証します。

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

3. Operator レジストリーイメージから **opm** バイナリーを抽出し、これをローカルファイルシステムにコピーします。

```
$ oc image extract registry.redhat.io/openshift4/ose-operator-registry:v4.5 \
  -a ${REG_CREDS} \ ❶
  --path /usr/bin/opm:. \
  --confirm
```

- ❶ レジストリー認証情報ファイルの場所を指定します。

4. バイナリーを実行可能にします。

```
$ chmod +x ./opm
```

5. ファイルを **PATH** の任意の場所に置きます (例: **/usr/local/bin/**)。

```
$ sudo mv ./opm /usr/local/bin/
```

6. クライアントが正しくインストールされていることを確認します。

```
$ opm version
```

出力例

```
Version: version.Version{OpmVersion:"1.12.3", GitCommit:"", BuildDate:"2020-07-01T23:18:58Z", GoOs:"linux", GoArch:"amd64"}
```

3.7.2.3. 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 1.4.4+
- 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。

手順

1. 新しいインデックスを開始します。

```
$ opm index add \
  --bundles quay.io/<namespace>/test-operator:v0.1.0 \ 1
  --tag quay.io/<namespace>/test-catalog:latest \ 2
  [--binary-image <registry_base_image>] 3
```

- 1** インデックスに追加するバンドルイメージのコンマ区切りの一覧。
- 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。

```
$ podman push quay.io/<namespace>/test-catalog:latest
```

3.7.2.4. 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カタログを作成し、これを OpenShift Container Platform クラスターに適用することができます。

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. **CatalogSource** オブジェクトをインデックスを参照するクラスターに適用します。

```
$ cat <<EOF | oc apply -f -

apiVersion: operators.coreos.com/v1alpha1
kind: CatalogSource
metadata:
  name: test-catalog
  namespace: openshift-marketplace
spec:
  sourceType: grpc
  image: quay.io/<namespace>/test-catalog:latest 1
  displayName: Test Catalog
  updateStrategy:
    registryPoll: 2
    interval: 30m
EOF
```

1. インデックスイメージを指定します。
 2. 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。
2. OpenShift Container Platform Web コンソールまたは CLI を使用して、カタログが正常に読み込まれ、パッケージが利用可能であることを確認します。たとえば、CLI を使用します。

- a. Pod を確認します。

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

- b. カタログソースを確認します。

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

- c. パッケージマニフェストを確認します。

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

3.7.2.5. インデックスイメージの更新

opm CLI を使用して既存の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 1.4.4+
- 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。
- クラスタに作成され、適用される **CatalogSource** オブジェクト。

手順

1. 既存のインデックスを更新します。

```
$ opm index add \
  --bundles quay.io/<namespace>/another-operator:v1 \ 1
  --from-index quay.io/<namespace>/test-catalog:latest \ 2
  --tag quay.io/<namespace>/test-catalog:latest 3
```

1. インデックスに追加する追加のバンドルイメージ。
2. 以前にプッシュされた既存のインデックス。
3. 更新されたインデックスイメージに含めるイメージタグ。

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 quay.io/<namespace>/test-catalog:latest
```

3. Operator Lifecycle Manager (OLM) が一定の間隔でインデックスイメージを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
```

3.8. 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 Quay.io 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 OperatorHub ソースのローカルミラーを準備するために使用され、リモート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。

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ます。
- OLM を、デフォルトのリモートソースからではなく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。

非接続モードでの実行をサポートする Red Hat Operator の一覧については、以下の Red Hat ナレッジベースの記事を参照してください。

<https://access.redhat.com/articles/4740011>

追加リソース

- [ネットワークが制限された環境についての Operator の有効化](#)

3.8.1. Operator カタログイメージについて

Operator Lifecycle Manager (OLM) は常に Operator カタログの最新バージョンから Operator をインストールします。OpenShift Container Platform 4.5 では、Red Hat が提供する Operator は、quay.io から Quay App Registry カタログ経由で配布されます。

表3.3 Red Hat が提供する App Registry カタログ

| カタログ | 説明 |
|----------------------------|--|
| redhat-operators | Red Hat によってパッケージ化され、出荷される Red Hat 製品のパブリックカタログ。Red Hat によってサポートされます。 |
| certified-operators | 大手独立系ソフトウェアベンダー (ISV) の製品のパブリックカタログ。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。 |
| community-operators | operator-framework/community-operators GitHub リポジトリで関連するエンティティによってメンテナンスされる、オプションで表示可能になるソフトウェアのパブリックカタログ。正式なサポートはありません。 |

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。この動作により、再現可能なインストールを維持することが徐々に難しくな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、quay.io からカタログに直接アクセスすることはできません。

oc adm catalog build コマンドを使用して、クラスター管理者は **Operator カタログイメージ**を作成できます。以下は Operator カタログイメージの説明です。

- App Registry タイプカタログのコンテンツの特定の時点のエクスポート。
- App Registry カタログをコンテナイメージタイプカタログに変換した結果。
- イミュータブルなアーティファクト。

Operator カタログイメージを作成する方法は、前述の問題を引き起こさずにこのコンテンツを使用できる簡単な方法です。

3.8.2. Operator カタログイメージのビルド

クラスター管理者は、Operator Lifecycle Manager (OLM) によって使用される Package Manifest Format に基づいてカスタム Operator カタログイメージをビルドできます。カタログイメージは、[Docker v2-2](https://docs.docker.com/engine/release-notes/2.2/) をサポートするコンテナイメージレジストリーにプッシュできます。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。

以下の例では、お使いのネットワークとインターネットの両方にアクセスできるミラーレジストリーを使用することを前提としています。



注記

Windows および macOS のバージョンは **oc adm catalog build** コマンドを提供しないため、この手順では **oc** クライアントの Linux バージョンのみを使用できます。

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
- **oc** バージョン 4.3.5+ Linux クライアント
- **podman** version 1.4.4+
- [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
- 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
```

- [quay.io](#) アカウントがアクセスできるプライベート namespace を使用している場合、Quay 認証トークンを設定する必要があります。[quay.io](#) 認証情報を使用してログイン API に対して要求を行うことにより、**--auth-token** フラグで使用できる **AUTH_TOKEN** 環境変数を設定します。

```
$ AUTH_TOKEN=$(curl -sH "Content-Type: application/json" \
-XPOST https://quay.io/cnr/api/v1/users/login -d '
{
  "user": {
    "username": "<quay_username>",
    "password": "<quay_password>"
  }
}' | jq -r '.token')
```

手順

1. 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、ターゲットミラーレジストリーを使用して認証を行います。

```
$ podman login <registry_host_name>
```

また、ビルド時にベースイメージをプルできるように、**registry.redhat.io** で認証します。

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

2. Quay.io から **redhat-operators** カタログをベースにカタログイメージをビルドし、そのイメージにタグを付け、ミラーレジストリーにプッシュします。

```
$ oc adm catalog build \
--appregistry-org redhat-operators \ 1
```

```
--from=registry.redhat.io/openshift4/ose-operator-registry:v4.5 \ ❷
--filter-by-os="linux/amd64" \ ❸
--to=<registry_host_name>:<port>/olm/redhat-operators:v1 \ ❹
[-a ${REG_CREDS}] \ ❺
[--insecure] \ ❻
[--auth-token "${AUTH_TOKEN}"] ❼
```

- ❶ App Registry インスタンスからのプルに使用する組織 (namespace)。
- ❷ 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して、**--from** を **ose-operator-registry** ベースイメージに設定します。
- ❸ **--filter-by-os** を、ターゲットの OpenShift Container Platform クラスターと一致する必要のある、ベースイメージに使用するオペレーティングシステムおよびアーキテクチャーに設定します。使用できる値は、**linux/amd64**、**linux/ppc64le**、および **linux/s390x** です。
- ❹ カタログイメージに名前を付け、**v1** などのタグを追加します。
- ❺ 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。
- ❻ 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。
- ❼ オプション: 公開されていない他のアプリケーションレジストリーカタログが使用されている場合、Quay 認証トークンを指定します。

出力例

```
INFO[0013] loading Bundles
dir=/var/folders/st/9cskxqs53ll3wdn434vw4cd80000gn/T/300666084/manifests-829192605
...
Pushed sha256:f73d42950021f9240389f99ddc5b0c7f1b533c054ba344654ff1edaf6bf827e3
to example_registry:5000/olm/redhat-operators:v1
```

無効なマニフェストが Red Hat のカタログに誤って導入されることがあります。これが実際に生じる場合には、以下のようなエラーが表示される可能性があります。

エラーのある出力の例

```
...
INFO[0014] directory
dir=/var/folders/st/9cskxqs53ll3wdn434vw4cd80000gn/T/300666084/manifests-829192605
file=4.2 load=package
W1114 19:42:37.876180 34665 builder.go:141] error building database: error loading
package into db: fuse-camel-k-operator.v7.5.0 specifies replacement that couldn't be found
Uploading ... 244.9kB/s
```

通常、これらのエラーは致命的なエラーではなく、該当する Operator パッケージにインストールする予定の Operator やその依存関係が含まれない場合、それらを見捨てることができます。

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](#)

3.8.3. ネットワークが制限された環境向けの OperatorHub の設定

クラスター管理者は、カスタム Operator カタログイメージを使用し、OLM および OperatorHub をネットワークが制限された環境でローカルコンテンツを使用するように設定できます。この例では、以前にビルドされ、サポートされているレジストリーにプッシュされたカスタム **redhat-operators** カタログイメージを使用します。

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
- サポートされているレジストリーにプッシュされるカスタム Operator カタログイメージ
- **oc** version 4.3.5+
- **podman** version 1.4.4+
- [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
- 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. **disableAllDefaultSources: true** を仕様に追加してデフォルトの **OperatorSource** オブジェクトを無効にします。

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

これにより、OpenShift Container Platform のインストール時にデフォルトで設定されるデフォルトソースが無効になります。

2. **oc adm catalog mirror** コマンドは、カスタム Operator カタログ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。以下のいずれかを選択できます。
 - コマンドのデフォルト動作で、マニフェストの生成後にすべてのイメージコンテンツをミラーレジストリーに自動的にミラーリングできるようにします。または、
 - **--manifests-only** フラグを追加して、ミラーリングに必要なマニフェストのみを生成しますが、これにより、イメージコンテンツがレジストリーに自動的にミラーリングされる訳ではありません。これは、ミラーリングする内容を確認するのに役立ちます。また、コンテンツ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、そのファイルを **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。

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ます。

```
$ oc adm catalog mirror \
  <registry_host_name>:<port>/olm/redhat-operators:v1 1
```



```
<registry_host_name>:<port> \
[-a ${REG_CREDS}] \ ❷
[--insecure] \ ❸
--filter-by-os='*' \ ❹
[--manifests-only] ❺
```

- ❶ Operator カタログイメージを指定します。
- ❷ 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。
- ❸ 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。
- ❹ このフラグは、現時点で複数のアーキテクチャーのサポートに関して問題が存在するため必要になります。
- ❺ 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。



警告

--filter-by-os フラグが設定されていない状態か、または `.*` 以外の値に設定されている場合、コマンドが複数の異なるアーキテクチャーをフィルターし、マニフェスト一覧のダイジェスト (**multi-arch image** イメージとしても知られる) が変更されます。ダイジェストが間違っていると、それらのイメージおよび Operator の非接続クラスターでのデプロイメントに失敗します。詳細は、[BZ#1890951](#) を参照してください。

出力例

```
using database path mapping: /tmp/190214037
wrote database to /tmp/190214037
using database at: /tmp/190214037/bundles.db ❶
...
```

- ❶ コマンドで生成される一時的なデータベース。

コマンドの実行後に、**<image_name>-manifests/** ディレクトリーが現在のディレクトリーに作成され、以下のファイルが生成されます。

- 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。
- **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。

3. 直前の手順で **--manifests-only** フラグを使用して、コンテンツのサブセットのみをミラーリングする場合は、以下を実行します。
 - a. **mapping.txt** ファイルのイメージの一覧を仕様に変更します。ミラーリングするイメージのサブセットの名前とバージョンが不明な場合は、以下の手順で確認します。
 - i. **oc adm catalog mirror** コマンドで生成された一時的なデータベースに対して **sqlite3** ツールを実行し、一般的な検索クエリーに一致するイメージの一覧を取得します。出力は、後に **mapping.txt** ファイルを編集する方法を通知するのに役立ちます。たとえば、**clusterlogging.4.3** の文字列のようなイメージの一覧を取得するには、以下を実行します。

```
$ echo "select * from related_image \
      where operatorbundle_name like 'clusterlogging.4.3%';" \
      | sqlite3 -line /tmp/190214037/bundles.db ❶
```

- ❶ **oc adm catalog mirror** コマンドの直前の出力を参照し、データベースファイルのパスを見つけます。

出力例

```
image = registry.redhat.io/openshift4/ose-logging-
kibana5@sha256:aa4a8b2a00836d0e28aa6497ad90a3c116f135f382d8211e3c55f34f
b36dfe61
operatorbundle_name = clusterlogging.4.3.33-202008111029.p0

image = registry.redhat.io/openshift4/ose-oauth-
proxy@sha256:6b4db07f6e6c962fc96473d86c44532c93b146bbefe311d0c348117bf75
9c506
operatorbundle_name = clusterlogging.4.3.33-202008111029.p0
...
```

- ii. 直前の手順で取得した結果を使用して **mapping.txt** ファイルを編集し、ミラーリングする必要のあるイメージのサブセットのみを追加します。たとえば、前述の出力例の **image** 値を使用して、**mapping.txt** ファイルに以下の一致する行が存在することを確認できます。

mapping.txt の一致するイメージマッピング。

```
registry.redhat.io/openshift4/ose-logging-
kibana5@sha256:aa4a8b2a00836d0e28aa6497ad90a3c116f135f382d8211e3c55f34f
b36dfe61=<registry_host_name>:<port>/openshift4-ose-logging-kibana5:a767c8f0
registry.redhat.io/openshift4/ose-oauth-
proxy@sha256:6b4db07f6e6c962fc96473d86c44532c93b146bbefe311d0c348117bf75
9c506=<registry_host_name>:<port>/openshift4-ose-oauth-proxy:3754ea2b
```

この例では、これらのイメージのみをミラーリングする場合に、**mapping.txt** ファイルの他のすべてのエントリーを削除し、上記の 2 行のみを残します。

- b. ネットワークアクセスが無制限のワークステーション上で、変更した **mapping.txt** ファイルを使用し、**oc image mirror** コマンドを使用してイメージをレジストリーにミラーリングします。

```
$ oc image mirror \
  [-a ${REG_CREDS}] \
  --filter-by-os='.*' \
  -f ./redhat-operators-manifests/mapping.txt
```



警告

--filter-by-os フラグが設定されていない状態か、または `.*` 以外の値に設定されている場合、コマンドが複数の異なるアーキテクチャーをフィルタし、マニフェスト一覧のダイジェスト (**multi-arch image** イメージとしても知られる) が変更されます。ダイジェストが間違っていると、それらのイメージおよび Operator の非接続クラスターでのデプロイメントに失敗します。

4. **ImageContentSourcePolicy** オブジェクトを適用します。

```
$ oc apply -f ./redhat-operators-manifests/imageContentSourcePolicy.yaml
```

5. カタログ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
spec:
  sourceType: grpc
  image: <registry_host_name>:<port>/olm/redhat-operators:v1 ❶
  displayName: My Operator Catalog
  publisher: grpc
```

- ❶ Operator カタログイメージを指定します。

- 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 create -f catalogsource.yaml
```

6. 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 |
|------|---------------------|-----|
| etcd | My Operator Catalog | 34s |

ネットワークが制限された環境の OpenShift Container Platform クラスター Web コンソールで、**OperatorHub** ページから Operator をインストールできます。

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](#)

3.8.4. Operator カタログイメージの更新

クラスター管理者がカスタム Operator カタログイメージを使用するように OperatorHub を設定した後、管理者は Red Hat の App Registry カタログに追加された更新をキャプチャーして、OpenShift Container Platform クラスターを最新の Operator と共に最新の状態に保つことができます。これは、新規 Operator カタログイメージをビルドし、プッシュしてから、既存の **CatalogSource** オブジェクトの **spec.image** パラメーターを新規イメージダイジェストに置き換えることによって実行されます。

この例では、カスタムの **redhat-operators** カタログイメージが OperatorHub と使用するように設定されていることを前提としています。



注記

Windows および macOS のバージョンは **oc adm catalog build** コマンドを提供しないため、この手順では **oc** クライアントの Linux バージョンのみを使用できます。

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
- **oc** バージョン 4.3.5+ Linux クライアント

- **podman** version 1.4.4+
- [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
- カスタムカタログイメージを使用するように設定されている OperatorHub
- 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、以下ようになります。

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

- [quay.io](#) アカウントがアクセスできるプライベート namespace を使用している場合、Quay 認証トークンを設定する必要があります。[quay.io](#) 認証情報を使用してログイン API に対して要求を行うことにより、**--auth-token** フラグで使用する **AUTH_TOKEN** 環境変数を設定します。

```
$ AUTH_TOKEN=$(curl -sH "Content-Type: application/json" \
  -XPOST https://quay.io/cnr/api/v1/users/login -d '
  {
    "user": {
      "username": "<quay_username>",
      "password": "<quay_password>"
    }
  }' | jq -r '.token')
```

手順

1. 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、ターゲットミラーレジストリーを使用して認証を行います。

```
$ podman login <registry_host_name>
```

また、ビルド時にベースイメージをプルできるように、**registry.redhat.io** で認証します。

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

2. Quay.io から **redhat-operators** カタログをベースに新規カタログイメージをビルドし、そのイメージにタグを付け、ミラーレジストリーにプッシュします。

```
$ oc adm catalog build \
  --appregistry-org redhat-operators \ ❶
  --from=registry.redhat.io/openshift4/ose-operator-registry:v4.5 \ ❷
  --filter-by-os="linux/amd64" \ ❸
  --to=<registry_host_name>:<port>/olm/redhat-operators:v2 \ ❹
  [-a ${REG_CREDS}] \ ❺
  [--insecure] \ ❻
  [--auth-token "${AUTH_TOKEN}"] ❼
```

- ❶ App Registry インスタンスからのプルに使用する組織 (namespace)。
- ❷ 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して、**--from** を **ose-operator-registry** ベースイメージに設定します。

- 3 **--filter-by-os** を、ターゲットの OpenShift Container Platform クラスターと一致する必要のある、ベースイメージに使用するオペレーティングシステムおよびアーキテクチャーに
- 4 カタログイメージに名前を付け、タグを追加します (更新済みのカタログの場合は **v2** などのタグ)。
- 5 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。
- 6 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。
- 7 オプション: 公開されていない他のアプリケーションレジストリーカタログが使用されている場合、Quay 認証トークンを指定します。

出力例

```
INFO[0013] loading Bundles
dir=/var/folders/st/9cskxqs53ll3wdn434vw4cd80000gn/T/300666084/manifests-829192605
...
Pushed sha256:f73d42950021f9240389f99ddc5b0c7f1b533c054ba344654ff1edaf6bf827e3
to example_registry:5000/olm/redhat-operators:v2
```

3. カタログのコンテンツをターゲットレジストリーに対してミラーリングします。以下の **oc adm catalog mirror** コマンドは、カスタム Operator カタログ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、イメージをレジストリーにミラーリングします。

