OpenShift Container Platform 4.6

Operator

OpenShift Container Platform での Operator の使用

Last Updated: 2022-11-30

OpenShift Container Platform 4.6 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 の概要 **10**
1.1. 開発者の場合 10
1.2. 管理者の場合 10
1.3. 次のステップ 11

第2章 OPERATOR について **12**
2.1. OPERATOR について 12
2.1.1. Operator を使用する理由 12
2.1.2. Operator Framework 13
2.1.3. Operator 成熟度モデル 13
2.2. OPERATOR FRAMEWORK の一般的な用語の用語集 14
2.2.1. Common Operator Framework の一般的な用語 14
2.2.1.1. バンドル 14
2.2.1.2. バンドルイメージ 14
2.2.1.3. カタログソース 14
2.2.1.4. カタログイメージ 14
2.2.1.5. チャネル 15
2.2.1.6. チャネルヘッド 15
2.2.1.7. クラスターサービスバージョン 15
2.2.1.8. 依存関係 15
2.2.1.9. インデックスイメージ 15
2.2.1.10. インストール計画 15
2.2.1.11. Operator グループ 15
2.2.1.12. パッケージ 15
2.2.1.13. レジストリー 16
2.2.1.14. サブスクリプション 16
2.2.1.15. 更新グラフ 16
2.3. OPERATOR FRAMEWORK パッケージ形式 16
2.3.1. Bundle Format 16
2.3.1.1. マニフェスト 17
その他のサポート対象のオブジェクト 17
2.3.1.2. アノテーション 18
2.3.1.3. 依存関係ファイル 19
2.3.1.4. opm について 19
2.3.2. Package Manifest Format 20
2.4. OPERATOR LIFECYCLE MANAGER (OLM) 21
2.4.1. Operator Lifecycle Manager の概念およびリソース 21
2.4.1.1. Operator Lifecycle Manager について 21
2.4.1.2. OLM リソース 21
2.4.1.2.1. クラスターサービスバージョン 22
2.4.1.2.2. カタログソース 22
2.4.1.2.3. Subscription 25
2.4.1.2.4. インストール計画 25
2.4.1.2.5. Operator グループ 27
2.4.2. Operator Lifecycle Manager アーキテクチャー 28
2.4.2.1. コンポーネントのロール 28
2.4.2.2. OLM Operator 29
2.4.2.3. カタログ Operator 29
2.4.2.4. カタログレジストリー 30
2.4.3. Operator Lifecycle Manager ワークフロー 30
2.4.3.1. OLM での Operator のインストールおよびアップグレードのワークフロー 30

2.4.3.1.1. アップグレードパスの例 32
2.4.3.1.2. アップグレードの省略 32
2.4.3.1.3. 複数 Operator の置き換え 34
2.4.3.1.4. z-stream サポート 35
2.4.4. Operator Lifecycle Manager の依存関係の解決 36
2.4.4.1. 依存関係の解決 36
2.4.4.2. 依存関係ファイル 36
2.4.4.3. 依存関係の設定 37
2.4.4.3.1. カタログの優先順位 37
2.4.4.3.2. チャネルの順序付け 37
2.4.4.3.3. チャネル内での順序 38
2.4.4.3.4. その他の制約 38
2.4.4.4. CRD のアップグレード 38
2.4.4.5. 依存関係のベストプラクティス 38
2.4.4.6. 依存関係に関する注意事項 39
2.4.4.7. 依存関係解決のシナリオ例 40
例: 依存 API を非推奨にする 40
例: バージョンのデッドロック 40
2.4.5. Operator グループ 41
2.4.5.1. Operator グループについて 41
2.4.5.2. Operator グループメンバーシップ 41
2.4.5.3. ターゲット namespace の選択 42
2.4.5.4. Operator グループの CSV アノテーション 43
2.4.5.5. 提供される API アノテーション 43
2.4.5.6. ロールベースのアクセス制御 44
2.4.5.7. コピーされる CSV 47
2.4.5.8. 静的 Operator グループ 47
2.4.5.9. Operator グループの交差部分 47
交差のルール 48
2.4.5.10. マルチテナント Operator 管理の制限 49
2.4.5.10.1. 関連情報 49
2.4.5.11. Operator グループのトラブルシューティング 49
メンバーシップ 49
2.4.6. Operator Lifecycle Manager メトリクス 49
2.4.6.1. 公開されるメトリクス 49
2.4.7. Operator Lifecycle Manager での Webhook の管理 50
2.4.7.1. 追加リソース 50
2.5. OPERATORHUB について 50
2.5.1. OperatorHub について 50
2.5.2. OperatorHub アーキテクチャー 51
2.5.2.1. OperatorHub カスタムリソース 51
2.5.3. 関連情報 52
2.6. RED HAT が提供する OPERATOR カタログ 52
2.6.1. Operator カタログについて 52
2.6.2. Red Hat が提供する Operator カタログについて 53
2.7. CRD 54
2.7.1. カスタムリソース定義による Kubernetes API の拡張 54
2.7.1.1. カスタムリソース定義 54
2.7.1.2. カスタムリソース定義の作成 55
2.7.1.3. カスタムリソース定義のクラスターロールの作成 56
2.7.1.4. ファイルからのカスタムリソースの作成 58
2.7.1.5. カスタムリソースの検査 58
2.7.2. カスタムリソース定義からのリソースの管理 60

2.7.2.1. カスタムリソース定義 60
2.7.2.2. ファイルからのカスタムリソースの作成 60
2.7.2.3. カスタムリソースの検査 61

第3章 ユーザータスク 63
3.1. インストールされた OPERATOR からのアプリケーションの作成 63
3.1.1. Operator を使用した etcd クラスターの作成 63
3.2. NAMESPACE への OPERATOR のインストール 64
3.2.1. 前提条件 64
3.2.2. OperatorHub を使用した Operator のインストール 64
3.2.3. Web コンソールを使用した OperatorHub からのインストール 65
3.2.4. CLI を使用した OperatorHub からのインストール 66
3.2.5. Operator の特定バージョンのインストール 69

第4章 管理者タスク 71
4.1. OPERATOR のクラスターへの追加 71
4.1.1. OperatorHub を使用した Operator のインストール 71
4.1.2. Web コンソールを使用した OperatorHub からのインストール 71
4.1.3. CLI を使用した OperatorHub からのインストール 73
4.1.4. Operator の特定バージョンのインストール 76
4.2. インストールされた OPERATOR のアップグレード 77
4.2.1. Operator の更新チャネルの変更 77
4.2.2. 保留中の Operator アップグレードの手動による承認 78
4.3. クラスターからの OPERATOR の削除 78
4.3.1. Web コンソールの使用によるクラスターからの Operator の削除 78
4.3.2. CLI の使用によるクラスターからの Operator の削除 79
4.3.3. 障害のあるサブスクリプションの更新 80
4.4. OPERATOR LIFECYCLE MANAGER でのプロキシーサポートの設定 81
4.4.1. Operator のプロキシー設定の上書き 82
4.4.2. カスタム CA 証明書の挿入 83
4.5. OPERATOR ステータスの表示 85
4.5.1. Operator サブスクリプションの状態のタイプ 85
4.5.2. CLI を使用した Operator サブスクリプションステータスの表示 85
4.5.3. CLI を使った Operator カタログソースのステータス表示 86
4.6. 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 88
4.6.1. Operator インストールポリシーについて 88
4.6.1.1. インストールシナリオ 89
4.6.1.2. インストールワークフロー 89
4.6.2. Operator インストールのスコープ設定 90
4.6.2.1. 粒度の細かいパーミッション 92
4.6.3. 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 93
4.7. カスタムカタログの管理 94
4.7.1. Bundle Format を使用したカスタムカタログ 94
4.7.1.1. 前提条件 94
4.7.1.2. インデックスイメージの作成 94
4.7.1.3. インデックスイメージからのカタログの作成 95
4.7.1.4. インデックスイメージの更新 96
4.7.1.5. インデックスイメージのプルーニング 98
4.7.2. Package Manifest Format を使用したカスタムカタログ 100
4.7.2.1. Package Manifest Format カタログイメージのビルド 100
4.7.2.2. Package Manifest Format カタログイメージのミラーリング 102
4.7.2.3. Package Manifest Format カタログイメージの更新 106
4.7.2.4. Package Manifest Format カタログイメージのテスト 109

4.7.3. デフォルトの OperatorHub ソースの無効化 111
4.7.4. カスタムカタログの削除 111
4.8. ネットワークが制限された環境での OPERATOR LIFECYCLE MANAGER の使用 112
4.8.1. 前提条件 113
4.8.2. デフォルトの OperatorHub ソースの無効化 113
4.8.3. インデックスイメージのプルーニング 114
4.8.4. Operator カタログのミラーリング 116
4.8.5. インデックスイメージからのカタログの作成 121
4.8.6. インデックスイメージの更新 123

第5章 OPERATOR の開発 126
5.1. OPERATOR SDK について 126
5.1.1. Operator について 126
5.1.2. 開発ワークフロー 127
5.1.3. 関連情報 127
5.2. OPERATOR SDK CLI のインストール 127
5.2.1. GitHub リリースからの Operator SDK CLI のインストール 128
5.2.2. Homebrew からの Operator SDK CLI のインストール 130
5.2.3. ソースからの Operator SDK CLI のコンパイルおよびインストール 130
5.3. GO ベースの OPERATOR の作成 131
5.3.1. Operator SDK を使用した Go ベース Operator の作成 131
5.3.2. Operator の実行 141
5.3.2.1. クラスター外でローカルに実行する。 141
5.3.2.2. デプロイメントとしての実行 141
5.3.3. カスタムリソースの作成 142
5.3.4. 追加リソース 144
5.4. ANSIBLE ベース OPERATOR の作成 144
5.4.1. Operator SDK における Ansible サポート 144
5.4.1.1. カスタムリソースファイル 145
5.4.1.2. watches.yaml ファイル 146
5.4.1.2.1. 高度なオプション 147
5.4.1.3. Ansible に送信される追加変数 148
5.4.1.4. Ansible Runner ディレクトリー 148
5.4.2. Operator SDK を使用した Ansible ベースの Operator のビルド 149
5.4.3. 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 154
5.4.3.1. k8s Ansible モジュールのインストール 154
5.4.3.2. k8s Ansible モジュールのローカルでのテスト 155
5.4.3.3. Operator 内での k8s Ansible モジュールのテスト 157
5.4.3.3.1. Ansible ベース Operator のローカルでのテスト 157
5.4.3.3.2. Ansible ベース Operator のクラスター上でのテスト 159
5.4.4. 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 160
5.4.5. 追加リソース 162
5.5. HELM ベース OPERATOR の作成 162
5.5.1. Operator SDK での Helm チャートのサポート 162
5.5.2. Operator SDK を使用した Helm ベースの Operator のビルド 163
5.5.3. 追加リソース 168
5.6. クラスターサービスバージョン (CSV) の生成 168
5.6.1. CSV 生成の仕組み 169
ワークフロー 169
5.6.2. CSV 設定の設定 170
5.6.3. 手動で定義される CSV フィールド 170
5.6.3.1. Operator メタデータアノテーション 172
使用例 174

5.6.4. CSV の生成 174
5.6.5. ネットワークが制限された環境についての Operator の有効化 175
5.6.6. 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 177
5.6.6.1. 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 178
5.6.7. 推奨される namespace の設定 179
5.6.8. Webhook の定義 179
5.6.8.1. OLM についての Webhook の考慮事項 182
認証局についての制約 182
受付 Webhook ルールについての制約 182
変換 Webhook の制約 182
5.6.9. カスタムリソース定義 (CRD) について 183
5.6.9.1. 所有 CRD (Owned CRD) 183
5.6.9.2. 必須 CRD (Required CRD) 185
5.6.9.3. CRD のアップグレード 186
5.6.9.3.1. 新規 CRD バージョンの追加 186
5.6.9.3.2. CRD バージョンの非推奨または削除 187
5.6.9.4. CRD テンプレート 188
5.6.9.5. 内部オブジェクトの非表示 188
5.6.9.6. 必要なカスタムリソースの初期化 189
5.6.10. API サービスについて 190
5.6.10.1. 所有 API サービス 191
5.6.10.1.1. API サービスリソースの作成 192
5.6.10.1.2. API サービス提供証明書 192
5.6.10.2. 必要な API サービス 192
5.7. バンドルイメージの使用 193
5.7.1. バンドルイメージのビルド 193
5.7.2. 関連情報 195
5.8. スコアカードを使用した OPERATOR の検証 195
5.8.1. スコアカードツールについて 195
5.8.2. スコアカードの設定 195
5.8.2.1. 設定ファイル 195
5.8.2.2. コマンド引数 196
5.8.2.3. 設定ファイルのオプション 197
5.8.2.3.1. 基本的なプラグインおよび OLM プラグイン 197
5.8.3. 実行されるテスト 198
5.8.3.1. 基本的なプラグイン 198
5.8.3.2. OLM プラグイン 199
5.8.4. スコアカードの実行 200
5.8.5. OLM 管理の Operator を使用したスコアカードの実行 200
5.9. PROMETHEUS による組み込みモニターリングの設定 204
5.9.1. Prometheus Operator のサポート 205
5.9.2. メトリクスヘルパー 205
5.9.2.1. メトリクスポートの変更 206
5.9.3. サービスモニター 206
5.9.3.1. サービスモニターの作成 206
5.10. リーダー選択の設定 207
5.10.1. Operator リーダー選出の例 207
5.10.1.1. Leader-for-life 選択 208
5.10.1.2. Leader-with-lease 選択 208
5.11. OPERATOR SDK CLI リファレンス 208
5.11.1. alpha 209
5.11.1.1. scorecard 209
5.11.2. build 209