```
$ oc adm catalog mirror \
  <registry_host_name>:<port>/olm/redhat-operators:v2 ❶
  <registry_host_name>:<port> \
  [-a ${REG_CREDS}] ❷
  [--insecure] ❸
  --filter-by-os='*' ❹
```

- ❶ 新規の Operator カタログイメージを指定します。
- ❷ 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。
- ❸ 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。
- ❹ このフラグは、現時点で複数のアーキテクチャーのサポートに関して問題が存在するため必要になります。**--filter-by-os** フラグが設定されていない状態か、または ***** 以外の値に設定されている場合、コマンドが複数の異なるアーキテクチャーをフィルターし、マニフェスト一覧のダイジェスト (**multi-arch image** イメージとしても知られる) が変更されます。ダイジェストが間違っていると、それらのイメージおよび Operator の非接続クラスターでのデプロイメントに失敗します。詳細は、[BZ#1890951](#) を参照してください。

4. 新たに生成されたマニフェストを適用します。

```
$ oc apply -f ./redhat-operators-manifests
```



重要

imageContentSourcePolicy.yaml マニフェストを適用する必要がない場合があります。ファイルの **diff** を完了して、変更が必要かどうかを判断します。

5. カタログイメージを参照する **CatalogSource** オブジェクトを更新します。

a. この **CatalogSource** オブジェクトの元の **catalogsource.yaml** ファイルがある場合:

i. **catalogsource.yaml** ファイルを編集し、**spec.image** フィールドで新規カタログイメージを参照できるようにします。

```
apiVersion: operators.coreos.com/v1alpha1
kind: CatalogSource
metadata:
  name: my-operator-catalog
  namespace: openshift-marketplace
spec:
  sourceType: grpc
  image: <registry_host_name>:<port>/olm/redhat-operators:v2 1
  displayName: My Operator Catalog
  publisher: grpc
```

1 新規の Operator カタログイメージを指定します。

ii. 更新されたファイルを使用して **CatalogSource** オブジェクトを置き換えます。

```
$ oc replace -f catalogsource.yaml
```

b. または、以下のコマンドを使用してカタログソースを編集し、**spec.image** パラメーターで新規カタログイメージを参照します。

```
$ oc edit catalogsource <catalog_source_name> -n openshift-marketplace
```

更新された Operator は、OpenShift Container Platform クラスターの **OperatorHub** ページから利用できるようになりました。

関連情報

- [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](#)

3.8.5. Operator カタログイメージのテスト

Operator カタログイメージのコンテンツは、これをコンテナとして実行し、gRPC API をクエリーして検証できます。イメージをさらにテストするには、**CatalogSource** オブジェクトでイメージを参照して OLM サブスクリプションを解決できます。この例では、以前にビルドされ、サポートされているレジストリーにプッシュされたカスタム **redhat-operators** カタログイメージを使用します。

前提条件

- サポートされているレジストリーにプッシュされるカスタム Operator カタログイメージ
- **podman** version 1.4.4+

- **oc** version 4.3.5+
- [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
- [grpcurl](#)

手順

1. Operator カタログイメージをプルします。

```
$ podman pull <registry_host_name>:<port>/olm/redhat-operators:v1
```

2. イメージを実行します。

```
$ podman run -p 50051:50051 \
  -it <registry_host_name>:<port>/olm/redhat-operators:v1
```

3. **grpcurl** を使用して利用可能なパッケージの実行中のイメージをクエリーします。

```
$ grpcurl -plaintext localhost:50051 api.Registry/ListPackages
```

出力例

```
{
  "name": "3scale-operator"
}
{
  "name": "amq-broker"
}
{
  "name": "amq-online"
}
```

4. チャネルの最新の Operator バンドルを取得します。

```
$ grpcurl -plaintext -d '{"pkgName":"kiali-oss","channelName":"stable"}' localhost:50051
api.Registry/GetBundleForChannel
```

出力例

```
{
  "csvName": "kiali-operator.v1.0.7",
  "packageName": "kiali-oss",
  "channelName": "stable",
  ...
}
```

5. イメージのダイジェストを取得します。

```
$ podman inspect \
  --format='{{index .RepoDigests 0}}' \
  <registry_host_name>:<port>/olm/redhat-operators:v1
```

出力例


```
example_registry:5000/olm/redhat-
operators@sha256:f73d42950021f9240389f99ddc5b0c7f1b533c054ba344654ff1edaf6bf827e3
```

6. Operator グループが Operator とその依存関係をサポートする namespace **my-ns** にあることを前提とし、イメージダイジェストを使用して **CatalogSource** オブジェクトを作成します。以下に例を示します。

```
apiVersion: operators.coreos.com/v1alpha1
kind: CatalogSource
metadata:
  name: custom-redhat-operators
  namespace: my-ns
spec:
  sourceType: grpc
  image: example_registry:5000/olm/redhat-
operators@sha256:f73d42950021f9240389f99ddc5b0c7f1b533c054ba344654ff1edaf6bf827e3

  displayName: Red Hat Operators
```

7. カタログイメージから、利用可能な最新の **servicemeshoperator** およびその依存関係を解決するサブスクリプションを作成します。

```
apiVersion: operators.coreos.com/v1alpha1
kind: Subscription
metadata:
  name: servicemeshoperator
  namespace: my-ns
spec:
  source: custom-redhat-operators
  sourceNamespace: my-ns
  name: servicemeshoperator
  channel: "1.0"
```

第4章 OPERATOR の開発

4.1. OPERATOR SDK の使用を開始する

以下では、Operator SDK の基本事項についての概要を説明し、単純な Go ベースの Memcached Operator のビルドおよびインストールからアップグレードまでのそのライフサイクル管理の例を使って、(OpenShift Container Platform などの) クラスター管理者の Kubernetes ベースのクラスターへのアクセスを持つ Operator の作成者を支援します。

これは、Operator SDK (**operator-sdk** CLI ツールおよび **controller-runtime** ライブラリー API) と Operator Lifecycle Manager (OLM) という 2 つの Operator Framework の重要な設定要素を使用して実行されます。



注記

OpenShift Container Platform 4.5 は Operator SDK v0.17.2 をサポートします。

4.1.1. Operator SDK のアーキテクチャー

Operator Framework は **Operator** という Kubernetes ネイティブアプリケーションを効果的かつ自動化された拡張性のある方法で管理するためのオープンソースツールキットです。Operator は、プロビジョニング、スケーリング、バックアップおよび復元などのクラウドサービスの自動化の利点を提供し、同時に Kubernetes が実行されるいずれの場所でも実行できます。

Operator により、Kubernetes の上部に複雑で、ステートフルなアプリケーションを管理することが容易になります。ただし、現時点で Operator の作成は、低レベルの API の使用、スケルトンコードの作成、モジュール化の欠如による重複の発生などの課題があるために困難になる場合があります。

Operator SDK は、以下を提供して Operator をより容易に作成できるように設計されたフレームワークです。

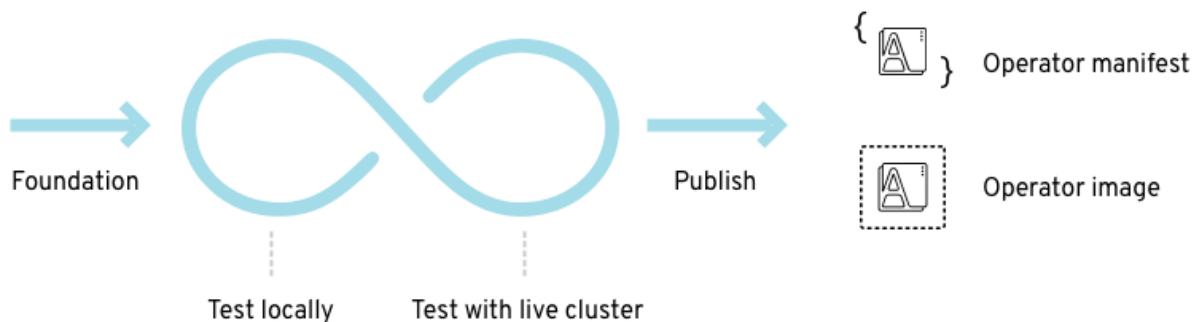
- 運用ロジックをより直感的に作成するための高レベルの API および抽象化
- 新規プロジェクトを迅速にブートストラップするためのスケルトンコードの作成およびコード生成ツール
- 共通する Operator ユースケースに対応する拡張機能

4.1.1.1. ワークフロー

Operator SDK は、新規 Operator を開発するために以下のワークフローを提供します。

1. Operator SDK コマンドラインインターフェイス (CLI) を使用した新規 Operator プロジェクトの作成。
2. カスタムリソース定義 (CRD) を追加することによる新規リソース API の定義。
3. Operator SDK API を使用した監視対象リソースの指定。
4. 指定されたハンドラーでの Operator 調整 (reconciliation) ロジックの定義、およびリソースと対話するための Operator SDK API の使用。
5. Operator Deployment マニフェストをビルドし、生成するための Operator SDK CLI の使用。

図4.1 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。

4.1.1.2. マネージャーファイル

Operator の主なプログラムは、**cmd/manager/main.go** のマネージャーファイルです。マネージャーは、**pkg/apis/** で定義されるすべてのカスタムリソース (CR) のスキームを自動的に登録し、**pkg/controller/** 下のすべてのコントローラーを実行します。

マネージャーは、すべてのコントローラーがリソースの監視に使用する namespace を制限できます。

```
mgr, err := manager.New(cfg, manager.Options{Namespace: namespace})
```

デフォルトでは、これは Operator が実行されている namespace です。すべての namespace を確認するには、namespace オプションのオプションを空のままにすることができます。

```
mgr, err := manager.New(cfg, manager.Options{Namespace: ""})
```

4.1.1.3. 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。

4.1.2. Operator SDK CLI のインストール

Operator SDK には、開発者による新規 Operator プロジェクトの作成、ビルドおよびデプロイを支援をする CLI ツールが含まれます。ワークステーションに SDK CLI をインストールして、独自の Operator のオーサリングを開始することができます。



注記

以下では、ローカル Kubernetes クラスターとしての [minikube](#) v0.25.0+ とパブリックレジストリーの [quay.io](#) を使用します。

4.1.2.1. GitHub リリースからのインストール

GitHub のプロジェクトから Operator SDK CLI の事前ビルドリリースのバイナリーをダウンロードし、インストールできます。

前提条件

- [Go](#) v1.13+
- **docker** v17.03+、**podman** v1.2.0+、または **buildah** v1.7+
- OpenShift CLI (**oc**) v4.5+ (インストール済み)
- Kubernetes v1.12.0+ に基づくクラスターへのアクセス
- コンテナレジストリーへのアクセス

手順

1. リリースバージョン変数を設定します。

```
$ RELEASE_VERSION=v0.17.2
```

2. リリースバイナリーをダウンロードします。

- Linux の場合

```
$ curl -OJL https://github.com/operator-framework/operator-  
sdk/releases/download/${RELEASE_VERSION}/operator-sdk-${RELEASE_VERSION}-  
x86_64-linux-gnu
```

- macOS の場合

```
$ curl -OJL https://github.com/operator-framework/operator-  
sdk/releases/download/${RELEASE_VERSION}/operator-sdk-${RELEASE_VERSION}-  
x86_64-apple-darwin
```

3. ダウンロードしたリリースのバイナリーを確認します。

- a. 提供された **.asc** ファイルをダウンロードします。

- Linux の場合

```
$ curl -OJL https://github.com/operator-framework/operator-  
sdk/releases/download/${RELEASE_VERSION}/operator-sdk-  
${RELEASE_VERSION}-x86_64-linux-gnu.asc
```

- macOS の場合

```
$ curl -OJL https://github.com/operator-framework/operator-
sdk/releases/download/${RELEASE_VERSION}/operator-sdk-
${RELEASE_VERSION}-x86_64-apple-darwin.asc
```

- b. バイナリーと対応する **.asc** ファイルを同じディレクトリーに置き、以下のコマンドを実行してバイナリーを確認します。

- Linux の場合

```
$ gpg --verify operator-sdk-${RELEASE_VERSION}-x86_64-linux-gnu.asc
```

- macOS の場合

```
$ gpg --verify operator-sdk-${RELEASE_VERSION}-x86_64-apple-darwin.asc
```

保守管理者のパブリックキーがワークステーションにない場合は、以下のエラーが出されます。

エラーのある出力例

```
$ gpg: assuming signed data in 'operator-sdk-${RELEASE_VERSION}-x86_64-apple-
darwin'
$ gpg: Signature made Fri Apr 5 20:03:22 2019 CEST
$ gpg:         using RSA key <key_id> ❶
$ gpg: Can't check signature: No public key
```

- ❶ RSA キー文字列。

キーをダウンロードするには、以下のコマンドを実行し、**<key_id>** を直前のコマンドの出力で提供された RSA キー文字列に置き換えます。

```
$ gpg [--keyserver keys.gnupg.net] --recv-key "<key_id>" ❶
```

- ❶ キーサーバーが設定されていない場合、これを **--keyserver** オプションで指定します。

4. リリースバイナリーを **PATH** にインストールします。

- Linux の場合

```
$ chmod +x operator-sdk-${RELEASE_VERSION}-x86_64-linux-gnu
```

```
$ sudo cp operator-sdk-${RELEASE_VERSION}-x86_64-linux-gnu
/usr/local/bin/operator-sdk
```

```
$ rm operator-sdk-${RELEASE_VERSION}-x86_64-linux-gnu
```

- macOS の場合

```
$ chmod +x operator-sdk-${RELEASE_VERSION}-x86_64-apple-darwin
```

-

```
$ sudo cp operator-sdk-${RELEASE_VERSION}-x86_64-apple-darwin  
/usr/local/bin/operator-sdk
```

```
$ rm operator-sdk-${RELEASE_VERSION}-x86_64-apple-darwin
```

5. CLI ツールが正しくインストールされていることを確認します。

```
$ operator-sdk version
```

4.1.2.2. Homebrew からのインストール

Homebrew を使用して SDK CLI をインストールできます。

前提条件

- [Homebrew](#)
- **docker** v17.03+、**podman** v1.2.0+、または **buildah** v1.7+
- OpenShift CLI (**oc**) v4.5+ (インストール済み)
- Kubernetes v1.12.0+ に基づくクラスターへのアクセス
- コンテナレジストリーへのアクセス

手順

1. **brew** コマンドを使用して SDK CLI をインストールします。

```
$ brew install operator-sdk
```

2. CLI ツールが正しくインストールされていることを確認します。

```
$ operator-sdk version
```

4.1.2.3. ソースを使用したコンパイルおよびインストール

Operator SDK ソースコードを取得して、SDK CLI をコンパイルし、インストールできます。

前提条件

- [Git](#)
- [Go](#) v1.13+
- **docker** v17.03+、**podman** v1.2.0+、または **buildah** v1.7+
- OpenShift CLI (**oc**) v4.5+ (インストール済み)
- Kubernetes v1.12.0+ に基づくクラスターへのアクセス
- コンテナレジストリーへのアクセス

手順

1. **operator-sdk** リポジトリのクローンを作成します。

```
$ mkdir -p $GOPATH/src/github.com/operator-framework
```

```
$ cd $GOPATH/src/github.com/operator-framework
```

```
$ git clone https://github.com/operator-framework/operator-sdk
```

```
$ cd operator-sdk
```

2. 必要なリリースブランチをチェックアウトします。

```
$ git checkout master
```

3. SDK CLI ツールをコンパイルし、インストールします。

```
$ make dep
```

```
$ make install
```

これにより、**\$GOPATH/bin** に CLI バイナリー **operator-sdk** がインストールされます。

4. CLI ツールが正しくインストールされていることを確認します。

```
$ operator-sdk version
```

4.1.3. Operator SDK を使用した Go ベースの Operator のビルド

Operator SDK は、詳細なアプリケーション固有の運用上の知識を必要とする可能性のあるプロセスである、Kubernetes ネイティブアプリケーションのビルドを容易にします。SDK はこの障壁を低くするだけでなく、メータリングやモニタリングなどの数多くの一般的な管理機能に必要なスケルトンコードの量を減らします。

この手順では、SDK によって提供されるツールおよびライブラリーを使用して単純な Memcached Operator をビルドする例を示します。

前提条件

- 開発ワークステーションにインストールされる Operator SDK CLI
- OpenShift Container Platform 4.5 などの、Kubernetes ベースのクラスター (v1.8 以上の **apps/v1beta2** API グループをサポートするもの) にインストールされる Operator Lifecycle Manager (OLM)
- **cluster-admin** パーミッションのあるアカウントを使用したクラスターへのアクセス
- OpenShift CLI (**oc**) v4.5+ (インストール済み)

手順

1. 新規プロジェクトを作成します。

CLI を使用して新規 **memcached-operator** プロジェクトを作成します。

```
$ mkdir -p $GOPATH/src/github.com/example-inc/
$ cd $GOPATH/src/github.com/example-inc/
$ operator-sdk new memcached-operator
$ cd memcached-operator
```

2. 新規カスタムリソース定義 (CRD) を追加します。

- a. **APIVersion** を **cache.example.com/v1alpha1** に設定し、**Kind** を **Memcached** に設定した状態で、CLI を使用して **Memcached** という新規 CRD API を追加します。

```
$ operator-sdk add api \
  --api-version=cache.example.com/v1alpha1 \
  --kind=Memcached
```

これにより、**pkg/apis/cache/v1alpha1/** の下で Memcached resource API のスキュフォールディングが実行されます。

- b. **pkg/apis/cache/v1alpha1/memcached_types.go** ファイルで、**Memcached** カスタムリソース (CR) の仕様およびステータスを変更します。

```
type MemcachedSpec struct {
    // Size is the size of the memcached deployment
    Size int32 `json:"size"`
}
type MemcachedStatus struct {
    // Nodes are the names of the memcached pods
    Nodes []string `json:"nodes"`
}
```

- c. ***_types.go** ファイルを変更後は、以下のコマンドを常に行き、該当するリソースタイプ用に生成されたコードを更新します。

```
$ operator-sdk generate k8s
```

3. オプション: カスタム検証を CRD に追加します。

OpenAPI v3.0 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。

さらに、**pkg/apis/<group>/<version>/zz_generated.openapi.go** ファイルが生成されます。このファイルには、デフォルトで存在する **+k8s:openapi-gen=true** annotation が **Kind** 型の宣言の上に存在する場合に、この検証ブロックの Go 表現が含まれます。この自動生成コードは Go の **Kind** タイプの OpenAPI モデルです。これを使用して完全な OpenAPI 仕様を作成し、クライアントを生成できます。

Operator の作成者は Kubebuilder マーカー (アノテーション) を使用して API のカスタム検証を設定できます。これらのマーカーには、**+kubebuilder:validation** 接頭辞が常に必要です。たとえば、以下のマーカーを追加して enum 型の仕様を追加できます。


```
// +kubebuilder:validation:Enum=Lion;Wolf;Dragon
type Alias string
```

API コードのマーカーの使用については、Kubebuilder ドキュメントの [Generating CRDs](#) および [Markers for Config/Code Generation](#) を参照してください。OpenAPIv3 検証マーカーの詳細の一覧については、Kubebuilder ドキュメントの [CRD Validation](#) を参照してください。

カスタム検証を追加する場合は、以下のコマンドを実行し、CRD の **deploy/crds/cache.example.com_memcacheds_crd.yaml** ファイルの OpenAPI 検証セクションを更新します。

```
$ operator-sdk generate crds
```

生成される YAML の例

```
spec:
  validation:
    openAPIV3Schema:
      properties:
        spec:
          properties:
            size:
              format: int32
              type: integer
```

4. 新規コントローラーを追加します。

- a. 新規コントローラーをプロジェクトに追加し、**Memcached** リソースを確認し、調整します。