5.11.3. bundle 210
5.11.3.1. validate 210
5.11.4. cleanup 210
5.11.4.1. packagemanifests 210
5.11.5. completion 211
5.11.6. create 212
5.11.6.1. api 212
5.11.6.2. webhook 212
5.11.7. generate 212
5.11.7.1. bundle 212
5.11.7.2. kustomize 213
5.11.7.2.1. manifests 213
5.11.7.3. packagemanifests 214
5.11.8. init 215
5.11.9. new 216
5.11.10. olm 217
5.11.10.1. install 217
5.11.10.2. status 218
5.11.10.3. uninstall 218
5.11.11. run 218
5.11.11.1. packagemanifests 218
5.12. 付録 219
5.12.1. Operator プロジェクトのスキャフォールディングレイアウト 219
5.12.1.1. Ansible ベースのプロジェクト 219
5.12.1.2. Helm ベースのプロジェクト 220

第6章 RED HAT OPERATOR 221
6.1. CLOUD CREDENTIAL OPERATOR 221
目的 221
デフォルトの動作 221
モード 221
Mint モード 221
管理者レベルの認証情報の削除またはローテーション機能を持つ mint モード 223
passthrough モード 223
passthrough モードのパーミッション要件 223
passthrough モードの認証情報のメンテナーンス 224
インストール後のパーミッションの縮小 225
手動モード 225
無効にされた CCO 225
プロジェクト 225
CRD 225
設定オブジェクト 225
6.2. クラスター認証 OPERATOR 226
目的 226
プロジェクト 226
6.3. CLUSTER AUTOSCALER OPERATOR 226
目的 226
プロジェクト 226
CRD 226
6.4. CLUSTER IMAGE REGISTRY OPERATOR 226
目的 226
プロジェクト 226
6.5. クラスターモニタリング OPERATOR 227

目的 227
プロジェクト 227
CRD 227
設定オブジェクト 227
6.6. CLUSTER NETWORK OPERATOR 227
目的 227
6.7. OPENSHIFT CONTROLLER MANAGER OPERATOR 227
目的 227
プロジェクト 228
6.8. CLUSTER SAMPLES OPERATOR 228
目的 228
プロジェクト 229
6.9. CLUSTER STORAGE OPERATOR 229
目的 229
プロジェクト 229
設定 229
注記 229
6.10. CLUSTER VERSION OPERATOR 229
目的 229
プロジェクト 229
関連情報 229
6.11. CONSOLE OPERATOR 229
目的 229
プロジェクト 229
6.12. DNS OPERATOR 230
目的 230
プロジェクト 230
6.13. ETCD CLUSTER OPERATOR 230
目的 230
プロジェクト 230
CRD 230
設定オブジェクト 230
6.14. INGRESS OPERATOR 230
目的 230
プロジェクト 230
CRD 230
設定オブジェクト 231
注記 231
6.15. KUBERNETES API SERVER OPERATOR 231
目的 231
プロジェクト 231
CRD 231
設定オブジェクト 232
6.16. KUBERNETES CONTROLLER MANAGER OPERATOR 232
目的 232
プロジェクト 232
6.17. KUBERNETES SCHEDULER OPERATOR 232
目的 232
プロジェクト 232
設定 232
6.18. MACHINE API OPERATOR 233
目的 233
プロジェクト 233

CRD 233
6.19. MACHINE CONFIG OPERATOR 233
目的 233
プロジェクト 233
6.20. MARKETPLACE OPERATOR 233
目的 233
プロジェクト 233
6.21. NODE TUNING OPERATOR 234
目的 234
プロジェクト 234
6.22. OPERATOR LIFECYCLE MANAGER OPERATOR 234
目的 234
CRD 235
OLM Operator 236
カタログ Operator 236
カタログレジストリー 237
関連情報 237
6.23. OPENSHIFT API SERVER OPERATOR 237
目的 237
プロジェクト 237
CRD 237
6.24. PROMETHEUS OPERATOR 237
目的 237
プロジェクト 237
6.25. WINDOWS MACHINE CONFIG OPERATOR 237
目的 238
プロジェクト 238

第1章 OPERATOR の概要

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

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

どちらも同様の Operator の概念と目標に従いますが、OpenShift Container Platform の Operator は、目的に応じて 2 つの異なるシステムによって管理されます。

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

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

1.1. 開発者の場合

開発者は、次の Operator タスクを実行できます。

- Operator SDKCLI をインストールする。
- Go ベースの Operator、 Ansible ベースの Operator、およびHelmベースの Operator を作成する。
- Operator SDK を使用して、Operator をビルド、テスト、およびデプロイする。
- Operator をインストールして namespace にサブスクライブする。
- インストールされた Operator から Web コンソールを介してアプリケーションを作成する。

1.2. 管理者の場合

クラスター管理者は、次の Operator タスクを実行できます。

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

- プロキシーサポートを設定する
- ネットワークが制限された環境での Operator Lifecycle Manager の使用

operator-sdk CLI から生成されたファイルおよびディレクトリーについては、付録 を参照してください。

Red Hat が提供する Operator の詳細は、Red Hat Operator を参照してください。

1.3. 次のステップ

Operator の詳細はOperator とはを参照してください。

第2章 OPERATOR について

2.1. OPERATOR について

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

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

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

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

2.1.1. Operator を使用する理由

Operator は以下を提供します。

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

Kubernetes にデプロイする理由

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

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

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

Operator とサービスブローカーとの比較

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

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

2.1.2. Operator Framework

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

Operator SDK

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

Operator Lifecycle Manager

Operator Lifecycle Manager 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 4.6 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。