```
$ operator-sdk add controller \
  --api-version=cache.example.com/v1alpha1 \
  --kind=Memcached
```

これにより、**pkg/controller/memcached/** の下で新規コントローラー実装のスキファオルディングが実行されます。

- b. この例では、生成されたコントローラーファイル **pkg/controller/memcached/memcached_controller.go** を [実装例](#) に置き換えます。コントローラーのサンプルは、それぞれの **Memcached** リソース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)。
- Deployment のサイズが、**Memcached** CR 仕様で指定されるのと同じであることを確認します。
- **Memcached** リソースのステータスを Memcached Pod の名前で更新します。

次の2つのサブステップでは、コントローラーがリソースを監視する方法および調整ループがトリガーされる方法を確認します。これらの手順を省略し、直接 Operator のビルドおよび実行に進むことができます。

- c. **pkg/controller/memcached/memcached_controller.go** ファイルでコントローラーの実装を確認し、コントローラーのリソースの監視方法を確認します。

最初の監視は、プライマリソースとしての **Memcached** タイプに対して実行します。それぞれの Add、Update、または Delete イベントについて、reconcile ループに **Memcached** オブジェクトの reconcile **Request** (`<namespace>:<name>` キー) が送られます。

```
err := c.Watch(
    &source.Kind{Type: &cachev1alpha1.Memcached{}},
    &handler.EnqueueRequestForObject{}
```

次の監視は、**Deployment** オブジェクトに対して実行されますが、イベントハンドラーは各イベントを、デプロイメントのオーナーの reconcile (調整) **Request** にマップします。この場合、これはデプロイメントが作成された **Memcached** オブジェクトです。これにより、コントローラーはデプロイメントをセカンダリリソースとして監視できます。

```
err := c.Watch(&source.Kind{Type: &appsv1.Deployment{}},
    &handler.EnqueueRequestForOwner{
        IsController: true,
        OwnerType:    &cachev1alpha1.Memcached{},
    })
```

- d. すべてのコントローラーには、reconcile ループを実装する **Reconcile()** メソッドのある **Reconciler** オブジェクトがあります。この reconcile ループには、キャッシュからプライマリリソースオブジェクトの **Memcached** を検索するために使用される `<namespace>:<name>` キーである **Request** 引数が渡されます。

```
func (r *ReconcileMemcached) Reconcile(request reconcile.Request) (reconcile.Result,
error) {
    // Lookup the Memcached instance for this reconcile request
    memcached := &cachev1alpha1.Memcached{}
    err := r.client.Get(context.TODO(), request.NamespacedName, memcached)
    ...
}
```

Reconcile() 関数の返り値に応じて、reconcile **Request** は再度キューに入れられ、ループが再びトリガーされる可能性があります。

```
// Reconcile successful - don't requeue
return reconcile.Result{}, nil
// Reconcile failed due to error - requeue
return reconcile.Result{}, err
// Requeue for any reason other than error
return reconcile.Result{Requeue: true}, nil
```

5. Operator をビルドし、実行します。

- a. Operator の実行前に、CRD を Kubernetes API サーバーに再度登録する必要があります。

```
$ oc create \
    -f deploy/crds/cache_v1alpha1_memcached_crd.yaml
```

- b. CRD の登録後に、Operator を実行するための 2 つのオプションを選択できます。

- Kubernetes クラスター内の Deployment を使用

- クラスター内の Go プログラムを使用

以下の方法のいずれかを選択します。

i. オプション A: クラスター内のデプロイメントとして実行する。

- A. **memcached-operator** イメージをビルドし、これをレジストリーにプッシュします。

```
$ operator-sdk build quay.io/example/memcached-operator:v0.0.1
```

- B. デプロイメントマニフェストは **deploy/operator.yaml** に生成されます。デフォルトはプレースホルダーでしかないので、以下のようにデプロイメントイメージを更新します。

```
$ sed -i 's|REPLACE_IMAGE|quay.io/example/memcached-operator:v0.0.1|g'
  deploy/operator.yaml
```

- C. 次のステップについての quay.io にアカウントがあることを確認するか、または優先しているコンテナレジストリーで置き換えます。レジストリーには、[memcached-operator](#) という名前の **新規パブリックイメージ** リポジトリを作成します。

- D. イメージをレジストリーにプッシュします。

```
$ podman push quay.io/example/memcached-operator:v0.0.1
```

- E. RBAC をセットアップし、**memcached-operator** マニフェストを作成します。

```
$ oc create -f deploy/role.yaml
```

```
$ oc create -f deploy/role_binding.yaml
```

```
$ oc create -f deploy/service_account.yaml
```

```
$ oc create -f deploy/operator.yaml
```

- F. **memcached-operator** デプロイが稼働していることを確認します。

```
$ oc get deployment
```

出力例

```
NAME                DESIRED  CURRENT  UP-TO-DATE  AVAILABLE  AGE
memcached-operator   1        1        1            1          1m
```

ii. オプション B: クラスター外でローカルに実行する。

この方法は、迅速にデプロイメントおよびテストを実行するための開発サイクルで優先される方法です。

\$HOME/.kube/config にあるデフォルトの Kubernetes 設定ファイルを使用して Operator をローカルで実行します。

```
$ operator-sdk run --local --namespace=default
```

フラグ **--kubeconfig=<path/to/kubeconfig>** を使用して特定の **kubeconfig** を使用できます。

6. **Memcached** CR を作成して、Operator が **Memcached** アプリケーションをデプロイできることを確認します。

- a. **deploy/crds/cache_v1alpha1_memcached_cr.yaml** で生成された **Memcached** CR のサンプルを作成します。

- b. ファイルを表示します。

```
$ cat deploy/crds/cache_v1alpha1_memcached_cr.yaml
```

出力例

```
apiVersion: "cache.example.com/v1alpha1"
kind: "Memcached"
metadata:
  name: "example-memcached"
spec:
  size: 3
```

- c. オブジェクトを作成します。

```
$ oc apply -f deploy/crds/cache_v1alpha1_memcached_cr.yaml
```

- d. **memcached-operator** が CR のデプロイメントを作成できることを確認します。

```
$ oc get deployment
```

出力例

| NAME | DESIRED | CURRENT | UP-TO-DATE | AVAILABLE | AGE |
|--------------------|---------|---------|------------|-----------|-----|
| memcached-operator | 1 | 1 | 1 | 2m | |
| example-memcached | 3 | 3 | 3 | 1m | |

- e. CR ステータスが 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 を確認します。

```
$ oc get pods
```

出力例

| NAME | READY | STATUS | RESTARTS | AGE |
|------------------------------------|-------|---------|----------|-----|
| example-memcached-6fd7c98d8-7dqdr | 1/1 | Running | 0 | 1m |
| example-memcached-6fd7c98d8-g5k7v | 1/1 | Running | 0 | 1m |
| example-memcached-6fd7c98d8-m7vn7 | 1/1 | Running | 0 | 1m |
| memcached-operator-7cc7cfd86-vvjgk | 1/1 | Running | 0 | 2m |

```
$ oc get memcached/example-memcached -o yaml
```

出力例

```

apiVersion: cache.example.com/v1alpha1
kind: Memcached
metadata:
  clusterName: ""
  creationTimestamp: 2018-03-31T22:51:08Z
  generation: 0
  name: example-memcached
  namespace: default
  resourceVersion: "245453"
  selfLink:
/apis/cache.example.com/v1alpha1/namespaces/default/memcacheds/example-
memcached
  uid: 0026cc97-3536-11e8-bd83-0800274106a1
spec:
  size: 3
status:
  nodes:
  - example-memcached-6fd7c98d8-7dqdr
  - example-memcached-6fd7c98d8-g5k7v
  - example-memcached-6fd7c98d8-m7vn7

```

7. デプロイメントのサイズを更新し、Operator がデプロイ済みの Memcached アプリケーションを管理できることを確認します。

- a. **memcached** CR の **spec.size** フィールドを **3** から **4** に変更します。

```
$ cat deploy/crds/cache_v1alpha1_memcached_cr.yaml
```

出力例

```

apiVersion: "cache.example.com/v1alpha1"
kind: "Memcached"
metadata:
  name: "example-memcached"
spec:
  size: 4

```

- b. 変更を適用します。

```
$ oc apply -f deploy/crds/cache_v1alpha1_memcached_cr.yaml
```

- c. Operator が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 deployment
```

出力例

```

NAME           DESIRED  CURRENT  UP-TO-DATE  AVAILABLE  AGE
example-memcached  4        4        4           4          5m

```

8. リソースをクリーンアップします。

```
$ oc delete -f deploy/crds/cache_v1alpha1_memcached_cr.yaml
```

```
$ oc delete -f deploy/crds/cache_v1alpha1_memcached_crd.yaml
```

```
$ oc delete -f deploy/operator.yaml
```

```
$ oc delete -f deploy/role.yaml
```

```
$ oc delete -f deploy/role_binding.yaml
```

```
$ oc delete -f deploy/service_account.yaml
```

追加リソース

- CRD の OpenAPI v3.0 検証スキーマについての詳細は、[Kubernetes ドキュメント](#) を参照してください。

4.1.4. Operator Lifecycle Manager を使用した Go ベースの Operator の管理

直前のセクションでは、Operator を手動で実行することについて説明しました。次のセクションでは、実稼働環境で実行される Operator のより堅牢なデプロイメントモデルを可能にする Operator Lifecycle Manager (OLM) の使用方法について説明します。

OLM は、Kubernetes クラスターで Operator およびそれらの関連サービスをインストールし、更新し、通常はそれらすべての Operator のライフサイクルを管理するのに役立ちます。これは、Kubernetes 拡張として実行され、追加のツールなしにすべてのライフサイクル管理機能について **oc** を使用できます。

前提条件

- OLM が (**apps/v1beta2** API グループをサポートする v1.8 以上のバージョンの) Kubernetes ベースのクラスターにインストールされていること (例: OpenShift Container Platform 4.5)。
- Memcached Operator がビルドされていること。

手順

1. Operator マニフェストを生成します。

Operator マニフェストは、アプリケーションを表示し、作成し、管理する方法について説明します (この場合は Memcached)。これは **ClusterServiceVersion** (CSV) オブジェクトで定義され、OLM が機能するために必要です。

Memcached Operator のビルド時に作成された **memcached-operator/** ディレクトリーから CSV を生成します。

```
$ operator-sdk generate csv --csv-version 0.0.1
```



注記

マニフェストファイルの手動による定義についての詳細は、[Building a CSV for the Operator Framework](#) を参照してください。

2. Operator がターゲットとする namespace を指定する **Operator グループを作成します**。以下の Operator グループを、CSV を作成する namespace に作成します。この例では、**default** namespace が使用されます。

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

3. **Operator をデプロイします**。これらのファイルは、Memcached Operator のビルド時に Operator SDK によって **deploy/** ディレクトリーに生成されたファイルを使用します。
 - a. 各カスタムリソース定義 (CRD) **kind** の **displayName** フィールドを **spec.customresourcedefinitions.owned** セクションに追加して、生成される CSV マニフェストファイルを編集します。

deploy/olm-catalog/memcached-operator/0.0.1/memcached-operator.v0.0.1.clusterserviceversion.yaml ファイル

```
...
spec:
  customresourcedefinitions:
    owned:
      - kind: Memcached
        name: memcacheds.cache.example.com
        version: v1alpha1
        description: Memcached is the Schema for the memcacheds API
        displayName: Memcached ❶
...

```

- ❶ CRD の表示名を指定します。

- b. CSV マニフェストをクラスターの指定された namespace に適用します。

```
$ oc apply -f deploy/olm-catalog/memcached-operator/0.0.1/memcached-operator.v0.0.1.clusterserviceversion.yaml
```

このマニフェストを適用する際に、クラスターはマニフェストで指定された要件を満たしていないためにすぐに更新を実行しません。

- c. リソースパーミッションを Operator に付与するためにロール、ロールバインディング、およびサービスアカウントを作成し、Operator が管理する **Memcached** カスタムリソースを作成するためにカスタムリソース定義 (CRD) を作成します。

```
$ oc create -f deploy/crds/cache.example.com_memcacheds_crd.yaml
```

```
$ oc create -f deploy/service_account.yaml
```

```
$ oc create -f deploy/role.yaml
```

```
$ oc create -f deploy/role_binding.yaml
```

マニフェストの適用時に OLM は Operator を特定の namespace に作成するため、管理者は、Operator をインストールできるユーザーを制限するためのネイティブの Kubernetes RBAC パーミッションモデルを利用できます。

4. アプリケーションインスタンスを作成します。

Memcached Operator が **default** namespace で実行されるようになります。ユーザーはカスタムリソースのインスタンス経由で Operator と対話します。この場合、リソースには **Memcached** の種類が設定されます。ネイティブの Kubernetes RBAC はカスタムリソースに適用され、管理者は各 Operator と対話できるユーザーへの制御が可能になります。

この namespace で **Memcached** オブジェクトのインスタンスを作成することにより、Operator で管理される **memcached** サーバーを実行する Pod をインスタンス化するために Memcached Operator がトリガーされます。カスタムリソースをより多く作成すると、Memcached アプリケーションのより多くの固有なインスタンスがこの namespace で実行されている Memcached Operator によって管理されます。

```
$ cat <<EOF | oc apply -f -
apiVersion: "cache.example.com/v1alpha1"
kind: "Memcached"
metadata:
  name: "memcached-for-wordpress"
spec:
  size: 1
EOF
```

```
$ cat <<EOF | oc apply -f -
apiVersion: "cache.example.com/v1alpha1"
kind: "Memcached"
metadata:
  name: "memcached-for-drupal"
spec:
  size: 1
EOF
```

```
$ oc get Memcached
```

出力例

```
NAME                AGE
memcached-for-drupal 22s
memcached-for-wordpress 27s
```

```
$ oc get pods
```

出力例

```
NAME                                READY   STATUS    RESTARTS   AGE
memcached-app-operator-66b5777b79-pnsfj  1/1     Running   0          14m
memcached-for-drupal-5476487c46-qbd66    1/1     Running   0          3s
memcached-for-wordpress-65b75fd8c9-7b9x7  1/1     Running   0          8s
```


4.1.5. 追加リソース

- Operator SDK によって作成されるプロジェクトディレクトリ構造についての詳細は、[Appendices](#) を参照してください。
- [Operator Development Guide for Red Hat Partners](#)

4.2. ANSIBLE ベース OPERATOR の作成

以下では、Operator SDK における Ansible サポートについての概要を説明し、Operator の作成者に、Ansible Playbook およびモジュールを使用する **operator-sdk** CLI ツールを使って Ansible ベースの Operator をビルドし、実行するサンプルを示します。

4.2.1. Operator SDK における Ansible サポート

[Operator Framework](#) は **Operator** という Kubernetes ネイティブアプリケーションを効果的かつ自動化された拡張性のある方法で管理するためのオープンソースツールキットです。このフレームワークには Operator SDK が含まれ、これは Kubernetes API の複雑性を把握していなくても、それぞれの専門知識に基づいて Operator のブートストラップおよびビルドを実行できるように開発者を支援します。

Operator プロジェクトを生成するための Operator SDK のオプションの1つに、Go コードを作成することなしに Kubernetes リソースを統一されたアプリケーションとしてデプロイするために既存の Ansible Playbook およびモジュールを使用できるオプションがあります。

4.2.1.1. カスタムリソースファイル

Operator は Kubernetes の拡張メカニズムであるカスタムリソース定義 (CRD) を使用するため、カスタムリソース (CR) は、組み込み済みのネイティブ Kubernetes オブジェクトのように表示され、機能します。

CR ファイル形式は Kubernetes リソースファイルです。オブジェクトには、必須およびオプションフィールドが含まれます。

表4.1 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4.2 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"
```

4.2.1.2. 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4.3 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。

4.2.1.2.1. 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4.4 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
```

4.2.1.3. Ansible に送信される追加変数

追加の変数を Ansible に送信し、Operator で管理できます。カスタマーリソース (CR) の **spec** セクションでは追加変数としてキーと値のペアを渡します。これは、**ansible-playbook** コマンドに渡される追加変数と同等です。

また Operator は、CR の名前および CR の namespace についての **meta** フィールドの下に追加の変数を渡します。

以下は CR の例になります。

```
apiVersion: "app.example.com/v1alpha1"
kind: "Database"
```

```

metadata:
  name: "example"
spec:
  message:"Hello world 2"
  newParameter: "newParam"

```

追加変数として Ansible に渡される構造は以下のとおりです。

```

{ "meta": {
  "name": "<cr_name>",
  "namespace": "<cr_namespace>",
},
"message": "Hello world 2",
"new_parameter": "newParam",
"_app_example_com_database": {
  <full_crd>
},
}

```

message および **newParameter** フィールドは追加変数として上部に設定され、**meta** は Operator に定義されるように CR の関連メタデータを提供します。**meta** フィールドは、Ansible のドット表記などを使用してアクセスできます。

```

- debug:
  msg: "name: {{ meta.name }}, {{ meta.namespace }}"

```

4.2.1.4. 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。

4.2.2. Operator SDK CLI のインストール

Operator SDK には、開発者による新規 Operator プロジェクトの作成、ビルドおよびデプロイを支援をする CLI ツールが含まれます。ワークステーションに SDK CLI をインストールして、独自の Operator のオーサリングを開始することができます。



注記

以下では、ローカル Kubernetes クラスターとしての [minikube](#) v0.25.0+ とパブリックレジストリーの [quay.io](#) を使用します。

4.2.2.1. GitHub リリースからのインストール

GitHub のプロジェクトから Operator SDK CLI の事前ビルドリリースのバイナリーをダウンロードし、インストールできます。

前提条件

- [Go](#) v1.13+
- **docker** v17.03+、**podman** v1.2.0+、または **buildah** v1.7+
- OpenShift CLI (**oc**) v4.5+ (インストール済み)
- Kubernetes v1.12.0+ に基づくクラスターへのアクセス
- コンテナレジストリーへのアクセス

手順

1. リリースバージョン変数を設定します。

```
$ RELEASE_VERSION=v0.17.2
```

2. リリースバイナリーをダウンロードします。

- Linux の場合

```
$ curl -OJL https://github.com/operator-framework/operator-  
sdk/releases/download/${RELEASE_VERSION}/operator-sdk-${RELEASE_VERSION}-  
x86_64-linux-gnu
```

- macOS の場合

```
$ curl -OJL https://github.com/operator-framework/operator-  
sdk/releases/download/${RELEASE_VERSION}/operator-sdk-${RELEASE_VERSION}-  
x86_64-apple-darwin
```

3. ダウンロードしたリリースのバイナリーを確認します。

- a. 提供された **.asc** ファイルをダウンロードします。

- Linux の場合

```
$ curl -OJL https://github.com/operator-framework/operator-  
sdk/releases/download/${RELEASE_VERSION}/operator-sdk-  
${RELEASE_VERSION}-x86_64-linux-gnu.asc
```

- macOS の場合

```
$ curl -OJL https://github.com/operator-framework/operator-  
sdk/releases/download/${RELEASE_VERSION}/operator-sdk-  
${RELEASE_VERSION}-x86_64-apple-darwin.asc
```

- b. バイナリーと対応する **.asc** ファイルを同じディレクトリーに置き、以下のコマンドを実行してバイナリーを確認します。

- Linux の場合

```
$ gpg --verify operator-sdk-${RELEASE_VERSION}-x86_64-linux-gnu.asc
```

- macOS の場合

■

```
$ gpg --verify operator-sdk-${RELEASE_VERSION}-x86_64-apple-darwin.asc
```

保守管理者のパブリックキーがワークステーションにない場合は、以下のエラーが出されます。

エラーのある出力例

```
$ gpg: assuming signed data in 'operator-sdk-${RELEASE_VERSION}-x86_64-apple-darwin'
$ gpg: Signature made Fri Apr 5 20:03:22 2019 CEST
$ gpg:         using RSA key <key_id> ❶
$ gpg: Can't check signature: No public key
```

❶ RSA キー文字列。

キーをダウンロードするには、以下のコマンドを実行し、**<key_id>** を直前のコマンドの出力で提供された RSA キー文字列に置き換えます。

```
$ gpg [--keyserver keys.gnupg.net] --recv-key "<key_id>" ❶
```

❶ キーサーバーが設定されていない場合、これを **--keyserver** オプションで指定します。

4. リリースバイナリーを **PATH** にインストールします。

- Linux の場合

```
$ chmod +x operator-sdk-${RELEASE_VERSION}-x86_64-linux-gnu
```

```
$ sudo cp operator-sdk-${RELEASE_VERSION}-x86_64-linux-gnu
/usr/local/bin/operator-sdk
```

```
$ rm operator-sdk-${RELEASE_VERSION}-x86_64-linux-gnu
```

- macOS の場合

```
$ chmod +x operator-sdk-${RELEASE_VERSION}-x86_64-apple-darwin
```

```
$ sudo cp operator-sdk-${RELEASE_VERSION}-x86_64-apple-darwin
/usr/local/bin/operator-sdk
```

```
$ rm operator-sdk-${RELEASE_VERSION}-x86_64-apple-darwin
```

5. CLI ツールが正しくインストールされていることを確認します。

```
$ operator-sdk version
```

4.2.2.2. Homebrew からのインストール

Homebrew を使用して SDK CLI をインストールできます。

前提条件

- [Homebrew](#)
- **docker** v17.03+、**podman** v1.2.0+、または **buildah** v1.7+
- OpenShift CLI (**oc**) v4.5+ (インストール済み)
- Kubernetes v1.12.0+ に基づくクラスターへのアクセス
- コンテナレジストリーへのアクセス

手順

1. **brew** コマンドを使用して SDK CLI をインストールします。

```
$ brew install operator-sdk
```

2. CLI ツールが正しくインストールされていることを確認します。

```
$ operator-sdk version
```

4.2.2.3. ソースを使用したコンパイルおよびインストール

Operator SDK ソースコードを取得して、SDK CLI をコンパイルし、インストールできます。

前提条件

- [Git](#)
- [Go](#) v1.13+
- **docker** v17.03+、**podman** v1.2.0+、または **buildah** v1.7+
- OpenShift CLI (**oc**) v4.5+ (インストール済み)
- Kubernetes v1.12.0+ に基づくクラスターへのアクセス
- コンテナレジストリーへのアクセス

手順

1. **operator-sdk** リポジトリのクローンを作成します。

```
$ mkdir -p $GOPATH/src/github.com/operator-framework
```

```
$ cd $GOPATH/src/github.com/operator-framework
```

```
$ git clone https://github.com/operator-framework/operator-sdk
```

```
$ cd operator-sdk
```


2. 必要なりリリースブランチをチェックアウトします。

```
$ git checkout master
```

3. SDK CLI ツールをコンパイルし、インストールします。

```
$ make dep
```

```
$ make install
```

これにより、`$GOPATH/bin` に CLI バイナリー **operator-sdk** がインストールされます。

4. CLI ツールが正しくインストールされていることを確認します。

```
$ operator-sdk version
```

4.2.3. Operator SDK を使用した Ansible ベースの Operator のビルド

以下の手順では、Operator SDK が提供するツールおよびライブラリーを使用した Ansible Playbook がサポートする単純な Memcached Operator のビルドの例について説明します。

前提条件

- 開発ワークステーションにインストールされる Operator SDK CLI
- **cluster-admin** パーミッションを持つアカウントを使用した Kubernetes ベースのクラスター `v1.11.3+` (OpenShift Container Platform 4.5 など) へのアクセス
- OpenShift CLI (**oc**) `v4.5+` (インストール済み)
- **ansible** `v2.9.0+`
- **ansible-runner** `v1.1.0+`
- **ansible-runner-http** `v1.0.0+`

手順

1. **新規 Operator プロジェクトを作成します。** namespace スコープの Operator は単一 namespace でリソースを監視し、管理します。namespace スコープの Operator は柔軟性があるために優先して使用されます。これらの Operator は切り離されたアップグレード、障害対応およびモニタリングのための namespace の分離、および API 定義の差異化を可能にします。

新規の Ansible ベース、namespace スコープの **memcached-operator** プロジェクトを作成し、新規ディレクトリーに切り換えるには、以下のコマンドを使用します。

```
$ operator-sdk new memcached-operator \
  --api-version=cache.example.com/v1alpha1 \
  --kind=Memcached \
  --type=ansible
```

```
$ cd memcached-operator
```

これにより、とくに API バージョン **example.com/v1alpha1** および Kind **Memcached** の **Memcached** リソースを監視するための **memcached-operator** プロジェクトが作成されます。

2. Operator ロジックをカスタマイズします。

この例では、**memcached-operator** はそれぞれの **Memcached** カスタムリソース (CR) について以下の調整 (reconciliation) ロジックを実行します。

- **memcached** 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 で指定されるのと同じであることを確認します。

デフォルトで、**memcached-operator** は **watches.yaml** ファイルに示されるように **Memcached** リソースイベントを監視し、Ansible ロール **Memcached** を実行します。

```
- version: v1alpha1
  group: cache.example.com
  kind: Memcached
```

オプションで、以下のロジックを **watches.yaml** ファイルでカスタマイズできます。

- a. **role** オプションを指定して、**ansible-runner** を Ansible ロールを使って起動する際に Operator がこの特定のパスを使用するように設定します。デフォルトで、**operator-sdk new** コマンドでは、ロールが置かれる場所への絶対パスを入力します。

```
- version: v1alpha1
  group: cache.example.com
  kind: Memcached
  role: /opt/ansible/roles/memcached
```

- b. **playbook** オプションを **watches.yaml** ファイルに指定して、**ansible-runner** を Ansible Playbook で起動する際に Operator がこの指定されたパスを使用するように設定します。

```
- version: v1alpha1
  group: cache.example.com
  kind: Memcached
  playbook: /opt/ansible/playbook.yaml
```

3. Memcached Ansible ロールをビルドします。

生成された Ansible ロールを **roles/memcached/** ディレクトリの下で変更します。この Ansible ロールは、リソースの変更時に実行されるロジックを制御します。

- a. **Memcached 仕様を定義します。**

Ansible ベースの Operator の定義は Ansible 内ですべて実行できます。Ansible Operator は CR 仕様フィールドのすべてのキー/値ペアを **変数** として Ansible に渡します。仕様フィールドのすべての変数の名前は、Ansible の実行前に Operator によってスネークケース (小文字 + アンダースコア) に変換されます。たとえば、仕様の **serviceAccount** は Ansible では **service_account** になります。

ヒント

Ansible で変数についてのタイプの検証を実行し、アプリケーションが予想される入力を受信できることを確認する必要があります。

ユーザーが **spec** フィールドを設定しない場合、**roles/memcached/defaults/main.yml** ファイルを変更してデフォルトを設定します。

```
size: 1
```

b. Memcached デプロイメントを定義します。

Memcached 仕様が定義された状態で、リソースの変更に対する Ansible の実行内容を定義できます。これは Ansible ロールであるため、デフォルトの動作は **roles/memcached/tasks/main.yml** ファイルでタスクを実行します。

ここでの目的は、Ansible で **memcached:1.4.36-alpine** イメージを実行するデプロイメントを作成することにあります (デプロイメントがない場合)。Ansible 2.7+ は [k8s Ansible モジュール](#) をサポートします。この例では、このモジュールを活用し、デプロイメントの定義を制御します。

roles/memcached/tasks/main.yml を以下に一致するように変更します。

```
- name: start memcached
  k8s:
    definition:
      kind: Deployment
      apiVersion: apps/v1
      metadata:
        name: '{{ meta.name }}-memcached'
        namespace: '{{ meta.namespace }}'
      spec:
        replicas: '{{size}}'
        selector:
          matchLabels:
            app: memcached
        template:
          metadata:
            labels:
              app: memcached
          spec:
            containers:
              - name: memcached
                command:
                  - memcached
                  - -m=64
                  - -o
                  - modern
                  - -v
                image: "docker.io/memcached:1.4.36-alpine"
            ports:
              - containerPort: 11211
```



注記

この例では、**size** 変数を使用し、**Memcached** デプロイメントのレプリカ数を制御しています。この例では、デフォルトを **1** に設定しますが、任意のユーザーがこのデフォルトを上書きする CR を作成することができます。

4. CRD をデプロイします。

Operator の実行前に、Kubernetes は Operator が監視する新規カスタムリソース定義 (CRD) について把握している必要があります。 **Memcached** CRD をデプロイします。

```
$ oc create -f deploy/crds/cache.example.com_memcacheds_crd.yaml
```

5. Operator をビルドし、実行します。

Operator をビルドし、実行する方法として 2 つの方法を使用できます。

- Kubernetes クラスター内の Pod を使用
- **operator-sdk up** コマンドを使用してクラスター外で Go プログラムを使用

以下の方法のいずれかを選択します。

- Kubernetes クラスター内で **Pod** として実行 します。これは実稼働環境での優先される方法です。

- memcached-operator** イメージをビルドし、これをレジストリーにプッシュします。

```
$ operator-sdk build quay.io/example/memcached-operator:v0.0.1
```

```
$ podman push quay.io/example/memcached-operator:v0.0.1
```

- Deployment マニフェストは **deploy/operator.yaml** ファイルに生成されます。このファイルの Deployment イメージは、プレースホルダー **REPLACE_IMAGE** から直前にビルドされたイメージに変更される必要があります。これを実行するには、以下を実行します。

```
$ sed -i 's|REPLACE_IMAGE|quay.io/example/memcached-operator:v0.0.1|g'
deploy/operator.yaml
```

- memcached-operator** マニフェストをデプロイします。

```
$ oc create -f deploy/service_account.yaml
```

```
$ oc create -f deploy/role.yaml
```

```
$ oc create -f deploy/role_binding.yaml
```

```
$ oc create -f deploy/operator.yaml
```

- memcached-operator** デプロイメントが稼働していることを確認します。

```
$ oc get deployment
```

```
NAME                DESIRED  CURRENT  UP-TO-DATE  AVAILABLE  AGE
memcached-operator  1        1        1           1          1m
```

- クラスター外で実行します。この方法は、デプロイメントおよびテストの速度を上げるために開発サイクル時に優先される方法です。

Ansible Runner および Ansible Runner HTTP フラグインがインストールされていることを確認します。インストールされていない場合、CR の作成時に Ansible Runner から予想しないエラーが発生します。

さらに、**watches.yaml** ファイルで参照されるロールパスがマシン上にある必要があります。通常、コンテナはディスク上のロールが置かれる場所で使用されるため、ロールは設定済みの Ansible ロールパス (例: **/etc/ansible/roles**) に手動でコピーされる必要があります。

- i. **\$HOME/.kube/config** にあるデフォルトの Kubernetes 設定ファイルを使って Operator をローカルに実行するには、以下を実行します。

```
$ operator-sdk run --local
```

提供された Kubernetes 設定ファイルを使って Operator をローカルに実行するには、以下を実行します。

```
$ operator-sdk run --local --kubeconfig=config
```

6. Memcached CR を作成します。

- a. 以下に示されるように **deploy/crds/cache_v1alpha1_memcached_cr.yaml** ファイルを変更し、**Memcached** CR を作成します。

```
$ cat deploy/crds/cache_v1alpha1_memcached_cr.yaml
```

出力例

```
apiVersion: "cache.example.com/v1alpha1"
kind: "Memcached"
metadata:
  name: "example-memcached"
spec:
  size: 3
```

```
$ oc apply -f deploy/crds/cache_v1alpha1_memcached_cr.yaml
```

- b. **memcached-operator** が CR のデプロイメントを作成できることを確認します。

```
$ oc get deployment
```

出力例

| NAME | DESIRED | CURRENT | UP-TO-DATE | AVAILABLE | AGE |
|--------------------|---------|---------|------------|-----------|-----|
| memcached-operator | 1 | 1 | 1 | 1 | 2m |
| example-memcached | 3 | 3 | 3 | 3 | 1m |

- c. Pod で 3 つのレプリカが作成されていることを確認します。

```
$ oc get pods
```

| NAME | READY | STATUS | RESTARTS | AGE |
|-----------------------------------|-------|---------|----------|-----|
| example-memcached-6fd7c98d8-7dqdr | 1/1 | Running | 0 | 1m |

```
example-memcached-6fd7c98d8-g5k7v 1/1 Running 0 1m
example-memcached-6fd7c98d8-m7vn7 1/1 Running 0 1m
memcached-operator-7cc7cfd86-vvjgk 1/1 Running 0 2m
```

7. サイズを更新します。

- a. **memcached** CR の **spec.size** フィールドを **3** から **4** に変更し、変更を適用します。

```
$ cat deploy/crds/cache_v1alpha1_memcached_cr.yaml
```

出力例

```
apiVersion: "cache.example.com/v1alpha1"
kind: "Memcached"
metadata:
  name: "example-memcached"
spec:
  size: 4
```

```
$ oc apply -f deploy/crds/cache_v1alpha1_memcached_cr.yaml
```

- b. Operator が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 deployment
```

出力例

```
NAME                DESIRED  CURRENT  UP-TO-DATE  AVAILABLE  AGE
example-memcached   4        4        4           4          5m
```

8. リソースをクリーンアップします。

```
$ oc delete -f deploy/crds/cache_v1alpha1_memcached_cr.yaml
```

```
$ oc delete -f deploy/operator.yaml
```

```
$ oc delete -f deploy/role_binding.yaml
```

```
$ oc delete -f deploy/role.yaml
```

```
$ oc delete -f deploy/service_account.yaml
```

```
$ oc delete -f deploy/crds/cache_v1alpha1_memcached_crd.yaml
```

4.2.4. K8S Ansible モジュールの使用によるアプリケーションライフサイクルの管理

Ansible を使用して Kubernetes でアプリケーションのライフサイクルを管理するには、[k8s Ansible モジュール](#)を使用できます。この Ansible モジュールにより、開発者は既存の Kubernetes リソースファイル (YAML で作成されている) を利用するか、またはネイティブの Ansible でライフサイクル管理を表現することができます。

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

このセクションでは、**k8s** Ansible モジュールの使用法を詳細に説明します。使用を開始するには、Playbook を使用してローカルワークステーションにモジュールをインストールし、これをテストしてから、Operator 内での使用を開始します。

4.2.4.1. k8s Ansible モジュールのインストール

k8s Ansible モジュールをローカルワークステーションにインストールするには、以下を実行します。

手順

1. Ansible 2.9+ をインストールします。

```
$ sudo yum install ansible
```

2. **pip** を使用して **OpenShift python クライアント** パッケージをインストールします。

```
$ sudo pip install openshift
```

```
$ sudo pip install kubernetes
```

4.2.4.2. k8s Ansible モジュールのローカルでのテスト

開発者が毎回 Operator を実行し、再ビルドするのではなく、Ansible コードをローカルマシンから実行する方が利点がある場合があります。

手順

1. **community.kubernetes** コレクションをインストールします。

```
$ ansible-galaxy collection install community.kubernetes
```

2. 新規 Ansible ベースの Operator プロジェクトを初期化します。

```
$ operator-sdk new --type ansible \
  --kind Test1 \
  --api-version test1.example.com/v1alpha1 test1-operator
```

出力例

```
Create test1-operator/tmp/init/galaxy-init.sh
Create test1-operator/tmp/build/Dockerfile
Create test1-operator/tmp/build/test-framework/Dockerfile
Create test1-operator/tmp/build/go-test.sh
Rendering Ansible Galaxy role [test1-operator/roles/test1]...
Cleaning up test1-operator/tmp/init
Create test1-operator/watches.yaml
Create test1-operator/deploy/rbac.yaml
Create test1-operator/deploy/crd.yaml
Create test1-operator/deploy/cr.yaml
```

```
Create test1-operator/deploy/operator.yaml
Run git init ...
Initialized empty Git repository in /home/user/go/src/github.com/user/opsdk/test1-
operator/.git/
Run git init done
```

```
$ cd test1-operator
```

3. 必要な Ansible ロジックを使用して **roles/test1/tasks/main.yml** ファイルを変更します。この例では、変数の切り替えと共に namespace を作成し、削除します。

```
- name: set test namespace to "{{ state }}"
  community.kubernetes.k8s:
    api_version: v1
    kind: Namespace
    state: "{{ state }}"
    name: test
    ignore_errors: true ❶
```

- ❶ **ignore_errors: true** を設定することにより、存在しないプロジェクトを削除しても失敗しません。

4. **roles/test1/defaults/main.yml** ファイルを、デフォルトで **state** を **present** に設定するように変更します。

```
state: present
```

5. 上部ディレクトリーに、**test1** ロールを含む Ansible Playbook **playbook.yml** を作成します。

```
- hosts: localhost
  roles:
    - test1
```

6. 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] *****
```

```
PROCEDURE [Gathering Facts]
```

```
*****
```

```
ok: [localhost]
```

```
Task [test1 : set test namespace to present]
```

```
changed: [localhost]
```



```
PLAY RECAP *****
localhost          : ok=2  changed=1  unreachable=0  failed=0
```

7. namespace が作成されていることを確認します。

```
$ oc get namespace
```

出力例

```
NAME      STATUS  AGE
default   Active  28d
kube-public Active  28d
kube-system Active  28d
test      Active  3s
```

8. **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] *****
```

```
PROCEDURE [Gathering Facts]
```

```
*****
```

```
ok: [localhost]
```

```
Task [test1 : set test namespace to absent]
```

```
changed: [localhost]
```

```
PLAY RECAP *****
```

```
localhost          : ok=2  changed=1  unreachable=0  failed=0
```

9. namespace が削除されていることを確認します。

```
$ oc get namespace
```

出力例

```
NAME      STATUS  AGE
default   Active  28d
kube-public Active  28d
kube-system Active  28d
```

4.2.4.3. Operator 内での k8s Ansible モジュールのテスト

k8s Ansible モジュールをローカルで使用することに慣れたら、カスタムリソース (CR) の変更時に Operator 内で同じ Ansible ロジックをトリガーできます。この例では、Ansible ロールを、Operator が

監視する特定の Kubernetes リソースにマップします。このマッピングは **watches.yaml** ファイルで実行されます。

4.2.4.3.1. Ansible ベース Operator のローカルでのテスト

Ansible ワークフローのテストをローカルで実行することに慣れたら、ローカルに実行される Ansible ベースの Operator 内でロジックをテストできます。

これを実行するには、Operator プロジェクトの上部ディレクトリーから **operator-sdk run --local** コマンドを使用します。このコマンドは **watches.yaml** ファイルから読み取り、**~/kube/config** ファイルを使用して **k8s** Ansible モジュールが実行するように Kubernetes クラスターと通信します。

手順

1. **run --local** コマンドは **watches.yaml** ファイルから読み取るため、Operator の作成者はいくつかのオプションを選択できます。**role** が単独で残される場合 (デフォルトでは **/opt/ansible/roles/<name>**)、ロールを Operator から **/opt/ansible/roles/** ディレクトリーに直接コピーする必要があります。これは、現行ディレクトリーからの変更が反映されないために複雑になります。この代わりに、**role** フィールドを現行ディレクトリーを参照するように変更し、既存の行をコメントアウトします。

```
- version: v1alpha1
  group: test1.example.com
  kind: Test1
  # role: /opt/ansible/roles/Test1
  role: /home/user/test1-operator/Test1
```

2. カスタムリソース定義 (CRD) およびカスタムリソース (CR) **Test1** の適切なロールベースアクセス制御 (RBAC) 定義を作成します。**operator-sdk** コマンドは、**deploy/** ディレクトリー内にこれらのファイルを自動生成します。