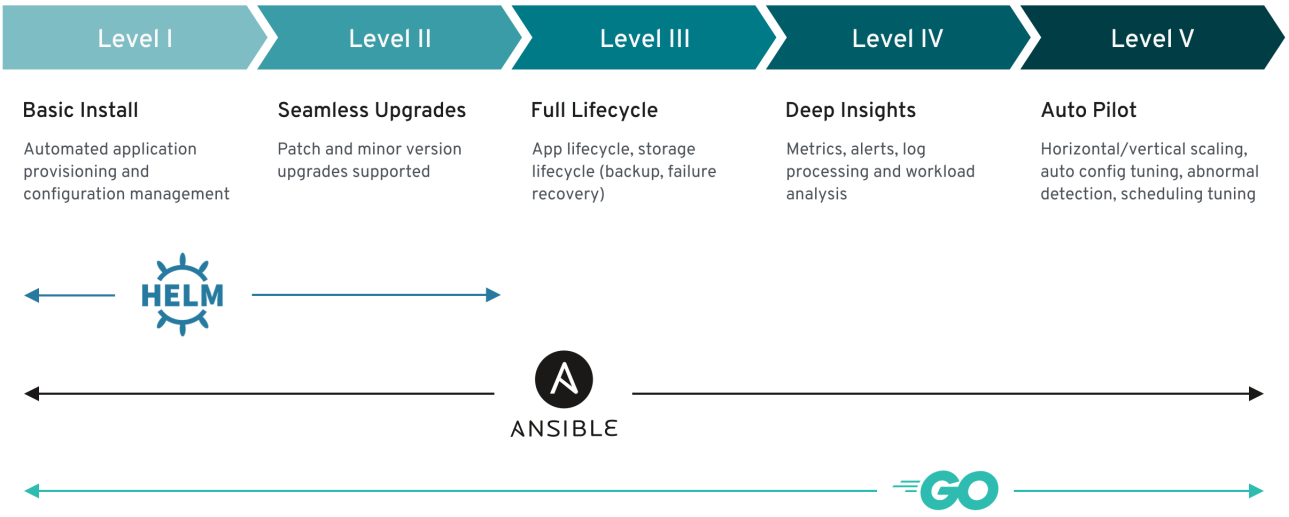
これらのツールは組み立て可能なツールとして設計されているため、役に立つと思われるツールを使用できます。

2.1.3. Operator 成熟度モデル

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

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

図2.1 Operator 成熟度モデル



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

2.2. OPERATOR FRAMEWORK 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。

2.2.1. Common Operator Framework の一般的な用語

2.2.1.1. 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。

2.2.1.2. 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ーイメージです。バンドルイメージは、Quay.io または DockerHub などの Open Container Initiative (OCI) 仕様コンテナーレジストリーによって保存され、配布されます。

2.2.1.3. 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ーです。

2.2.1.4. 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ー化されたデータストアです。

2.2.1.5. チャネル

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

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

2.2.1.6. チャネルヘッド

チャネルヘッド は、特定のチャネル内の最新の既知の更新を指します。

2.2.1.7. 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 コンテナーイメージに伴うメタデータです。

CSV は、Operator が必要とする RBAC ルールやそれが管理したり、依存したりするカスタムリソース (CR) などの Operator の実行に必要な技術情報の情報源でもあります。

2.2.1.8. 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。

2.2.1.9. 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。

2.2.1.10. 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。

2.2.1.11. 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。

2.2.1.12. 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。

2.2.1.13. 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ネルの最新バージョンおよび過去のバージョンすべてが含まれます。

2.2.1.14. サブスクリプション

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

2.2.1.15. 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。

2.3. 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。

2.3.1. 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。スケーラビリティーを向上させ、アップストリームユーザーがより効果的に独自のカタログをホストできるようにするために、Bundle Format 仕様は Operator メタデータのディストリビューションを単純化します。

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

Operator メタデータには以下を含めることができます。

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

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

- バンドルには、アノテーションで定義された 1 つ以上のチャネルが含まれる必要がある。
- すべてのバンドルには、1 つのクラスターサービスバージョン (CSV) がある。

- CSV がクラスターリソース定義 (CRD) を所有する場合、その CRD はバンドルに存在する必要がある。

2.3.1.1. マニフェスト

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

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

Bundle Format のレイアウトの例

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

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

サポート対象のオプションオブジェクトタイプ

- **ClusterRole**
- **clusterRoleBinding**
- **ConfigMap**
- **PodDisruptionBudget**
- **PriorityClass**
- **PrometheusRule**
- **ロール**
- **RoleBinding**
- **Secret**
- **Service**
- **ServiceAccount**
- **ServiceMonitor**
- **VerticalPodAutoscaler**

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

オプションオブジェクトのライフサイクル

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

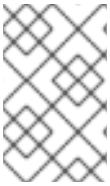
2.3.1.2. アノテーション

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

annotations.yaml の例

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

1. Operator バンドルのメディアタイプまたは形式。**registry+v1** 形式の場合、これに CSV および関連付けられた Kubernetes オブジェクトが含まれることを意味します。
2. Operator マニフェストが含まれるディレクトリーへのイメージのパス。このラベルは今後使用するために予約され、現時点ではデフォの **manifests/** に設定されています。**manifests.v1** の値は、バンドルに Operator マニフェストが含まれることを示します。
3. バンドルについてのメタデータファイルが含まれるディレクトリーへのイメージのパス。このラベルは今後使用するために予約され、現時点ではデフォの **metadata/** に設定されています。**metadata.v1** の値は、このバンドルに Operator メタデータがあることを意味します。
4. バンドルのパッケージ名。
5. Operator レジストリーに追加される際にバンドルがサブスクライブするチャネルの一覧。
6. レジストリーからインストールされる場合に Operator がサブスクライブされるデフォルトチャネル。



注記

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

2.3.1.3. 依存関係ファイル

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

依存関係の一覧には、依存関係の内容を指定するために各項目の **type** フィールドが含まれます。Operator の依存関係には、サポートされる 2 つのタイプがあります。

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

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

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

dependencies.yaml ファイルの例

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

関連情報

- Operator Lifecycle Manager の依存関係の解決

2.3.1.4. opm について

opm CLI ツールは、Operator Bundle Format で使用するために Operator Framework によって提供されます。このツールを使用して、ソフトウェアリポジトリーに相当する **index** と呼ばれるバンドルの一覧から Operator のカタログを作成し、維持することができます。結果として、**インデックスイメージ**というコンテナーイメージをコンテナーレジストリーに保存し、その後にクラスターにインストールできます。

インデックスには、コンテナーイメージの実行時に提供される組み込まれた API を使用してクエリーできる、Operator マニフェストコンテンツへのポインターのデータベースが含まれます。OpenShift Container Platform では、Operator Lifecycle Manager (OLM) はインデックスイメージを **CatalogSource** オブジェクトで参照し、これをカタログとして使用できます。これにより、クラスター上にインストールされた Operator への頻度の高い更新を可能にするためにイメージを一定の間隔でポーリングできます。