```
$ oc create -f deploy/crds/test1_v1alpha1_test1_crd.yaml
```

```
$ oc create -f deploy/service_account.yaml
```

```
$ oc create -f deploy/role.yaml
```

```
$ oc create -f deploy/role_binding.yaml
```

3. **run --local** コマンドを実行します。

```
$ operator-sdk run --local
```

出力例

```
[...]
INFO[0000] Starting to serve on 127.0.0.1:8888
INFO[0000] Watching test1.example.com/v1alpha1, Test1, default
```

4. Operator はリソース **Test1** でイベントを監視しているため、CR の作成により、Ansible ロールの実行がトリガーされます。**deploy/cr.yaml** ファイルを表示します。

```

apiVersion: "test1.example.com/v1alpha1"
kind: "Test1"
metadata:
  name: "example"

```

spec フィールドは設定されていないため、Ansible は追加の変数なしで起動します。次のセクションでは、追加の変数が CR から Ansible に渡される方法について説明します。このため、Operator に妥当なデフォルト値を設定することが重要になります。

5. デフォルト変数 **state** を **present** に設定し、**Test1** の CR インスタンスを作成します。

```
$ oc create -f deploy/cr.yaml
```

6. namespace **test** が作成されていることを確認します。

```
$ oc get namespace
```

出力例

```

NAME      STATUS  AGE
default   Active  28d
kube-public Active  28d
kube-system Active  28d
test      Active  3s

```

7. **deploy/cr.yaml** ファイルを、**state** フィールドを **absent** に設定するように変更します。

```

apiVersion: "test1.example.com/v1alpha1"
kind: "Test1"
metadata:
  name: "example"
spec:
  state: "absent"

```

8. 変更を適用し、namespace が定義されていることを確認します。

```
$ oc apply -f deploy/cr.yaml
```

```
$ oc get namespace
```

出力例

```

NAME      STATUS  AGE
default   Active  28d
kube-public Active  28d
kube-system Active  28d

```

4.2.4.3.2. Ansible ベース Operator のクラスター上でのテスト

Ansible ロジックを Ansible ベース Operator 内でローカルに実行することに慣れたら、OpenShift Container Platform などの Kubernetes クラスターの Pod 内で Operator をテストすることができます。Pod のクラスターでの実行は、実稼働環境で優先される方法です。

手順

1. **test1-operator** イメージをビルドし、これをレジストリーにプッシュします。

```
$ operator-sdk build quay.io/example/test1-operator:v0.0.1
```

```
$ podman push quay.io/example/test1-operator:v0.0.1
```

2. Deployment マニフェストは **deploy/operator.yaml** ファイルに生成されます。このファイルのデプロイメントイメージはプレースホルダーの **REPLACE_IMAGE** から以前にビルドされたイメージに変更される必要があります。これを実行するには、以下のコマンドを実行します。

```
$ sed -i 's|REPLACE_IMAGE|quay.io/example/test1-operator:v0.0.1|g' deploy/operator.yaml
```

macOS でこれらの手順を実行している場合には、代わりに以下のコマンドを実行します。

```
$ sed -i "" 's|REPLACE_IMAGE|quay.io/example/test1-operator:v0.0.1|g'
deploy/operator.yaml
```

3. **test1-operator** をデプロイします。

```
$ oc create -f deploy/crds/test1_v1alpha1_test1_crd.yaml ❶
```

❶ CRD が存在しない場合にのみ必要です。

```
$ oc create -f deploy/service_account.yaml
```

```
$ oc create -f deploy/role.yaml
```

```
$ oc create -f deploy/role_binding.yaml
```

```
$ oc create -f deploy/operator.yaml
```

4. **test1-operator** が稼働していることを確認します。

```
$ oc get deployment
```

出力例

```
NAME           DESIRED  CURRENT  UP-TO-DATE  AVAILABLE  AGE
test1-operator  1        1        1           1          1m
```

5. **test1-operator** の Ansible ログを表示できるようになります。

```
$ oc logs deployment/test1-operator
```

4.2.5. operator_sdk.util Ansible コレクションを使用したカスタムリソースのステータス管理

Ansible ベースの Operator は、カスタムリソース (CR) **status サブリソース** を以前の Ansible 実行についての一般的な情報で自動的に更新します。これには、以下のように成功したタスクおよび失敗したタスクの数と関連するエラーメッセージが含まれます。

```
status:
  conditions:
    - ansibleResult:
        changed: 3
        completion: 2018-12-03T13:45:57.13329
        failures: 1
        ok: 6
        skipped: 0
      lastTransitionTime: 2018-12-03T13:45:57Z
      message: 'Status code was -1 and not [200]: Request failed: <urlopen error [Errno 113] No route to host>'
      reason: Failed
      status: "True"
      type: Failure
    - lastTransitionTime: 2018-12-03T13:46:13Z
      message: Running reconciliation
      reason: Running
      status: "True"
      type: Running
```

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に、アプリケーションからステータスを手動で追跡することができます。

手順

1. CR ステータスをアプリケーションから手動で追跡するには、**manageStatus** フィールドを **false** に設定して **watches.yaml** ファイルを更新します。

```
- version: v1
  group: api.example.com
  kind: Test1
  role: Test1
  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 **test1** および値 **test2** 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: Test1
    name: "{{ meta.name }}"
    namespace: "{{ meta.namespace }}"
    status:
      test1: test2
```

コレクションは、新たにスキヤフォールディングされた Ansible Operator に含まれるロールの **meta/main.yml** で宣言することもできます。

```
collections:
  - operator_sdk.util
```

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:
  <snip>
  status:
    test1: test2
```

追加リソース

- Ansible ベース Operator からのユーザー主導のステータス管理を行う方法についての詳細は、[Ansible-based Operator Status Proposal for Operator SDK](#) を参照してください。

4.2.6. 追加リソース

- Operator SDK によって作成されるプロジェクトディレクトリー構造についての詳細は、[Appendices](#) を参照してください。
- [Reaching for the Stars with Ansible Operator](#) - Red Hat OpenShift Blog
- [Operator Development Guide for Red Hat Partners](#)

4.3. HELM ベース OPERATOR の作成

以下では、Operator SDK での Helm チャートのサポートについての概要を説明し、Operator 作成者を対象に、既存の Helm チャートを使用する **operator-sdk** CLI ツールで Nginx Operator をビルドし、実行する例を示します。

4.3.1. Operator SDK での Helm チャートのサポート

[Operator Framework](#) は **Operator** という Kubernetes ネイティブアプリケーションを効果的かつ自動化された拡張性のある方法で管理するためのオープンソースツールキットです。このフレームワークには Operator SDK が含まれ、これは Kubernetes API の複雑性を把握していなくても、それぞれの専門知識に基づいて Operator のブートストラップおよびビルドを実行できるように開発者を支援します。

Operator プロジェクトを生成するための Operator SDK のオプションの1つとして、Go コードを作成せずに既存の Helm チャートを使用して Kubernetes リソースを統一されたアプリケーションとしてデプロイするオプションがあります。このような Helm ベースの Operator では、変更はチャートの一部として生成される Kubernetes オブジェクトに適用されるため、ロールアウト時にロジックをほとんど必要としないステートレスなアプリケーションを使用する際に適しています。いくらか制限があるような印象を与えるかもしれませんが、Kubernetes コミュニティーがビルドする Helm チャートが急速に増加していることから分かるように、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。さらにこれは RBAC に準拠するため、実稼働環境の変更を簡単に防止することができます。

4.3.2. Operator SDK CLI のインストール

Operator SDK には、開発者による新規 Operator プロジェクトの作成、ビルドおよびデプロイを支援をする CLI ツールが含まれます。ワークステーションに SDK CLI をインストールして、独自の Operator のオーサリングを開始することができます。



注記

以下では、ローカル Kubernetes クラスターとしての [minikube](#) v0.25.0+ とパブリックレジストリーの [quay.io](#) を使用します。

4.3.2.1. GitHub リリースからのインストール

GitHub のプロジェクトから Operator SDK CLI の事前ビルドリリースのバイナリーをダウンロードし、インストールできます。

前提条件

- [Go](#) v1.13+
- **docker** v17.03+、**podman** v1.2.0+、または **buildah** v1.7+
- OpenShift CLI (**oc**) v4.5+ (インストール済み)
- Kubernetes v1.12.0+ に基づくクラスターへのアクセス

- コンテナレジストリーへのアクセス

手順

1. リリースバージョン変数を設定します。

```
$ RELEASE_VERSION=v0.17.2
```

2. リリースバイナリーをダウンロードします。

- Linux の場合

```
$ curl -OJL https://github.com/operator-framework/operator-  
sdk/releases/download/${RELEASE_VERSION}/operator-sdk-${RELEASE_VERSION}-  
x86_64-linux-gnu
```

- macOS の場合

```
$ curl -OJL https://github.com/operator-framework/operator-  
sdk/releases/download/${RELEASE_VERSION}/operator-sdk-${RELEASE_VERSION}-  
x86_64-apple-darwin
```

3. ダウンロードしたリリースのバイナリーを確認します。

- a. 提供された **.asc** ファイルをダウンロードします。

- Linux の場合

```
$ curl -OJL https://github.com/operator-framework/operator-  
sdk/releases/download/${RELEASE_VERSION}/operator-sdk-  
${RELEASE_VERSION}-x86_64-linux-gnu.asc
```

- macOS の場合

```
$ curl -OJL https://github.com/operator-framework/operator-  
sdk/releases/download/${RELEASE_VERSION}/operator-sdk-  
${RELEASE_VERSION}-x86_64-apple-darwin.asc
```

- b. バイナリーと対応する **.asc** ファイルを同じディレクトリーに置き、以下のコマンドを実行してバイナリーを確認します。

- Linux の場合

```
$ gpg --verify operator-sdk-${RELEASE_VERSION}-x86_64-linux-gnu.asc
```

- macOS の場合

```
$ gpg --verify operator-sdk-${RELEASE_VERSION}-x86_64-apple-darwin.asc
```

保守管理者のパブリックキーがワークステーションにない場合は、以下のエラーが出されます。

エラーのある出力例

■


```
$ gpg: assuming signed data in 'operator-sdk-${RELEASE_VERSION}-x86_64-apple-darwin'
$ gpg: Signature made Fri Apr  5 20:03:22 2019 CEST
$ gpg:          using RSA key <key_id> ❶
$ gpg: Can't check signature: No public key
```

❶ RSA キー文字列。

キーをダウンロードするには、以下のコマンドを実行し、**<key_id>** を直前のコマンドの出力で提供された RSA キー文字列に置き換えます。

```
$ gpg [--keyserver keys.gnupg.net] --recv-key "<key_id>" ❶
```

❶ キーサーバーが設定されていない場合、これを **--keyserver** オプションで指定します。

4. リリースバイナリーを **PATH** にインストールします。

- Linux の場合

```
$ chmod +x operator-sdk-${RELEASE_VERSION}-x86_64-linux-gnu
```

```
$ sudo cp operator-sdk-${RELEASE_VERSION}-x86_64-linux-gnu
/usr/local/bin/operator-sdk
```

```
$ rm operator-sdk-${RELEASE_VERSION}-x86_64-linux-gnu
```

- macOS の場合

```
$ chmod +x operator-sdk-${RELEASE_VERSION}-x86_64-apple-darwin
```

```
$ sudo cp operator-sdk-${RELEASE_VERSION}-x86_64-apple-darwin
/usr/local/bin/operator-sdk
```

```
$ rm operator-sdk-${RELEASE_VERSION}-x86_64-apple-darwin
```

5. CLI ツールが正しくインストールされていることを確認します。

```
$ operator-sdk version
```

4.3.2.2. Homebrew からのインストール

Homebrew を使用して SDK CLI をインストールできます。

前提条件

- [Homebrew](#)
- **docker** v17.03+、**podman** v1.2.0+、または **buildah** v1.7+

- OpenShift CLI (**oc**) v4.5+ (インストール済み)
- Kubernetes v1.12.0+ に基づくクラスターへのアクセス
- コンテナレジストリーへのアクセス

手順

1. **brew** コマンドを使用して SDK CLI をインストールします。

```
$ brew install operator-sdk
```

2. CLI ツールが正しくインストールされていることを確認します。

```
$ operator-sdk version
```

4.3.2.3. ソースを使用したコンパイルおよびインストール

Operator SDK ソースコードを取得して、SDK CLI をコンパイルし、インストールできます。

前提条件

- [Git](#)
- [Go](#) v1.13+
- **docker** v17.03+、**podman** v1.2.0+、または **buildah** v1.7+
- OpenShift CLI (**oc**) v4.5+ (インストール済み)
- Kubernetes v1.12.0+ に基づくクラスターへのアクセス
- コンテナレジストリーへのアクセス

手順

1. **operator-sdk** リポジトリのクローンを作成します。

```
$ mkdir -p $GOPATH/src/github.com/operator-framework
```

```
$ cd $GOPATH/src/github.com/operator-framework
```

```
$ git clone https://github.com/operator-framework/operator-sdk
```

```
$ cd operator-sdk
```

2. 必要なリリースブランチをチェックアウトします。

```
$ git checkout master
```

3. SDK CLI ツールをコンパイルし、インストールします。

```
$ make dep
```

```
$ make install
```

これにより、`$GOPATH/bin` に CLI バイナリー **operator-sdk** がインストールされます。

4. CLI ツールが正しくインストールされていることを確認します。

```
$ operator-sdk version
```

4.3.3. Operator SDK を使用した Helm ベースの Operator のビルド

以下の手順では、Operator SDK が提供するツールおよびライブラリーを使用して Helm チャートがサポートする単純な Nginx Operator のビルドの例について説明します。

ヒント

各チャートについて新規 Operator をビルドすることは最も効果的な方法と言えます。これにより、Helm ベースの Operator から移行して Go で完全装備の Operator を作成する場合などに、さらに多くのネイティブ動作をする Kubernetes API (例: **oc get Nginx**) の使用および柔軟性が可能になります。

前提条件

- 開発ワークステーションにインストールされる Operator SDK CLI
- **cluster-admin** パーミッションを持つアカウントを使用した Kubernetes ベースのクラスター `r v1.11.3+` (OpenShift Container Platform 4.5 など) へのアクセス
- OpenShift CLI (**oc**) `v4.5+` (インストール済み)

手順

1. **新規 Operator プロジェクトを作成します。** namespace スコープの Operator は単一 namespace でリソースを監視し、管理します。namespace スコープの Operator は柔軟性があるために優先して使用されます。これらの Operator は切り離されたアップグレード、障害対応およびモニタリングのための namespace の分離、および API 定義の差異化を可能にします。新規の Helm ベース、namespace スコープの **nginx-operator** プロジェクトを作成するには、以下のコマンドを使用します。

```
$ operator-sdk new nginx-operator \
  --api-version=example.com/v1alpha1 \
  --kind=Nginx \
  --type=helm
```

```
$ cd nginx-operator
```

これにより、とりわけ API バージョン **example.com/v1alpha1** および Kind **Nginx** の Nginx リソースを監視する目的で **nginx-operator** プロジェクトが作成されます。

2. **Operator ロジックをカスタマイズします。**
この例では、**nginx-operator** はそれぞれの **Nginx** カスタムリソース (CR) について以下の調整 (reconciliation) ロジックを実行します。

- Nginx デプロイメントを作成します (ない場合)。
- Nginx サービスを作成します (ない場合)。
- Nginx Ingress を作成します (有効にされているが存在しない場合)。
- 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** は **watches.yaml** ファイルに示されるように **Nginx** リソースイベントを監視し、指定されたチャートを使用して Helm リリースを実行します。

```
- version: v1alpha1
  group: example.com
  kind: Nginx
  chart: /opt/helm/helm-charts/nginx
```

a. **Nginx Helm チャートを確認します。**

Helm Operator プロジェクトの作成時に、Operator SDK は、単純な Nginx リリース用のテンプレートセットが含まれる Helm チャートのサンプルを作成します。

この例では、Helm チャート開発者がリリースについての役立つ情報を伝えるために使用する **NOTES.txt** テンプレートと共に、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 Chart 開発者用のドキュメント](#) を参照してください。

b. **Nginx CR 仕様を確認します。**

Helm は [値 \(value\)](#) という概念を使用して、Helm チャートの **values.yaml** ファイルに定義される Helm チャートのデフォルトをカスタマイズします。

CR 仕様に必要な値を設定し、これらのデフォルトを上書きします。例としてレプリカ数を使用することができます。

- まず、**helm-charts/nginx/values.yaml** ファイルで、チャートに **replicaCount** という値が含まれ、これがデフォルトで **1** に設定されていることを検査します。デプロイメントに 2 つの Nginx インスタンスを設定するには、CR 仕様に **replicaCount: 2** が含まれる必要があります。

deploy/crds/example.com_v1alpha1_nginx_cr.yaml ファイルを以下のように更新します。

```
apiVersion: example.com/v1alpha1
kind: Nginx
metadata:
  name: example-nginx
spec:
  replicaCount: 2
```

- 同様に、デフォルトのサービスポートは **80** に設定されます。**8080** を代わりに使用するには、サービスポートの上書きを追加して **deploy/crds/example.com_v1alpha1_nginx_cr.yaml** ファイルを再度更新します。

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

```

metadata:
  name: example-nginx
spec:
  replicaCount: 2
  service:
    port: 8080

```

Helm Operator は、**helm install -f ./overrides.yaml** コマンドが機能するように、仕様全体を values ファイルの内容のように適用します。

3. CRD をデプロイします。

Operator の実行前に、Kubernetes は Operator が監視する新規カスタムリソース定義 (CRD) について把握している必要があります。以下の CRD をデプロイします。

```
$ oc create -f deploy/crds/example_v1alpha1_nginx_crd.yaml
```

4. Operator をビルドし、実行します。

Operator をビルドし、実行する方法として 2 つの方法を使用できます。

- Kubernetes クラスター内の Pod を使用
- **operator-sdk up** コマンドを使用してクラスター外で Go プログラムを使用

以下の方法のいずれかを選択します。

- Kubernetes クラスター内で **Pod** として**実行** します。これは実稼働環境での優先される方法です。

- nginx-operator** イメージをビルドし、これをレジストリーにプッシュします。

```
$ operator-sdk build quay.io/example/nginx-operator:v0.0.1
```

```
$ podman push quay.io/example/nginx-operator:v0.0.1
```

- Deployment マニフェストは **deploy/operator.yaml** ファイルに生成されます。このファイルの Deployment イメージは、プレースホルダー **REPLACE_IMAGE** から直前にビルドされたイメージに変更される必要があります。これを実行するには、以下を実行します。

```
$ sed -i 's|REPLACE_IMAGE|quay.io/example/nginx-operator:v0.0.1|g'
deploy/operator.yaml
```

- nginx-operator** マニフェストをデプロイします。

```
$ oc create -f deploy/service_account.yaml
```

```
$ oc create -f deploy/role.yaml
```

```
$ oc create -f deploy/role_binding.yaml
```

```
$ oc create -f deploy/operator.yaml
```

- nginx-operator** デプロイメントが稼働していることを確認します。

```
$ oc get deployment
```

出力例

```
NAME                DESIRED  CURRENT  UP-TO-DATE  AVAILABLE  AGE
nginx-operator      1        1        1            1          1m
```

- b. クラスター外で実行します。この方法は、デプロイメントおよびテストの速度を上げるために開発サイクル時に優先される方法です。

watches.yaml ファイルで参照されるチャートパスがマシン上に存在している必要があります。デフォルトで、**watches.yaml** ファイルは **operator-sdk build** コマンドでビルドされる Operator イメージを使用できるようにスキャフォールディングされます。Operator を **operator-sdk run --local** コマンドで開発し、テストする場合、SDK はローカルファイルシステムでこのパスを検索します。

- i. この場所に、Helm チャートのパスを参照するシンボリックリンクを作成します。

```
$ sudo mkdir -p /opt/helm/helm-charts
```

```
$ sudo ln -s $PWD/helm-charts/nginx /opt/helm/helm-charts/nginx
```

- ii. **\$HOME/.kube/config** にあるデフォルトの Kubernetes 設定ファイルを使って Operator をローカルに実行するには、以下を実行します。

```
$ operator-sdk run --local
```

提供された Kubernetes 設定ファイルを使って Operator をローカルに実行するには、以下を実行します。

```
$ operator-sdk run --local --kubeconfig=<path_to_config>
```

5. Nginx CR をデプロイします。

これまでに変更した **Nginx** CR を適用します。

```
$ oc apply -f deploy/crds/example.com_v1alpha1_nginx_cr.yaml
```

nginx-operator が CR のデプロイメントを作成することを確認します。

```
$ oc get deployment
```

出力例

```
NAME                DESIRED  CURRENT  UP-TO-DATE  AVAILABLE  AGE
example-nginx-b9phnoz9spckcrua7ihrbkrt1  2        2        2            2          1m
```

Pod で 2 つのレプリカが作成されていることを確認します。

```
$ oc get pods
```

出力例

```
■
```

| NAME | READY | STATUS | RESTARTS | AGE |
|---|-------|---------|----------|-----|
| example-nginx-b9phnoz9spckcrua7ihrbkrt1-f8f9c875d-fjcr9 | 1/1 | Running | 0 | 1m |
| example-nginx-b9phnoz9spckcrua7ihrbkrt1-f8f9c875d-ljbzl | 1/1 | Running | 0 | 1m |

サービスポートが **8080** に設定されていることを確認します。

```
$ oc get service
```

出力例

| NAME | TYPE | CLUSTER-IP | EXTERNAL-IP | PORT(S) | AGE |
|---|-----------|------------|-------------|----------|-----|
| example-nginx-b9phnoz9spckcrua7ihrbkrt1 | ClusterIP | 10.96.26.3 | <none> | 8080/TCP | 1m |

6. `replicaCount` を更新し、ポートを削除します。

`spec.replicaCount` フィールドを **2** から **3** に変更し、**`spec.service`** フィールドを削除して、変更を適用します。

```
$ cat deploy/crds/example.com_v1alpha1_nginx_cr.yaml
```

出力例

```
apiVersion: "example.com/v1alpha1"
kind: "Nginx"
metadata:
  name: "example-nginx"
spec:
  replicaCount: 3
```

```
$ oc apply -f deploy/crds/example.com_v1alpha1_nginx_cr.yaml
```

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 deployment
```

出力例

| NAME | DESIRED | CURRENT | UP-TO-DATE | AVAILABLE | AGE |
|---|---------|---------|------------|-----------|-----|
| example-nginx-b9phnoz9spckcrua7ihrbkrt1 | 3 | 3 | 3 | 3 | 1m |

サービスポートがデフォルトの **80** に設定されていることを確認します。

```
$ oc get service
```

出力例

| NAME | TYPE | CLUSTER-IP | EXTERNAL-IP | PORT(S) | AGE |
|---|-----------|------------|-------------|---------|-----|
| example-nginx-b9phnoz9spckcrua7ihrbkrt1 | ClusterIP | 10.96.26.3 | <none> | 80/TCP | 1m |

7. リソースをクリーンアップします。

```
$ oc delete -f deploy/crds/example.com_v1alpha1_nginx_cr.yaml
```

```
$ oc delete -f deploy/operator.yaml
```

```
$ oc delete -f deploy/role_binding.yaml
```

```
$ oc delete -f deploy/role.yaml
```

```
$ oc delete -f deploy/service_account.yaml
```

```
$ oc delete -f deploy/crds/example_v1alpha1_nginx_crd.yaml
```

4.3.4. 追加リソース

- Operator SDK によって作成されるプロジェクトディレクトリー構造についての詳細は、[Appendices](#) を参照してください。
- [Operator Development Guide for Red Hat Partners](#)

4.4. クラスターサービスバージョン (CSV) の生成

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) のクラスターでの 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。

Operator SDK には、手動で定義された YAML マニフェストおよび Operator ソースファイルに含まれる情報を使用してカスタマイズされた現行 Operator プロジェクトの CSV を生成するための **generate csv** サブコマンドが含まれます。

CSV で生成されるコマンドにより、Operator の作成者が OLM について詳しく知らなくても、Operator が OLM と対話させたり、メタデータをカタログレジストリーに公開したりできます。また、Kubernetes および OLM の新機能が実装される過程で CSV 仕様は変更されるため、Operator SDK はその後の新規 CSV 機能を処理できるように更新システムを容易に拡張できるようになっています。

CSV バージョンは Operator のバージョンと同じであり、新規 CSV は Operator バージョンのアップグレード時に生成されます。Operator 作成者は **--csv-version** フラグを使用して、それらの Operator の状態を指定されたセマンティクスバージョンと共に CSV にカプセル化できます。

```
$ operator-sdk generate csv --csv-version <version>
```

このアクションはべき等であり、新規バージョンが指定されるか、または YAML マニフェストまたはソースファイルが変更される場合にのみ CSV ファイルを更新します。Operator の作成者は CSV マニフェストのほとんどのフィールドを直接変更する必要はありません。変更が必要なフィールドについて、本書で定義されています。たとえば、CSV バージョンについては **metadata.name** に組み込む必要があります。

4.4.1. CSV 生成の仕組み

Operator プロジェクトの **deploy/** ディレクトリーは、Operator をデプロイするために必要なすべてのマニフェストの標準的な場所です。Operator SDK は **deploy/** のマニフェストのデータを使用し、クラスターサービスバージョン (CSV) を作成できます。

以下がコマンドになります。

```
$ operator-sdk generate csv --csv-version <version>
```

デフォルトで、CSV YAML ファイルを **deploy/olm-catalog/** ディレクトリーに書き込みます。

3 つのタイプのマニフェストが CSV の生成に必要なになります。

- **operator.yaml**
- ***_{crd,cr}.yaml**
- RBAC ロールファイル (例: **role.yaml**)

Operator の作者にはこれらのファイルについてそれぞれ異なるバージョン管理の要件がある場合があります、**deploy/olm-catalog/csv-config.yaml** ファイルに組み込む特定のファイルを設定できます。

ワークフロー

検出される既存の CSV に応じて、またすべての設定のデフォルト値が使用されることを仮定すると、**generate csv** サブコマンドは以下のいずれかを実行します。

- 既存の場所および命名規則と同じ設定で、YAML マニフェストおよびソースファイルの利用可能なデータを使用して新規 CSV を作成します。
 - a. 更新メカニズムは、**deploy/** で既存の CSV の有無をチェックします。これが見つからない場合、ここでは **キャッシュ** と呼ばれる **ClusterServiceVersion** オブジェクトを作成し、Kubernetes API **ObjectMeta** などの Operator メタデータから派生するフィールドを簡単に設定できます。
 - b. 更新メカニズムは、**deploy/** で **Deployment** リソースなどの CSV が使用するデータが含まれるマニフェストを検索し、このデータを使ってキャッシュ内の該当する CSV フィールドを設定します。
 - c. 検索が完了したら、設定されたすべてのキャッシュフィールドが CSV YAML ファイルに書き込まれます。

または、以下を実行します。

- YAML マニフェストおよびソースファイルで利用可能なデータを使用して、現時点で事前に定義されている場所で既存の CSV を更新します。
 - a. 更新メカニズムは、**deploy/** で既存の CSV の有無をチェックします。これが見つかる場合、CSV YAML ファイルのコンテンツは CSV キャッシュにマーシャルされます。
 - b. 更新メカニズムは、**deploy/** で **Deployment** リソースなどの CSV が使用するデータが含まれるマニフェストを検索し、このデータを使ってキャッシュ内の該当する CSV フィールドを設定します。
 - c. 検索が完了したら、設定されたすべてのキャッシュフィールドが CSV YAML ファイルに書き込まれます。



注記

ファイル全体ではなく、個別の YAML フィールドが上書きされます。CSV の説明および他の生成されない部分が保持される必要があるためです。

4.4.2. CSV 設定の設定

Operator の作者者は、**deploy/olm-catalog/csv-config.yaml** ファイルでいくつかのフィールドを設定し、CSV の設定を設定できます。

| フィールド | 説明 |
|--|---|
| operator-path (文字列) | Operator リソースマニフェストファイルのパス。デフォルト: deploy/operator.yaml |
| crd-cr-path-list (string(, string)*) | CRD および CR マニフェストファイルのパス。デフォルト: [deploy/crds/*_{crd,cr}.yaml] |
| rbac-path-list (string(, string)*) | RBAC ロールマニフェストファイルのパス。デフォルト: [deploy/role.yaml] |

4.4.3. 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、人間が作成する、Operator および各種のカスタムリソース定義 (CRD) についての英語のメタデータです。

Operator 作成者はそれらの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。

表4.5 必須

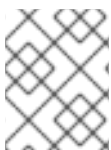
| フィールド | 説明 |
|------------------------------|--|
| metadata.name | CSV の固有名。Operator バージョンは、 app-operator.v0.1.1 などのように一意性を確保するために名前に含める必要があります。 |
| metadata.capabilities | Operator の成熟度モデルに応じた機能レベル。オプションには、 Basic Install 、 Seamless Upgrades 、 Full Lifecycle 、 Deep Insights 、および Auto Pilot が含まれます。 |
| 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。 |

| フィールド | 説明 |
|---------------------------------------|--|
| 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 |
| 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4.6 オプション

| フィールド | 説明 |
|----------------------|---|
| spec.replaces | この CSV によって置き換えられる CSV の名前。 |
| spec.links | それぞれが name および url を持つ、Operator および管理されているアプリケーションに関する URL (例: Web サイトおよびドキュメント)。 |
| spec.selector | Operator がクラスターでのリソースのペアの作成に使用するセレクター。 |
| spec.icon | mediatype で base64data フィールドに設定される、Operator に固有の base64 でエンコーディングされるアイコン。 |
| spec.maturity | 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](#)

4.4.4. CSV の生成

前提条件

- Operator プロジェクトが Operator SDK を使用して生成されている

手順

1. Operator プロジェクトで、必要な場合に **deploy/olm-catalog/csv-config.yaml** ファイルを変更して CSV 設定を設定します。
2. CSV を生成します。

```
$ operator-sdk generate csv --csv-version <version>
```

3. **deploy/olm-catalog/** ディレクトリーに生成される新規 CSV で、すべての必須で、手動で定義されたフィールドが適切に設定されていることを確認します。

4.4.5. 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、CSV が Operator がネットワークが制限された環境で適切に実行されるよう以下の追加要件を満たすことを確認する必要があります。

- 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 CSV の 2 つの場所で関連するイメージへの SHA 参照を使用する必要があります。

- **spec.relatedImages:**

```
...
spec:
  relatedImages: ❶
    - name: etcd-operator ❷
      image: quay.io/etcd-operator/operator@sha256:d134a9865524c29fcf75bbc4469013bc38d8a15cb5f41acfd6b9e492f556e4 ❸
    - name: etcd-image
      image: quay.io/etcd-operator/etcd@sha256:13348c15263bd8838ec1d5fc4550ede9860fcbb0f843e48cbccec07810e4ebb68
  ...
```

- 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。

- 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 Operator Deployment の **env** セクションで、以下を実行します。

```
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 1
                        value: quay.io/etcd-
operator/etcd@sha256:13348c15263bd8838ec1d5fc4550ede9860fcb0f843e48cbccec07810e
ebb68 2
                      - name: ETCD_LOG_LEVEL
                        value: INFO
                      image: quay.io/etcd-
operator/operator@sha256:d134a9865524c29fcf75bbc4469013bc38d8a15cb5f41acfd6b6b9e4
92f556e4 3
                    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
```

- 1 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。
- 2 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。
- 3 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。

- **Disconnected** アノテーションを探します。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。

4.4.6. 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 クラスターで利用可能な場合に、Operator が他のアーキテクチャーでのワークロードの管理をサポートするかどうかを指定できます。

Operator が AMD64 および Linux 以外のバリ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 1
  operatorframework.io/os.<os>: supported 2
```

- 1 **<arch>** をサポートされる文字列に設定します。
- 2 **<os>** をサポートされる文字列に設定します。



注記

デフォルトチャ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があります。

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。

4.4.6.1. 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4.7 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4.8 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 |
| 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。

4.4.7. 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> ❶

```

- ❶ 提案された namespace を設定します。

4.4.8. 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。

4.4.8.1. 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4.9 所有 CRD フィールド

| フィールド | 説明 | 必須/オプション |
|--------------------|--|----------|
| Name | CRD のフルネーム。 | 必須 |
| Version | オブジェクト API のバージョン。 | 必須 |
| Kind | CRD の機械可読名。 | 必須 |
| DisplayName | CRD 名の人間が判読できるバージョン (例: MongoDB Standalone)。 | 必須 |
| 説明 | Operator がこの CRD を使用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が必要とし、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.

```

4.4.8.2. 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4.10 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.
```

4.4.8.3. CRD テンプレート

Operator のユーザーは、どのオプションが必須またはオプションである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>"}}]}
```

4.4.8.4. 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トアノテーション

```
apiVersion: operators.coreos.com/v1alpha1
kind: ClusterServiceVersion
metadata:
  name: my-operator-v1.2.3
  annotations:
    operators.operatorframework.io/internal-objects:
      ["my.internal.crd1.io","my.internal.crd2.io"] ❶
  ...
```

- ❶ 内部 CRD を文字列の配列として設定します。

4.4.9. 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。

4.4.9.1. 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4.11 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 |

| フィールド | 説明 | 必須/オプション |
|---|--|----------|
| 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 |

4.4.9.1.1. 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。

4.4.9.1.2. 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。

4.4.9.2. 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依存できます。

表4.12 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 |

4.5. バンドルイメージの使用

Bundle Format を使用し、Operator SDK で Operator をパッケージ化できます。

4.5.1. バンドルイメージのビルド

Operator SDK を使用して Operator バンドルイメージをビルドし、プッシュし、検証できます。

前提条件

- Operator SDK バージョン 0.17.2
- **podman** version 1.4.4+
- Operator プロジェクトが Operator SDK を使用して生成されている

手順

1. Operator プロジェクトディレクトリーから、Operator SDK を使用してバンドルイメージをビルドします。

```
$ operator-sdk bundle create \
  quay.io/<namespace>/test-operator:v0.1.0 \1
  -b podman 2
```

- 1 バンドルイメージで使用するイメージタグ。
- 2 コンテナイメージをビルドするために使用する CLI ツール (**docker** (デフォルト)、**podman**、または **buildah** のいずれか)。この例では、**podman** を使用します。



注記

ローカルマニフェストがデフォルトの **<project_root>/deploy/olm-catalog/test-operator/manifests** がない場合、**--directory** フラグで場所を指定します。

2. バンドルイメージをプッシュするレジストリーにログインします。以下は例になります。

```
$ podman login quay.io
```

3. バンドルイメージをレジストリーにプッシュします。

```
$ podman push quay.io/<namespace>/test-operator:v0.1.0
```

4. リモートレジストリーのバンドルイメージを検証します。

```
$ operator-sdk bundle validate \
  quay.io/<namespace>/test-operator:v0.1.0 \
  -b podman
```

出力例

```
INFO[0000] Unpacked image layers          bundle-dir=/tmp/bundle-041168359
container-tool=podman
INFO[0000] running podman pull           bundle-dir=/tmp/bundle-041168359
container-tool=podman
INFO[0002] running podman save           bundle-dir=/tmp/bundle-041168359
container-tool=podman
INFO[0002] All validation tests have completed successfully bundle-dir=/tmp/bundle-
041168359 container-tool=podman
```

4.5.2. 追加リソース

- Bundle Format についての詳細は、[Operator Framework パッケージ形式](#) を参照してください。

4.6. スコアカードを使用した OPERATOR の検証

Operator の作成者は、Operator が適切にパッケージ化されていることと、構文エラーがないことを確認する必要があります。Operator の作成者は、Operator SDK のスコアカードツールを使用して Operator のパッケージ化を検証し、テストを実行できます。



注記

OpenShift Container Platform 4.5 は Operator SDK v0.17.2 をサポートします。

4.6.1. スコアカードツールについて

Operator を検証するには、Operator SDK で提供されるスコアカードツールを、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して開始します。スコアカードは、その後 API サーバーへの呼び出しを記録し、一部のテストを実行するために使用されるプロキシコンテナを Operator のデプロイメントに作成します。実行されるテストは CR の一部のパラメーターも検査します。

4.6.2. 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ファイルと、複数のグローバル設定オプションを使用します。

4.6.2.1. 設定ファイル

スコアカードツールの設定のデフォルトの場所は `<project_dir>/osdk-scorecard.*` です。以下は、YAML 形式の設定ファイルの例になります。

スコアカード設定ファイル

```
scorecard:
  output: json
  plugins:
    - basic: ❶
      cr-manifest:
        - "deploy/crds/cache.example.com_v1alpha1_memcached_cr.yaml"
        - "deploy/crds/cache.example.com_v1alpha1_memcachedrs_cr.yaml"
    - olm: ❷
      cr-manifest:
        - "deploy/crds/cache.example.com_v1alpha1_memcached_cr.yaml"
        - "deploy/crds/cache.example.com_v1alpha1_memcachedrs_cr.yaml"
      csv-path: "deploy/olm-catalog/memcached-operator/0.0.3/memcached-operator.v0.0.3.clusterserviceversion.yaml"
```

❶ 2つのカスタムリソース (CR) をテストするために設定される **basic** テスト。

❷ 2つの CR をテストするために設定された **olm** テスト。

グローバルオプションの設定方法の優先度は最も高いものから低いものへの順になります。

コマンド引数 (利用可能な場合) → 設定ファイル → デフォルト

設定ファイルは YAML 形式である必要があります。設定ファイルは、今後すべての **operator-sdk** サブコマンドの設定を許可するように拡張される可能性があるため、スコアカードの設定は **scorecard** サブセクションの下に置く必要があります。



注記

設定ファイルのサポートは **viper** パッケージで提供されます。**viper** 設定がどのように機能するかについての詳細は、[README](#) を参照してください。

4.6.2.2. コマンド引数

ほとんどのスコアカードツールの設定は設定ファイルを使用して行われますが、以下の引数を使用することもできます。

表4.13 スコアカードツール引数

| フラグ | タイプ | 説明 |
|-------------------------|--------|--|
| --bundle, -b | string | バンドル検証テストに使用するバンドルディレクトリーへのパス。 |
| --config | string | スコアカード設定ファイルへのパス。デフォルトは <project_dir>/osdk-scorecard です。ファイルタイプおよび拡張子は .yaml である必要があります。設定ファイルが指定されていないか、デフォルトの場所にある場合は、エラーを出して終了します。 |
| --output, -o | string | 出力形式有効なオプションは text および json です。デフォルトの形式は text です。これは人間が判読できることを目的として設計されています。 json 形式は、後に定義されるプラグインに使用される JSON スキーマ出力形式を使用します。 |
| --kubeconfig, -o | string | kubeconfig ファイルへのパス。内部プラグインの kubeconfig を設定します。 |
| --version | string | 実行するスコアカードのバージョン。デフォルトおよび唯一の有効なオプションは v1alpha2 です。 |
| --selector, -l | string | テストのフィルターに使用するラベルセクター。 |
| --list, -L | bool | true の場合、セクターのフィルターに基づいて実行されるテスト名のみを出力します。 |

4.6.2.3. 設定ファイルのオプション

スコアカード設定ファイルは以下のオプションを提供します。

表4.14 スコアカード設定ファイルのオプション

| オプション | 型 | 説明 |
|---------------|--------|---|
| bundle | string | --bundle フラグと同等です。Operator Lifecycle Manager (OLM) バンドルディレクトリーパス (指定されている場合) はバンドルの検証を実行します。 |

| オプション | 型 | 説明 |
|-------------------|--------|--|
| output | string | --output フラグと同等です。このオプションが設定ファイルとフラグの両方で定義されている場合、フラグの値が優先されます。 |
| kubeconfig | string | --kubeconfig フラグと同等です。このオプションが設定ファイルとフラグの両方で定義されている場合、フラグの値が優先されます。 |
| plugins | array | プラグイン名の配列。 |