- **opm** CLI のインストール手順については、CLI ツール を参照してください。

2.3.2. 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。

2.4. OPERATOR LIFECYCLE MANAGER (OLM)

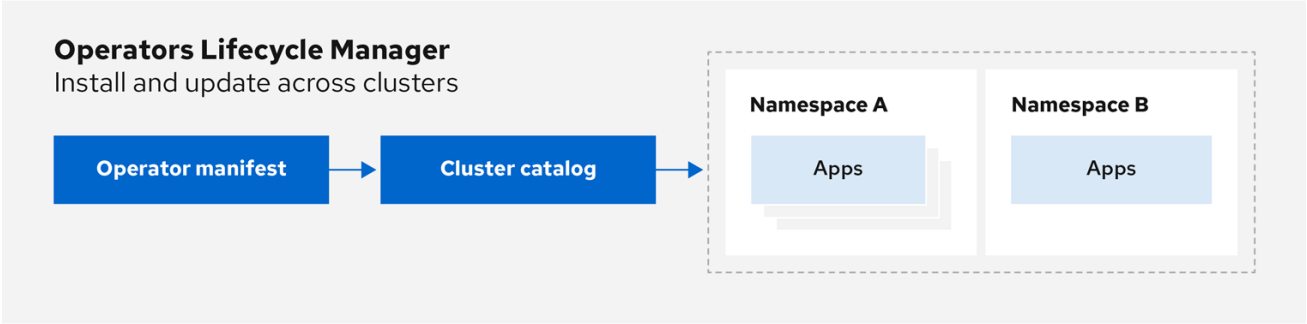
2.4.1. Operator Lifecycle Manager の概念およびリソース

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

2.4.1.1. Operator Lifecycle Manager について

Operator Lifecycle Manager (OLM) を使用することにより、ユーザーは Kubernetes ネイティブアプリケーション (Operator) および OpenShift Container Platform クラスター全体で実行される関連サービスについてインストール、更新、およびそのライフサイクルの管理を実行できます。これは、Operator を効果的かつ自動化された拡張可能な方法で管理するために設計されたオープンソースツールキットの Operator Framework の一部です。

図2.2 Operator Lifecycle Manager ワークフロー



OLM は OpenShift Container Platform 4.6 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。

2.4.1.2. OLM リソース

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

表2.1 OLM およびカタログ Operator で管理される CRD

リソース	短縮名	説明
ClusterServiceVersion (CSV)	**csv**	アプリケーションメタデータ:例: 名前、バージョン、アイコン、必須リソース。
CatalogSource	**catsrc**	CSV、CRD、およびアプリケーションを定義するパッケージのリポジトリー。
サブスクリプション	**sub**	パッケージのチャネルを追跡して CSV を最新の状態に保ちます。
InstallPlan	**ip**	CSV を自動的にインストールするか、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。

2.4.1.2.1. クラスターサービスバージョン

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

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

CSV には、ユーザーインターフェイスに名前、バージョン、説明、ラベル、リポジトリーリンクおよびロゴなどの情報を設定するために使用される Operator コンテナーイメージに伴うメタデータが含まれます。

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

2.4.1.2.2. カタログソース

カタログソース は、通常コンテナーレジストリーに保存されている **インデックスイメージ** を参照してメタデータのストアを表します。Operator Lifecycle Manager(OLM) はカタログソースをクエリーし、Operator およびそれらの依存関係を検出してインストールします。OpenShift Container Platform Web コンソールの OperatorHub は、カタログソースで提供される Operator も表示します。

ヒント

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

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

例2.1 CatalogSource オブジェクトの例

```
apiVersion: operators.coreos.com/v1alpha1
kind: CatalogSource
metadata:
  generation: 1
  name: example-catalog (1)
  namespace: openshift-marketplace (2)
spec:
  displayName: Example Catalog (3)
  image: quay.io/example-org/example-catalog:v1 (4)
  priority: -400 (5)
  publisher: Example Org
  sourceType: grpc (6)
  updateStrategy:
    registryPoll: (7)
      interval: 30m0s
status:
  connectionState:
    address: example-catalog.openshift-marketplace.svc:50051
    lastConnect: 2021-08-26T18:14:31Z
    lastObservedState: READY (8)
  latestImageRegistryPoll: 2021-08-26T18:46:25Z (9)
  registryService: (10)
    createdAt: 2021-08-26T16:16:37Z
    port: 50051
    protocol: grpc
    serviceName: example-catalog
    serviceNamespace: openshift-marketplace
```

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

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

(3) Web コンソールおよび CLI でのカタログの表示名。

(4) カタログのインデックスイメージ。

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

6 ソースタイプには以下が含まれます。

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

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

8 カタログ接続が最後に監視された状態。以下に例を示します。

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

詳細は、gRPC ドキュメントの 接続の状態 を参照してください。

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

10 カタログの Operator レジストリーサービスのステータス情報。

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

例2.2 カタログソースを参照する Subscription オブジェクトの例

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

関連情報

- OperatorHub について
- Red Hat が提供する Operator カタログ
- カタログの優先順位

- CLI を使った Operator カタログソースのステータス表示

2.4.1.2.3. Subscription

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

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

Subscription オブジェクトの例

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

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

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

OpenShift Container Platform Web コンソールから簡単に表示されるだけでなく、関連するサブスクリプションのステータスを確認して、Operator の新規バージョンが利用可能になるタイミングを特定できます。**currentCSV** フィールドに関連付けられる値は OLM に認識される最新のバージョンであり、**installedCSV** はクラスターにインストールされるバージョンです。

関連情報

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

2.4.1.2.4. インストール計画

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

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

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

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

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

例2.3 InstallPlan オブジェクトの例

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

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

関連情報

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

2.4.1.2.5. Operator グループ

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

ターゲット namespace のセットは、クラスターサービスバージョン (CSV) の **olm.targetNamespaces** アノテーションに保存されるコンマ区切りの文字列によって指定されます。このアノテーションは、メンバー Operator の CSV インスタンスに適用され、それらのデプロインメントに展開されます。

関連情報

- Operator グループ

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

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

2.4.2.1. コンポーネントのロール

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

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

表2.2 OLM およびカタログ Operator で管理される CRD

リソース	短縮名	所有する Operator	説明
ClusterServiceVersion (CSV)	**csv**	OLM	アプリケーションのメタデータ: 名前、バージョン、アイコン、必須リソース、インストールなど。
InstallPlan	**ip**	カタログ	CSV を自動的にインストールするか、またはアップグレードするために作成されるリソースの計算された一覧。
CatalogSource	**catsrc**	カタログ	CSV、CRD、およびアプリケーションを定義するパッケージのリポジトリー。
サブスクリプション	**sub**	カタログ	パッケージのチャネルを追跡して CSV を最新の状態に保つために使用されます。
OperatorGroup	**og**	OLM	**OperatorGroup** オブジェクトと同じ namespace にデプロイされたすべての Operator を、namespace の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2.3 OLM およびカタログ Operator によって作成されるリソース

リソース	所有する Operator
Deployments	OLM
ServiceAccounts	
(Cluster)Role	
(Cluster)RoleBinding	

リソース	所有する Operator
CustomResourceDefinitions (CRDs)	カタログ
ClusterServiceVersions	

2.4.2.2. OLM Operator

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

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

OLM Operator は以下のワークフローを使用します。

1. namespace でクラスターサービスバージョン (CSV) の有無を確認し、要件を満たしていることを確認します。
2. 要件が満たされている場合、CSV のインストールストラテジーを実行します。

注記

CSV は、インストールストラテジーの実行を可能にするために Operator グループのアクティブなメンバーである必要があります。

2.4.2.3. カタログ Operator

カタログ Operator はクラスターサービスバージョン (CSV) およびそれらが指定する必須リソースを解決し、インストールします。また、カタログソースでチャネル内のパッケージへの更新の有無を確認し、必要な場合はそれらを利用可能な最新バージョンに自動的にアップグレードします。

チャネル内のパッケージを追跡するために、必要なパッケージ、チャネル、および更新のプルに使用する **CatalogSource** オブジェクトを設定して **Subscription** オブジェクトを作成できます。更新が見つかると、ユーザーに代わって適切な **InstallPlan** オブジェクトの namespace への書き込みが行われます。

カタログ Operator は以下のワークフローを使用します。

1. クラスターの各カタログソースに接続します。
2. ユーザーによって作成された未解決のインストール計画の有無を確認し、これがあった場合は以下を実行します。
    a. 要求される名前に一致する CSV を検索し、これを解決済みリソースとして追加します。
    b. 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。

2.4.2.4. カタログレジストリー

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

パッケージマニフェスト は、パッケージアイデンティティーを CSV のセットに関連付けるカタログレジストリー内のエントリーです。パッケージ内で、チャネルは特定の CSV を参照します。CSV は置き換え対象の CSV を明示的に参照するため、パッケージマニフェストはカタログ Operator に対し、CSV をチャネル内の最新バージョンに更新するために必要なすべての情報を提供します (各中間バージョンをステップスルー)。

2.4.3. Operator Lifecycle Manager ワークフロー

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

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

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

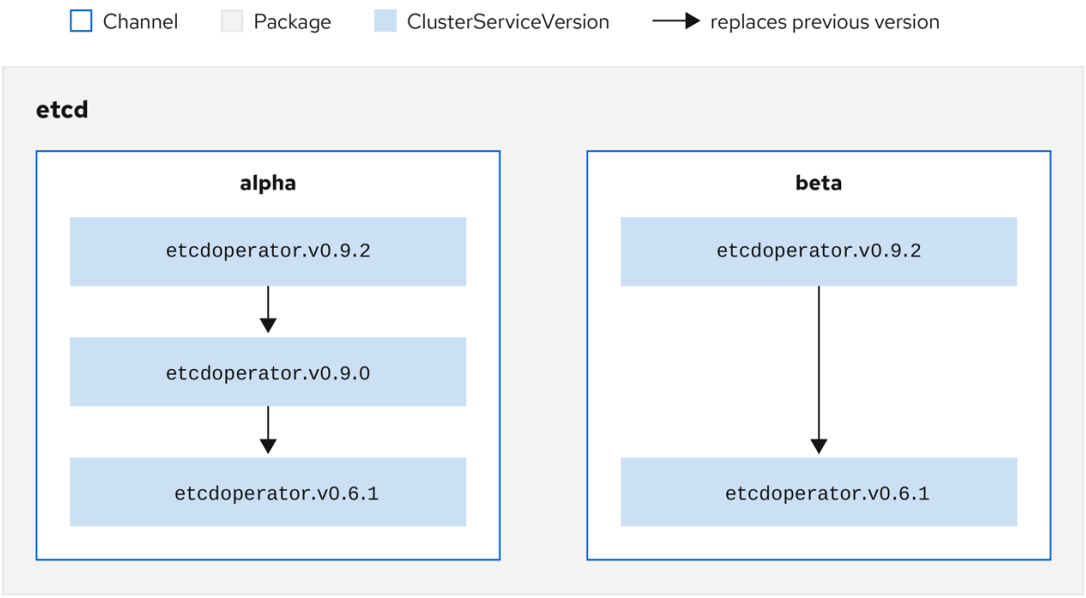
- **ClusterServiceVersion** (CSV)
- **CatalogSource**
- **サブスクリプション**