4.6.2.3.1. 基本的なプラグインおよび OLM プラグイン

スコアボードは、内部の **basic** プラグインおよび **olm** プラグインをサポートします。これは、設定ファイルの **plugins** セクションで設定されます。

表4.15 プラグインオプション

| オプション | 型 | 説明 |
|---------------------|----------|---|
| cr-manifest | []string | テストされる CR のパス。 olm-deployed が設定されていないか、または false の場合に必要です。 |
| csv-path | string | Operator のクラスターサービスバージョン (CSV) へのパス。OLM テストまたは olm-deployed が true に設定されている場合に必要です。 |
| olm-deployed | bool | CSV および関連する CRD が OLM によってクラスターにデプロイされていることを示します。 |
| kubeconfig | string | kubeconfig ファイルへのパス。グローバルの kubeconfig とこのフィールドの両方が設定されている場合、このフィールドはプラグインに使用されます。 |
| namespace | string | プラグインを実行する namespace。設定されていない場合、 kubeconfig ファイルで指定されるデフォルトが使用されます。 |
| init-timeout | int | Operator の初期化時のタイムアウトまでの時間 (秒単位)。 |
| crds-dir | string | クラスターにデプロイする必要がある CRD が含まれるディレクトリーへのパス。 |

| オプション | 型 | 説明 |
|----------------------------|--------|--|
| namespaced-manifest | string | namespace 内で実行されるすべてのリソースが含まれるマニフェストファイル。デフォルトで、スコアカードは、 service_account.yaml 、 role.yaml 、 role_binding.yaml 、および operator.yaml ファイルを統合し、 deploy ディレクトリーから、namespace を使用したマニフェストとして使用する一時的なマニフェストに移動します。 |
| global-manifest | 文字列 | グローバルに実行される必須リソースが含まれるマニフェスト (namespace を使用したマニフェストではない)。デフォルトで、スコアカードは crds-dir ディレクトリーのすべての CRD を、グローバルマニフェストとして使用する一時的なマニフェストに統合します。 |



注記

現在、CSV でスコアカードを使用しても、複数の CR マニフェストを CLI、設定ファイル、または CSV アノテーションを使用して設定することはできません。Operator をクラスターで破棄し、再デプロイし、テストされる各 CR のスコアカードを再実行する必要があります。

追加リソース

- **cr-manifest** または CSV の **metadata.annotations['alm-examples']** のいずれかを設定し、CR をスコアカードに提供できますが、これらの両方を設定することはできません。詳細は、[CRD テンプレート](#) を参照してください。

4.6.3. 実行されるテスト

デフォルトでは、スコアカードツールには実行可能な内部テストのセットがあり、これらは 2 つの内部プラグインで利用できます。複数の CR がプラグインに対して指定される場合、各 CR がクリーンなテスト環境を取得できるように、テスト環境は完全にクリーンアップされます。

各テストには、テストを一意に識別する短縮名があります。これは、実行する特定のテストを選択する場合に役立ちます。以下は例になります。

```
$ operator-sdk scorecard -o text --selector=test=checkspectest
```

```
$ operator-sdk scorecard -o text --selector='test in (checkspectest,checkstatustest)'
```

4.6.3.1. 基本的なプラグイン

以下の基本的な Operator テストは、**basic** プラグインから入手できます。

表 4.16 **basic** プラグインのテスト

| テスト | 説明 | 短縮名 |
|--------------------------------|--|---|
| Spec Block Exists | このテストは、クラスターで作成されたカスタムリソース (CR) をチェックし、すべての CR に spec ブロックがあることを確認します。このテストの最大スコアは 1 です。 | checkspectest |
| Status Block Exists | このテストは、クラスターで作成された CR をチェックし、すべての CR に status ブロックがあることを確認します。このテストの最大スコアは 1 です。 | checkstatustest |
| Writing Into CRs Has An Effect | このテストは、スコアカードプロキシのログを読み取り、Operator が PUT または POST 、またはその両方を API サーバーに対して要求していることを検証します。これは、リソースが変更されていることを示します。このテストの最大スコアは 1 です。 | writingintocrsh aseffecttest |

4.6.3.2. OLM プラグイン

olm プラグインから、以下の Operator Lifecycle Manager (OLM) 統合テストを利用できます。

表4.17 **olm** プラグインのテスト

| テスト | 説明 | 短縮名 |
|----------------------------------|--|-------------------------------|
| OLM Bundle Validation | このテストは、バンドルフラグで指定されたバンドルディレクトリーにある OLM 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。 | bundlevalidationtest |
| Provided APIs Have Validation | このテストは、提供された CR の 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。このテストの最大スコアは、 cr-manifest オプションによって提供される CR 数と等しくなります。 | crdshavevalidationtest |
| 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、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 resources 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。このテストの最大スコアは、 cr-manifest オプションによって提供される CR 数と等しくなります。 | crdshaveresourcestest |

| テスト | 説明 | 短縮名 |
|--------------------------------|---|-----------------------------|
| Spec Fields With Descriptors | このテストは、カスタムリソースの 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。このテストの最大スコアは、 cr-manifest オプションで渡される各カスタムリソースの spec セクションにあるフィールドの合計数と等しくなります。 | specdescriptorstest |
| Status Fields With Descriptors | このテストは、カスタムリソースの 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。このテストの最大スコアは、 cr-manifest オプションで渡される各カスタムリソースの status セクションのフィールドの合計数と等しくなります。 | statusdescriptortest |

追加リソース

- [所有 CRD \(Owned CRD\)](#)

4.6.4. スコアカードの実行

前提条件

Operator プロジェクトの以下の前提条件は、スコアカードツールでチェックされます。

- Kubernetes 1.11.3 以降を実行するクラスターへのアクセス。
- スコアカードを使用して Operator Lifecycle Manager (OLM) で Operator プロジェクトの統合をチェックする必要がある場合、クラスターサービスバージョン (CSV) ファイルも必要になります。これは、**olm-deployed** オプションを使用する場合の要件です。
- Operator SDK を使用して生成されなかった Operator (SDK Operator 以外) の場合:
 - Operator およびカスタムリソース (CR) のインストールおよび設定用のリソースマニフェスト。
 - **clientcmd** または **controller-runtime** 設定 getter などの **KUBECONFIG** 環境変数からの読み取りをサポートする設定 getter。これは、スコアカードプロキシが正常に機能するために必要になります。

手順

1. **.osdk-scorecard.yaml** 設定ファイルを Operator プロジェクトで定義します。
2. RBAC ファイル (**role_binding**) で定義される namespace を作成します。
3. Operator プロジェクトのルートディレクトリーからスコアカードを実行します。

```
$ operator-sdk scorecard
```

実行されたテキストのいずれかがパスしなかった場合、スコアカードのリターンコードは **1** になり、選択したすべてのテストにパスすると **0** になります。

4.6.5. OLM 管理の Operator を使用したスコアカードの実行

スコアカードはクラスターサービスバージョン (CSV) を使用して実行でき、クラスター対応および Operator SDK 以外の Operator をテストする方法を提供します。

手順

1. スコアカードでは、Operator のログを読み取るために、Operator のデプロイメント Pod にプロキシコンテナが必要になります。Operator Lifecycle Manager (OLM) で Operator をデプロイする **前** に、CSV の変更および1つの追加オブジェクトの作成が必要になります。この手順は、bash 関数を使用して、手動または自動で実行できます。以下の方法のいずれかを選択します。

- 手動の方法:

- a. ローカル **kubeconfig** を含むプロキシサーバーシークレットを作成します。
 - i. スコアカードプロキシの namespace を使用した所有者参照を使用してユーザー名を生成します。

```
$ echo '{"apiVersion":"","kind":"","name":"scorecard","uid":"","Namespace":"",""}' | base64 -w 0
```

1. **<namespace>** を Operator がデプロイに使用する namespace に置き換えます。
- ii. 以下のテンプレートを使用して **Config** マニフェスト **scorecard-config.yaml** を作成し、**<username>** を直前の手順で生成される base64 ユーザー名に置き換えます。

```
apiVersion: v1
kind: Config
clusters:
- cluster:
    insecure-skip-tls-verify: true
    server: http://<username>@localhost:8889
    name: proxy-server
contexts:
- context:
    cluster: proxy-server
    user: admin/proxy-server
    name: <namespace>/proxy-server
current-context: <namespace>/proxy-server
preferences: {}
users:
- name: admin/proxy-server
  user:
    username: <username>
    password: unused
```

- iii. **Config** を base64 としてエンコードします。

```
$ cat scorecard-config.yaml | base64 -w 0
```

- iv. **Secret** マニフェストの **-secret.yaml** を作成します。

```
apiVersion: v1
kind: Secret
metadata:
  name: scorecard-kubeconfig
  namespace: <namespace> ①
data:
  kubeconfig: <kubeconfig_base64> ②
```

- ① **<namespace>** を Operator がデプロイに使用する namespace に置き換えます。

- ② **<kubeconfig_base64>** を、base64 としてエンコードされる **Config** に置き換えます。

- v. シークレットを適用します。

```
$ oc apply -f scorecard-secret.yaml
```

- vi. シークレットを参照するボリュームを Operator のデプロイメントに挿入します。

```
spec:
  install:
    spec:
      deployments:
        - name: memcached-operator
          spec:
            ...
            template:
              ...
              spec:
                containers:
                  ...
                  volumes:
                    - name: scorecard-kubeconfig ①
                      secret:
                        secretName: scorecard-kubeconfig
                        items:
                          - key: kubeconfig
                            path: config
```

- ① スコアカードの **kubeconfig** ボリューム。

- b. ボリュームマウントおよび **KUBECONFIG** 環境変数を Operator のデプロイメントの各コンテナに挿入します。

```
spec:
```



```

install:
spec:
  deployments:
  - name: memcached-operator
    spec:
      ...
      template:
        ...
        spec:
          containers:
          - name: container1
            ...
            volumeMounts:
            - name: scorecard-kubeconfig ❶
              mountPath: /scorecard-secret
            env:
            - name: KUBECONFIG ❷
              value: /scorecard-secret/config
          - name: container2 ❸
            ...

```

- ❶ スコアカードの **kubeconfig** ボリュームマウント。
- ❷ スコアカードの **kubeconfig** 環境変数。
- ❸ これと同じ手順を他のコンテナについても繰り返します。

c. スコアカードプロキシコンテナを Operator のデプロイメントに挿入します。

```

spec:
install:
spec:
  deployments:
  - name: memcached-operator
    spec:
      ...
      template:
        ...
        spec:
          containers:
          ...
          - name: scorecard-proxy ❶
            command:
            - scorecard-proxy
            env:
            - name: WATCH_NAMESPACE
              valueFrom:
                fieldRef:
                  apiVersion: v1
                  fieldPath: metadata.namespace
            image: quay.io/operator-framework/scorecard-proxy:master
            imagePullPolicy: Always
            ports:
            - name: proxy
              containerPort: 8889

```

■

1 スコアカードプロキシコンテナ。

● 自動的な方法:

community-operators リポジトリには、直前の手順を実行できるいくつかの bash 関数が含まれます。

- a. 以下の **curl** コマンドを実行します。

```
$ curl -Lo csv-manifest-modifiers.sh \
  https://raw.githubusercontent.com/operator-framework/community-
  operators/master/scripts/lib/file
```

- b. **csv-manifest-modifiers.sh** ファイルを取得します。

```
$ . ./csv-manifest-modifiers.sh
```

- c. **kubeconfig** シークレットファイルを作成します。

```
$ create_kubeconfig_secret_file scorecard-secret.yaml "<namespace>" 1
```

- 1 **<namespace>** を Operator がデプロイに使用する namespace に置き換えます。

- d. シークレットを適用します。

```
$ oc apply -f scorecard-secret.yaml
```

- e. **kubeconfig** ボリュームを挿入します。

```
$ insert_kubeconfig_volume "<csv_file>" 1
```

- 1 **<csv_file>** を、CSV マニフェストへのパスに置き換えます。

- f. **kubeconfig** シークレットマウントを挿入します。

```
$ insert_kubeconfig_secret_mount "<csv_file>"
```

- g. プロキシコンテナを挿入します。

```
$ insert_proxy_container "<csv_file>" "quay.io/operator-framework/scorecard-
  proxy:master"
```

2. プロキシコンテナの挿入後に、**Operator SDK の使用を開始する**の手順に従い、CSV およびカスタムリソース定義 (CRD) をバンドルし、Operator を OLM にデプロイします。
3. Operator が OLM にデプロイされた後に、**.osdk-scorecard.yaml** 設定ファイルを Operator プロジェクトに定義し、**csv-path: <csv_manifest_path>** および **olm-deployed** オプションの両方が設定されていることを確認します。
4. **csv-path: <csv_manifest_path>** および **olm-deployed** オプションの両方をスコアカード設定ファイルに設定した状態でスコアカードを実行します。

\$ operator-sdk scorecard

追加リソース

- [Operator Lifecycle Manager を使用した Go ベースの Operator の管理](#)

4.7. PROMETHEUS による組み込みモニターリングの設定

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ー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。

4.7.1. Prometheus Operator のサポート

[Prometheus](#) はオープンソースのシステムモニター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。

4.7.2. 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" ①
    metricsPort int32 = 8383 ②
)
...
func main() {
    ...
    // Pass metrics address to controller-runtime manager
```

```

mgr, err := manager.New(cfg, manager.Options{
    Namespace:      namespace,
    MetricsBindAddress: fmt.Sprintf("%s:%d", metricsHost, metricsPort),
})

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

```

- ① メトリクスの公開に使用されるホスト。
- ② メトリクスの公開に使用されるポート。

4.7.2.1. 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
```

4.7.3. 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ーリソースであり、**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。

4.7.3.1. 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.
    }
    ...
}
```

4.8. 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、スピリットブレインが [特定の状況](#) で生じる場合があります。詳細は、[Leader-with-lease](#) Go ドキュメントを参照してください。

デフォルトで、Operator SDK は Leader-for-life 実装を有効にします。実際のユースケースに適した選択ができるように両方のアプローチのトレードオフについて、関連する Go ドキュメントを参照してください。

以下の例は、これらの 2 つのオプションを使用する方法について説明しています。

4.8.1. Leader-for-life 選択の使用

Leader-for-life 選択の実装の場合、**leader.Become()** の呼び出しは、**memcached-operator-lock** という名前の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。

4.8.2. Leader-with-lease 選択の使用

Leader-with-lease 実装は、リーダー選択について [Manager オプション](#) を使用して有効にできます。

```
import (
    ...
    "sigs.k8s.io/controller-runtime/pkg/manager"
)

func main() {
    ...
    opts := manager.Options{
        ...
        LeaderElection: true,
        LeaderElectionID: "memcached-operator-lock"
    }
}
```

```
mgr, err := manager.New(cfg, opts)
...
}
```

Operator がクラスターで実行されていない場合、Manager はリーダー選択用の設定マップを作成するための Operator の namespace を検出できないことから開始時にエラーを返します。Manager の **LeaderElectionNamespace** オプションを設定してこの namespace を上書きできます。

4.9. OPERATOR SDK CLI リファレンス

以下では、Operator SDK CLI コマンドおよびそれらの構文について説明します。

```
$ operator-sdk <command> [<subcommand>] [<argument>] [<flags>]
```

4.9.1. build

operator-sdk build コマンドはコードをコンパイルし、実行可能プロジェクトをビルドします。**build** が完了すると、イメージはローカルコンテナエンジンを使用してビルドされます。これは次にリモートレジストリーにプッシュされる必要があります。

表4.18 build 引数

| 引数 | 説明 |
|---------|--|
| <image> | ビルドされるコンテナイメージ (例: quay.io/example/operator:v0.0.1)。 |

表4.19 build フラグ

| フラグ | 説明 |
|------------------------------------|---|
| --enable-tests (ブール) | テストバイナリーをイメージに追加することにより、クラスター内でのテストを有効にします。 |
| --namespaced-manifest (文字列) | テスト用の namespace を使用したリソースマニフェストのパス。デフォルト: deploy/operator.yaml |
| --test-location (文字列) | テストの場所。デフォルト: ./test/e2e |
| -h, --help | 使用方法についてのヘルプの出力。 |

--enable-tests が設定される場合、**build** コマンドはテストバイナリーもビルドし、これをコンテナイメージに追加して、ユーザーがテストをクラスター上で Pod として実行できるように **deploy/test-pod.yaml** ファイルを生成します。

以下は例になります。

```
$ operator-sdk build quay.io/example/operator:v0.0.1
```

出力例

```
building example-operator...
```

```
building container quay.io/example/operator:v0.0.1...
```

```
Sending build context to Docker daemon 163.9MB
```

```
Step 1/4 : FROM alpine:3.6
```

```
---> 77144d8c6bdc
```

```
Step 2/4 : ADD tmp/_output/bin/example-operator /usr/local/bin/example-operator
```

```
---> 2ada0d6ca93c
```

```
Step 3/4 : RUN adduser -D example-operator
```

```
---> Running in 34b4bb507c14
```

```
Removing intermediate container 34b4bb507c14
```

```
---> c671ec1cff03
```

```
Step 4/4 : USER example-operator
```

```
---> Running in bd336926317c
```

```
Removing intermediate container bd336926317c
```

```
---> d6b58a0fcb8c
```

```
Successfully built d6b58a0fcb8c
```

```
Successfully tagged quay.io/example/operator:v0.0.1
```

4.9.2. completion

operator-sdk completion コマンドは、CLI コマンドをより迅速に、より容易に実行できるようにシェル補完を生成します。

表4.20 **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4.21 **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
```

4.9.3. print-deps

operator-sdk print-deps コマンドは、Operator が必要とする最新の Golang パッケージおよびバージョンを出力します。これはデフォルトで単票形式 (columnar format) の出力を行います。

表4.22 print-deps フラグ

| フラグ | 説明 |
|------------------|---|
| --as-file | Gopkg.toml 形式でパッケージおよびバージョンを出力します。 |

以下は例になります。

```
$ operator-sdk print-deps --as-file
```

出力例

```
required = [
  "k8s.io/code-generator/cmd/defaulter-gen",
  "k8s.io/code-generator/cmd/deepcopy-gen",
  "k8s.io/code-generator/cmd/conversion-gen",
  "k8s.io/code-generator/cmd/client-gen",
  "k8s.io/code-generator/cmd/lister-gen",
  "k8s.io/code-generator/cmd/informer-gen",
  "k8s.io/code-generator/cmd/openapi-gen",
  "k8s.io/gengo/args",
]

[[override]]
  name = "k8s.io/code-generator"
  revision = "6702109cc68eb6fe6350b83e14407c8d7309fd1a"
...
```

4.9.4. generate

operator-sdk generate コマンドは特定のジェネレーターを起動して、必要に応じてコードを生成します。

4.9.4.1. crds

generate crds サブコマンドはカスタムリソース定義 (CRD) を生成するか、またはすでに存在する場合は **deploy/crds/__crd.yaml** でそれらを更新します。OpenAPI V3 検証 YAML は **validation** オブジェクトとして生成されます。

表4.23 generate crds フラグ

| フラグ | 説明 |
|----------------------------|--------------------------------------|
| --csv-version (文字列) | 生成する CRD バージョン。デフォルト: v1beta1 |
| -h, --help | generate crds のヘルプ |