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

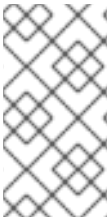
図2.3 カタログソースの概要

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

図2.4 カタログソースのパッケージおよびチャネル



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

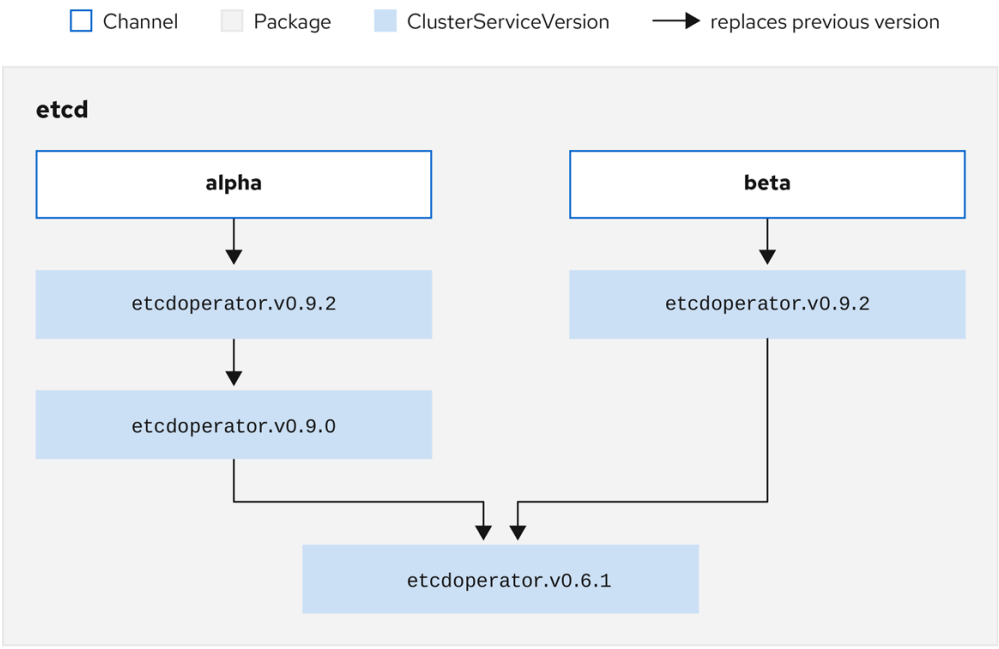


注記

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

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

図2.5 利用可能なチャネル更新についての OLM グラフ



パッケージのチャネルの例

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

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

2.4.3.1.1. アップグレードパスの例

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

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

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

2.4.3.1.2. アップグレードの省略

OLM のアップグレードの基本パスは以下の通りです。

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

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

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

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

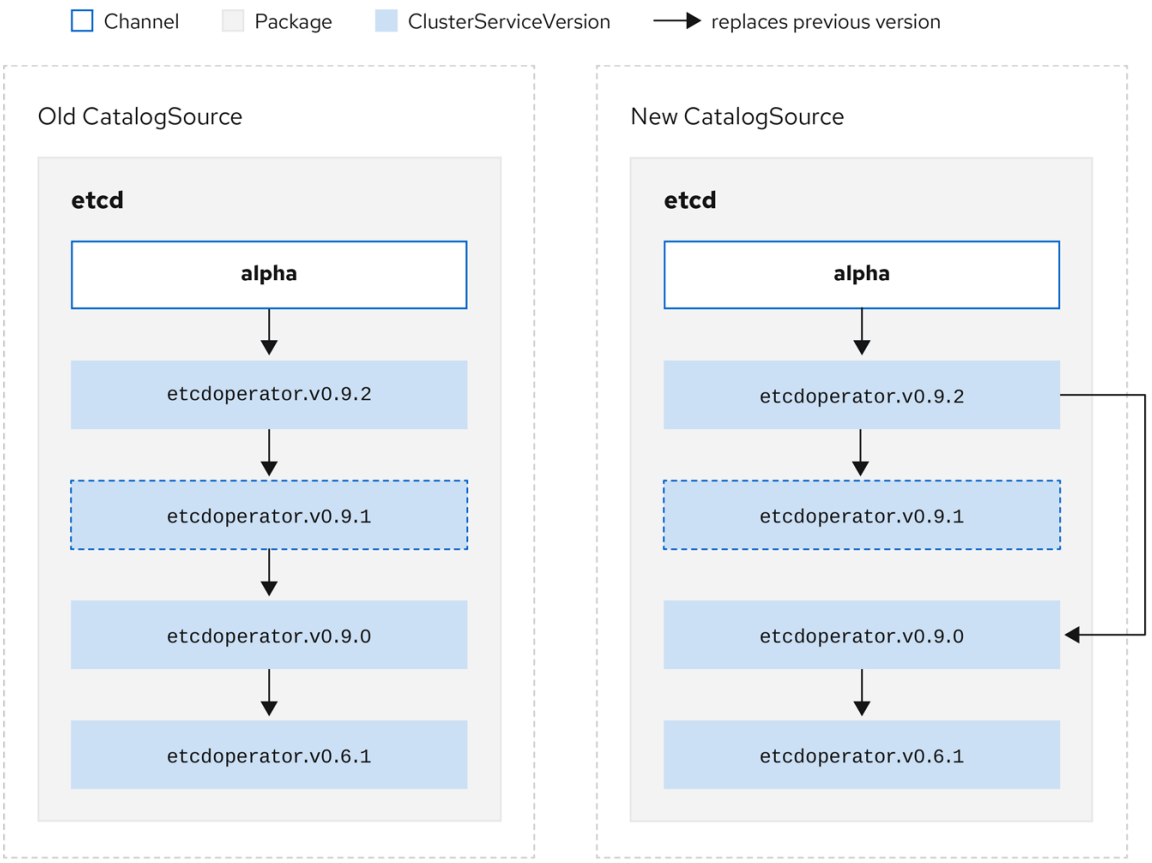
OLM は、新規カタログを送り、**省略された**リリースを追加することで、クラスターの状態や問題のある更新が発見されたかどうかにかかわらず、単一の固有の更新を常に取得することができます。

省略されたリリースの CSV 例

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

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

図2.6 更新のスキップ



このグラフは、以下を示しています。

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

2.4.3.1.3. 複数 Operator の置き換え

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

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

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

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

以下は動作が実行される順序になります。

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

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

skipRange を含む CSV の例

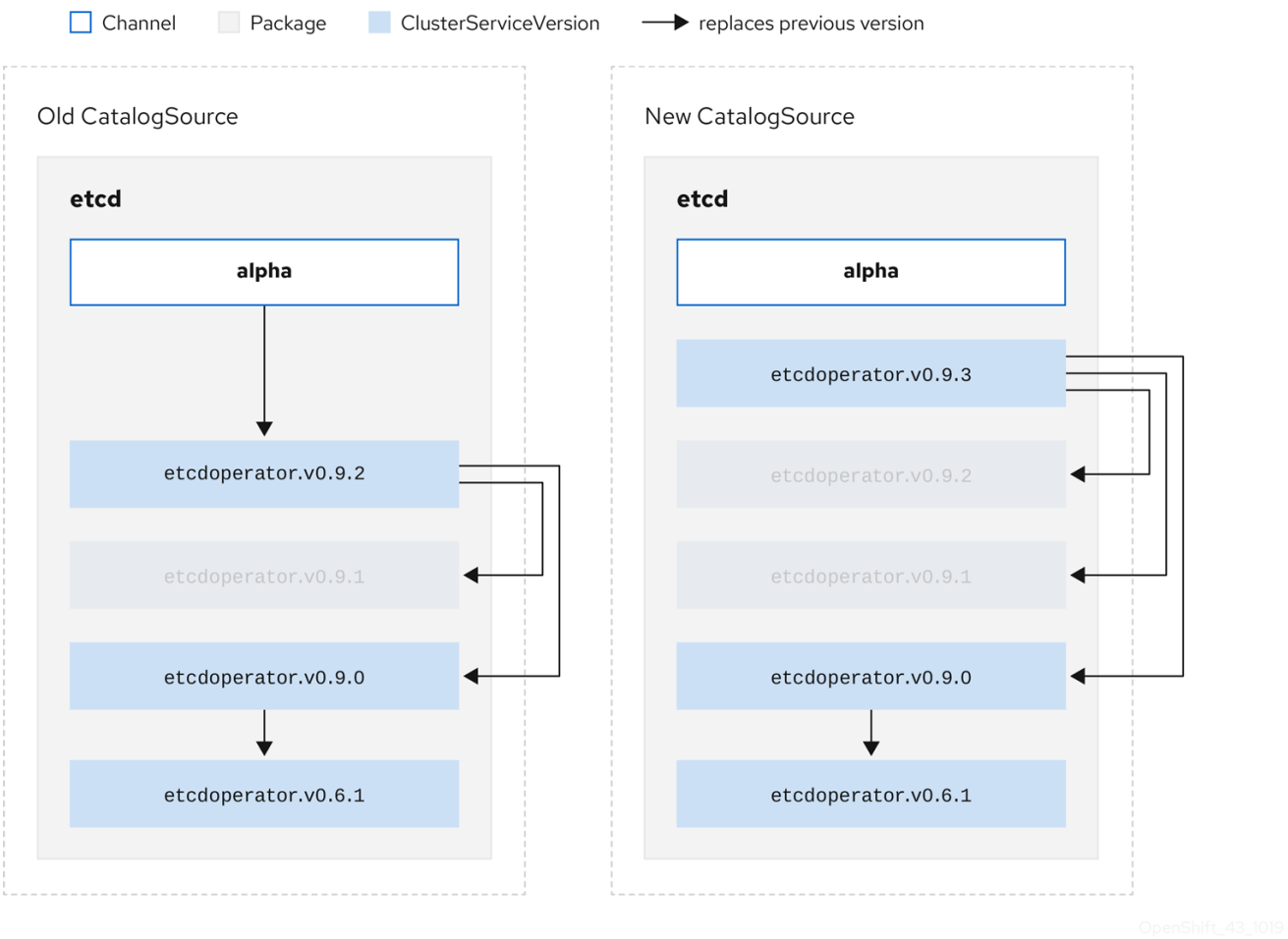
```
apiVersion: operators.coreos.com/v1alpha1
kind: ClusterServiceVersion
metadata:
  name: elasticsearch-operator.v4.1.2
  namespace: <namespace>
  annotations:
    olm.skipRange: '>=4.1.0 <4.1.2'
```