以下は例になります。

```
$ operator-sdk generate crds
```

```
$ tree deploy/crds
```

出力例

```
├── deploy/crds/app.example.com_v1alpha1_appservice_cr.yaml
└── deploy/crds/app.example.com_appservices_crd.yaml
```

4.9.4.2. csv

csv サブコマンドは、Operator Lifecycle Manager (OLM) で使用するクラスターサービスバージョン (CSV) マニフェストを作成します。また、オプションで CRD ファイルを **deploy/olm-catalog/<operator_name>/<csv_version>** に書き込みます。

表4.24 generate csv フラグ

| フラグ | 説明 |
|-----------------------------|--|
| --csv-channel (文字列) | CSV のパッケージマニフェスト下での登録に使用する必要があるチャンネル。 |
| --csv-config (文字列) | CSV 設定ファイルへのパス。デフォルト: deploy/olm-catalog/csv-config.yaml 。 |
| --csv-version (文字列) | CSV マニフェストのセマンティックバージョン。必須。 |
| --default-channel | パッケージマニフェストのデフォルトチャンネルとして --csv-channel に渡されるチャンネルを使用します。 --csv-channel が設定されている場合にのみ有効です。 |
| --from-version (文字列) | 新規バージョンのベースとして使用する CSV マニフェストのセマンティックバージョン。 |
| --operator-name | CSV の生成時に使用する Operator 名。 |
| --update-crds | 最新の CRD マニフェストを使用して deploy/<operator_name>/<csv_version> で CRD マニフェストを更新します。 |

以下は例になります。

```
$ operator-sdk generate csv \
  --csv-version 0.1.0 \
  --update-crds
```

出力例

```
INFO[0000] Generating CSV manifest version 0.1.0
INFO[0000] Fill in the following required fields in file deploy/olm-catalog/operator-
```

```
name/0.1.0/operator-name.v0.1.0.clusterserviceversion.yaml:
spec.keywords
spec.maintainers
spec.provider
spec.labels
INFO[0000] Created deploy/olm-catalog/operator-name/0.1.0/operator-
name.v0.1.0.clusterserviceversion.yaml
```

4.9.4.3. k8s

k8s サブコマンドは、**pkg/apis/** の下のすべての CRD API の Kubernetes [code-generator](#) を実行します。現時点で、**k8s** は **deepcopy-gen** のみを実行し、すべてのカスタムリソース (CR) タイプに必要な **DeepCopy()** 関数を生成します。



注記

このコマンドは、カスタムリソースの API (**spec** および **status**) が更新されるたびに実行される必要があります。

以下は例になります。

```
$ tree pkg/apis/app/v1alpha1/
```

出力例

```
pkg/apis/app/v1alpha1/
├── appservice_types.go
├── doc.go
└── register.go
```

```
$ operator-sdk generate k8s
```

出力例

```
Running code-generation for Custom Resource (CR) group versions: [app:v1alpha1]
Generating deepcopy funcs
```

```
$ tree pkg/apis/app/v1alpha1/
```

出力例

```
pkg/apis/app/v1alpha1/
├── appservice_types.go
├── doc.go
├── register.go
└── zz_generated.deepcopy.go
```

4.9.5. new

operator-sdk new コマンドは新規の Operator アプリケーションを作成し、入力された **<project_name>** に基づいてデフォルトのプロジェクトディレクトリーのレイアウトの生成 (または スキャフォールディング) を実行します。

表4.25 new 引数

| 引数 | 説明 |
|-----------------------------|--------------|
| <project_name> | 新規プロジェクトの名前。 |

表4.26 new フラグ

| フラグ | 説明 |
|--|---|
| --api-version | <group_name>/<version> 形式の CRD API バージョン (例: app.example.com/v1alpha1)。 ansible または helm タイプで使用されます。 |
| --generate-playbook | Ansible Playbook のスケルトンを生成します。 ansible タイプで使用されます。 |
| --header-file <string> | 生成される Go ファイルのヘッダーを含むファイルへのパスです。 hack/boilerplate.go.txt にコピーされます。 |
| --helm-chart <string> | 既存の Helm チャートで Helm Operator を初期化します。 <url> 、 <repo>/<name> 、またはローカルパス。 |
| --helm-chart-repo <string> | 要求される Helm チャートのチャートリポジトリ URL。 |
| --helm-chart-version <string> | Helm チャートの特定バージョン。デフォルト: 最新バージョン。 |
| --help, -h | 使用方法およびヘルプの出力。 |
| --kind <string> | CRD の Kind (例: AppService)。 ansible または helm タイプで使用されます。 |
| --skip-git-init | ディレクトリーを Git リポジトリとして実行しません。 |
| --type | 初期化する Operator のタイプ: go 、 ansible または helm 。デフォルト: go 。 |



注記

Operator SDK v0.12.0 以降では、**--dep-manager** フラグおよび **dep** ベースのプロジェクトのサポートが削除されました。Go プロジェクトは Go モジュールを使用できるようにスキャフォールディングされています。

Go プロジェクトの使用例

```
$ mkdir $GOPATH/src/github.com/example.com/
```

```
$ cd $GOPATH/src/github.com/example.com/
```

```
$ operator-sdk new app-operator
```

Ansible プロジェクトの使用例

```
$ operator-sdk new app-operator \
  --type=ansible \
  --api-version=app.example.com/v1alpha1 \
  --kind=AppService
```

4.9.6. add

operator-sdk add コマンドは、コントローラーまたはリソースをプロジェクトに追加します。コマンドは、Operator プロジェクトのルートディレクトリーから実行される必要があります。

表4.27 add サブコマンド

| サブコマンド | 説明 |
|-------------------|---|
| api | 新規カスタムリソース (CR) の新規 API 定義を pkg/apis の下に追加し、カスタムリソース定義 (CRD) および CR ファイルを deploy/crds/ の下に生成します。API が pkg/apis/<group>/<version> にすでにある場合には、コマンドは上書きせず、エラーを返します。 |
| controller | 新規コントローラーを pkg/controller/<kind>/ の下に追加します。コントローラーは operator-sdk add api --kind=<kind> --api-version=<group/version> コマンドで pkg/apis/<group>/<version> の下にすでに定義されている必要のある CR タイプを使用することを予想します。該当する kind のコントローラーパッケージが pkg/controller/<kind> にすでに存在する場合、コマンドは上書きせず、エラーが返されます。 |
| crd | CRD および CR ファイルを追加します。 <project_name>/deploy パスがすでに存在している必要があります。 --api-version および --kind フラグが、新規 Operator アプリケーションを生成するために必要です。 <ul style="list-style-type: none"> 生成される CRD ファイル名: <project_name>/deploy/crds/<group>_<version>_<kind>_crd.yaml 生成される CR ファイル名: <project_name>/deploy/crds/<group>_<version>_<kind>_cr.yaml |

表4.28 add api フラグ

| フラグ | 説明 |
|----------------------------|--|
| --api-version (文字列) | <group_name>/<version> 形式の CRD API バージョン (例: app.example.com/v1alpha1)。 . |

| フラグ | 説明 |
|----------------------|--|
| --image (文字列) | CRD Kind (例: AppService)。 |

以下は例になります。

```
$ operator-sdk add api \
  --api-version app.example.com/v1alpha1 \
  --kind AppService
```

出力例

```
Create pkg/apis/app/v1alpha1/appservice_types.go
Create pkg/apis/addtoscheme_app_v1alpha1.go
Create pkg/apis/app/v1alpha1/register.go
Create pkg/apis/app/v1alpha1/doc.go
Create deploy/crds/app_v1alpha1_appservice_cr.yaml
Create deploy/crds/app_v1alpha1_appservice_crd.yaml
Running code-generation for Custom Resource (CR) group versions: [app:v1alpha1]
Generating deepcopy funcs
```

```
$ tree pkg/apis
```

出力例

```
pkg/apis/
├── addtoscheme_app_appservice.go
├── apis.go
├── app
│   └── v1alpha1
│       ├── doc.go
│       ├── register.go
│       └── types.go
```

```
$ operator-sdk add controller \
  --api-version app.example.com/v1alpha1 \
  --kind AppService
```

出力例

```
Create pkg/controller/appservice/appservice_controller.go
Create pkg/controller/add_appservice.go
```

```
$ tree pkg/controller
```

出力例

```
pkg/controller/
├── add_appservice.go
```

```

├── appservice
│   └── appservice_controller.go
└── controller.go

```

```

$ operator-sdk add crd \
  --api-version app.example.com/v1alpha1 \
  --kind AppService

```

出力例

```

Generating Custom Resource Definition (CRD) files
Create deploy/crds/app_v1alpha1_appservice_crd.yaml
Create deploy/crds/app_v1alpha1_appservice_cr.yaml

```

4.9.7. test

operator-sdk test コマンドは Operator をローカルでテストできます。

4.9.7.1. local

local サブコマンドは、Operator SDK のテストフレームワークを使用してビルドされた Go テストをローカルで実行します。

表4.29 test local 引数

| 引数 | 説明 |
|------------------------------------|---|
| <test_location> (文字列) | エンドツーエンド (e2e) テストファイルの場所 (例: ./test/e2e/)。 |

表4.30 test local フラグ

| フラグ | 説明 |
|------------------------------------|---|
| --kubeconfig (文字列) | クラスターの kubeconfig の場所。デフォルト: ~/ .kube/config 。 |
| --global-manifest (文字列) | グローバルリソースのマニフェストへのパス。デフォルト: deploy/crd.yaml 。 |
| --namespaced-manifest (文字列) | テスト別の namespace を使用したリソースのマニフェストへのパス。デフォルト: deploy/service_account.yaml 、 deploy/rbac.yaml 、および deploy/operator.yaml の組み合わせ。 |
| --namespace (文字列) | 空ではない場合、テストを実行する単一の namespace (例: operator-test)。デフォルト: "" |
| --go-test-flags (文字列) | go test に渡す追加の引数 (例: -f "-v -parallel=2")。 |

| フラグ | 説明 |
|----------------------|--|
| --up-local | クラスターのイメージとしてではなく、 go run を使用した Operator のローカルの実行を有効にします。 |
| --no-setup | テストリソースの作成を無効にします。 |
| --image (文字列) | namespace を使用したマニフェストで指定されたイメージとは異なる Operator イメージを使用します。 |
| -h, --help | 使用方法についてのヘルプの出力。 |

以下に例を示します。

```
$ operator-sdk test local ./test/e2e/
```

出力例

```
ok github.com/operator-framework/operator-sdk-samples/memcached-operator/test/e2e 20.410s
```

4.9.8. run

operator-sdk run コマンドは、各種の環境で Operator を起動できるオプションを提供します。

表4.31 new 引数

| 引数 | 説明 |
|---------------------------|--|
| --kubeconfig (文字列) | Kubernetes 設定ファイルへのファイルパス。デフォルト: \$HOME/.kube/config |
| --local | Operator は、 kubeconfig ファイルを使用して Kubernetes クラスターにアクセスする機能を使って Operator バイナリーをビルドしてローカルに実行されます。 |
| --namespace (文字列) | Operator が変更の有無を監視する namespace。デフォルト: default |
| --operator-flags | ローカル Operator が必要とする可能性のあるフラグ。例: --flag1 value1 --flag2=value2--local フラグのみで使用する場合 |
| -h, --help | 使用方法についてのヘルプの出力。 |

4.9.8.1. --local

--local フラグは、**kubeconfig** ファイルを使用して Kubernetes クラスターにアクセスできる機能を使って Operator バイナリーをビルドし、Operator をローカルマシンで起動します。

以下は例になります。


```
$ operator-sdk run --local \
  --kubeconfig "mycluster.kubecfg" \
  --namespace "default" \
  --operator-flags "--flag1 value1 --flag2=value2"
```

以下の例では、デフォルトの **kubeconfig**、デフォルトの namespace 環境変数を使用し、Operator のフラグを渡します。Operator フラグを使用するには、Operator がこのオプションの処理方法を認識している必要があります。たとえば、**resync-interval** フラグを認識する Operator の場合は、以下を実行します。

```
$ operator-sdk run --local --operator-flags "--resync-interval 10"
```

デフォルト以外の namespace を使用することを予定している場合は、**--namespace** フラグを使用して、Operator が作成されるカスタムリソース (CR) を監視する場所を変更します。

```
$ operator-sdk run --local --namespace "testing"
```

これが機能させるには、Operator が **WATCH_NAMESPACE** 環境変数进行处理する必要があります。これは、Operator に [ユーティリティー機能](#) の **k8sutil.GetWatchNamespace** を使用して実行できます。

4.10. 付録

4.10.1. Operator プロジェクトのスキュフォールディングレイアウト

operator-sdk CLI は、それぞれの Operator プロジェクトに多数のパッケージを生成します。以下のセクションには、生成される各ファイルおよびディレクトリーの基本的な要約が含まれます。

4.10.1.1. Go ベースプロジェクト

operator-sdk new コマンドを使用して生成される Go ベースの Operator プロジェクト (デフォルトタイプ) には、以下のディレクトリーおよびファイルが含まれます。

| ファイル/フォルダー | 目的 |
|-----------------------|---|
| cmd/ | Operator のメインプログラムである manager/main.go ファイルが含まれます。これは、すべてのカスタムリソース定義 (CRD) を pkg/apis/ の下に定義し、すべてのコントローラーを pkg/controllers/ の下で起動する新規マネージャーをインスタンス化します。 |
| pkg/apis/ | CRD の API を定義するディレクトリーツリーが含まれます。ユーザーは pkg/apis/<group>/<version>/<kind>_types.go ファイルを編集し、各リソースタイプの API を定義し、それらのパッケージをコントローラーにインポートしてリソースタイプの有無について監視することが想定されます。 |
| pkg/controller | この pkg には、コントローラーの実装が含まれます。ユーザーは pkg/controller/<kind>/<kind>_controller.go ファイルを編集し、指定された Kind のリソースタイプを処理するためのコントローラーの調整 (reconciliation) ロジックを定義することが想定されます。 |

| ファイル/フォルダー | 目的 |
|--|---|
| build/ | Operator をビルドするために使用される Dockerfile およびビルドスクリプトが含まれます。 |
| deploy/ | CRD を登録し、RBAC をセットアップし、デプロイメントとして Operator をデプロイするための各種 YAML マニフェストが含まれます。 |
| Gopkg.toml Gopkg.lock | この Operator の外部の依存関係を記述する Go Dep マニフェスト。 |
| vendor/ | このプロジェクトのインポートの条件を満たす外部の依存関係のローカルコピーが含まれる Golang vendor フォルダー。Go Dep はベンダーを直接管理します。 |

4.10.1.2. Helm ベースのプロジェクト

operator-sdk new --type helm コマンドを使用して生成される Helm ベース Operator プロジェクトには、以下のディレクトリーおよびファイルが含まれます。

| ファイル/フォルダー | 目的 |
|---------------------------------|--|
| deploy/ | CRD を登録し、RBAC をセットアップし、Deployment として Operator をデプロイするための各種 YAML マニフェストが含まれます。 |
| helm-charts/<kind> | helm create コマンドと同等のコマンドを使用して初期化された Helm チャートが含まれます。 |
| build/ | Operator をビルドするために使用される Dockerfile およびビルドスクリプトが含まれます。 |
| watches.yaml | group/version/kind (GVK) および Helm チャートの場所が含まれます。 |

第5章 RED HAT OPERATOR

5.1. CLOUD CREDENTIAL OPERATOR

目的

Cloud Credential Operator 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。

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。

注記

- Cloud Credential Operator は **kube-system/aws-creds** からの認証情報を使用します。
- Cloud Credential Operator は、**credentialsrequest** に基づいてシークレットを作成します。

5.2. クラスター認証 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](#)

5.3. 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ンリソースです。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。

5.4. 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](#)

5.5. 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
```

5.6. 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。

5.7. 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](#)

5.8. 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](#)

5.9. 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。

5.10. CLUSTER VERSION OPERATOR

目的

プロジェクト

[cluster-version-operator](#)

5.11. 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](#)

5.12. 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 デーモンセットとして 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](#)

5.13. 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クスを有効にし、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**
 - スコープ: Cluster

- CR: **etcd**
- 検証: Yes

設定オブジェクト

```
$ oc edit etcd cluster
```

5.14. 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**
 - スコープ: 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]
```

5.15. 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 の library-go フレームワークをベースとしており、Cluster Version Operator (CVO) を使用して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
```

5.16. 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 の **library-go** フレームワークをベースとしており、これは Cluster Version Operator (CVO) 経由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 メトリクスを公開します。

プロジェクト

[cluster-kube-controller-manager-operator](#)

5.17. 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 はメトリクスサービス経由で Prometheus メトリクスを公開します。

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** 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。

5.18. 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**
- マシン
- **MachineHealthCheck**

5.19. 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](#)

5.20. 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](#)

5.21. 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

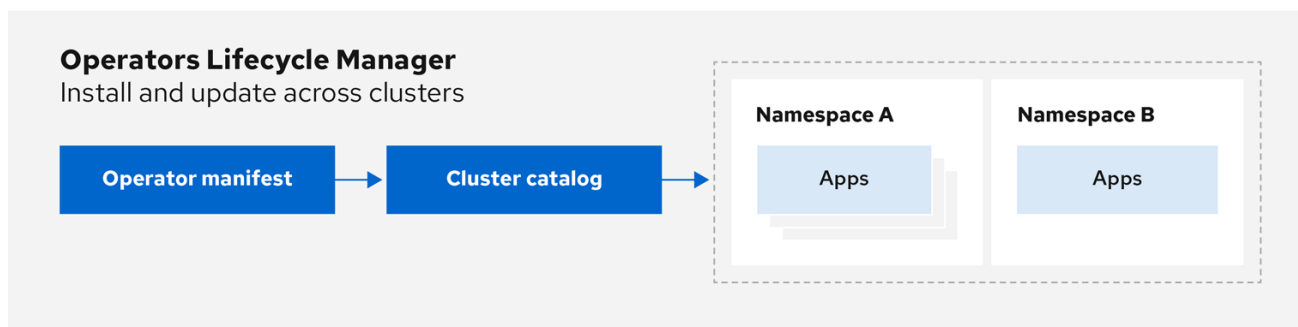
[cluster-node-tuning-operator](#)

5.22. 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5.1 Operator Lifecycle Manager ワークフロー



OpenShift_43_1019

OLM は OpenShift Container Platform 4.5 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 コンソールは、クラスター管理者が Operator をインストールしたり、クラスターで利用可能な Operator のカタログを使用できるように特定のプロジェクトアクセスを付与したりするのに使用する管理画面を提供します。

開発者の場合は、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。

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5.1 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。 |
| CatalogSource | catalog | 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。 |

| リソース | 短縮名 | 所有する Operator | 説明 |
|----------------------|-----------|---------------|--|
| 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5.2 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

詳細は、[Operator Lifecycle Manager \(OLM\)](#) についてのセクションを参照してください。

5.23. 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**
 - スコープ: Cluster
 - CR: **openshiftapiserver**

- 検証: Yes

5.24. PROMETHEUS OPERATOR

目的

Kubernetes の Prometheus Operator は、Kubernetes サービスおよび Prometheus インスタンスのデプロイメントおよび管理についての簡単なモニタリングの定義を提供します。

インストールされると、Prometheus Operator は以下の機能を提供します。

- 作成および破棄: Kubernetes namespace の Prometheus インスタンス、特定のアプリケーションまたはチームを Operator を使用して簡単に起動します。
- 単純な設定: ネイティブの Kubernetes リソースからのバージョン、永続性、保持ポリシー、レプリカなどの Prometheus の基礎的な設定を行います。
- ラベルによるサービスのターゲット設定: 従来の Kubernetes ラベルクエリーに基づいてモニタリングのターゲット設定を自動的に生成します。Prometheus 固有の設定言語を学習する必要はありません。

プロジェクト

[prometheus-operator](#)