2.4.3.1.4. z-stream サポート

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

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

図2.7 複数 Operator の置き換え



このグラフは、以下を示しています。

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

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

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

2.4.4.1. 依存関係の解決

OLM は、実行中の Operator の依存関係の解決およびアップグレードライフサイクルを管理します。多くの場合、OLM が直面する問題は **yum** や **rpm** などの他のオペレーティングシステムパッケージマネージャーと同様です。

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

つまり、OLM は以下を行うことができません。

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

2.4.4.2. 依存関係ファイル

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

依存関係の一覧には、依存関係の内容を指定するために各項目の **type** フィールドが含まれます。Operator の依存関係には、サポートされる 2 つのタイプがあります。

- **olm.package**: このタイプは、特定の Operator バージョンの依存関係であることを意味します。依存関係情報には、パッケージ名とパッケージのバージョンを semver 形式で含める必要があります。たとえば、**0.5.2** などの特定バージョンや **>0.5.1** などのバージョンの範囲を指定することができます。
- **olm.gvk**: **gvk** タイプの場合、作成者は CSV の既存の CRD および API ベースの使用方法と同様に group/version/kind (GVK) 情報で依存関係を指定できます。これは、Operator の作成者がすべての依存関係、API または明示的なバージョンを同じ場所に配置できるようにするパスです。

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

dependencies.yaml ファイルの例

```
dependencies:
```

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

2.4.4.3. 依存関係の設定

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

2.4.4.3.1. カタログの優先順位

OpenShift Container Platform クラスターでは、OLM はカタログソースを読み取り、インストールに使用できる Operator を確認します。

CatalogSource オブジェクトの例

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

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

カタログ設定を規定する 2 つのルールがあります。

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

2.4.4.3.2. チャネルの順序付け

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

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

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

2.4.4.3.3. チャネル内での順序

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

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

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

2.4.4.3.4. その他の制約

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

2.4.4.3.4.1. サブスクリプションの制約

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

2.4.4.3.4.2. パッケージの制約

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

2.4.4.4. CRD のアップグレード

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

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

関連情報

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

2.4.4.5. 依存関係のベストプラクティス

依存関係を指定する際には、ベストプラクティスを考慮する必要があります。

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

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

最小バージョンの設定

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

以下に例を示します。

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

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

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

最大バージョンを省略するか、または幅広いバージョンを許可します。

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

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

注記

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

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

関連情報

- Kubernetes ドキュメント: Changing the API

2.4.4.6. 依存関係に関する注意事項

依存関係を指定する際には、考慮すべき注意事項があります。

複合制約がない (AND)

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

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

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

namespace 間の互換性

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

2.4.4.7. 依存関係解決のシナリオ例

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

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

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

この結果は以下のようになります。

- B にはプロバイダーがなくなる。
- A は機能しなくなる。

これは OLM がアップグレードストラテジーで回避するケースです。

例: バージョンのデッドロック
A および B は API である:

- A のプロバイダーは B を必要とする。

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

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

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

2.4.5. Operator グループ

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

2.4.5.1. Operator グループについて

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

ターゲット namespace のセットは、クラスターサービスバージョン (CSV) の **olm.targetNamespaces** アノテーションに保存されるコンマ区切りの文字列によって指定されます。このアノテーションは、メンバー Operator の CSV インスタンスに適用され、それらのデプロインメントに展開されます。

2.4.5.2. Operator グループメンバーシップ

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

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

CSV のインストールモードは **InstallModeType** フィールドおよびブール値の **Supported** フィールドで設定されます。CSV の仕様には、4 つの固有の **InstallModeTypes** のインストールモードのセットを含めることができます。

表2.4 インストールモードおよびサポートされる Operator グループ

InstallMode タイプ	説明
OwnNamespace	Operator は、独自の namespace を選択する Operator グループのメンバーにすることができます。
SingleNamespace	Operator は 1 つの namespace を選択する Operator グループのメンバーにすることができます。
MultiNamespace	Operator は複数の namespace を選択する Operator グループのメンバーにすることができます。

InstallMode タイプ	説明
AllNamespaces	Operator はすべての namespace を選択する Operator グループのメンバーにすることができます (設定されるターゲット namespace は空の文字列 **""** です)。

注記

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

2.4.5.3. ターゲット namespace の選択

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

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

または、**spec.selector** パラメーターでラベルセレクターを使用して namespace を指定することもできます。

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

重要

spec.targetNamespaces で複数の namespace を一覧表示したり、**spec.selector** でラベルセレクターを使用したりすることは推奨されません。Operator グループの複数のターゲット namespace のサポートは今後のリリースで取り除かれる可能性があります。

spec.targetNamespaces と **spec.selector** の両方が定義されている場合、 **spec.selector** は無視されます。または、**spec.selector** と **spec.targetNamespaces** の両方を省略し、**global** Operator グループを指定できます。これにより、すべての namespace が選択されます。

```
apiVersion: operators.coreos.com/v1
kind: OperatorGroup
metadata:
```

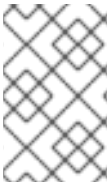
```
  name: my-group
  namespace: my-namespace
```

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

2.4.5.4. Operator グループの CSV アノテーション

Operator グループのメンバー CSV には以下のアノテーションがあります。

アノテーション	説明
olm.operatorGroup=<group_name>	Operator グループの名前が含まれます。
olm.operatorNamespace=<group_namespace>	Operator グループの namespace が含まれます。
olm.targetNamespaces=<target_namespaces>	Operator グループのターゲット namespace 選択を一覧表示するコンマ区切りの文字列が含まれます。



注記

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

2.4.5.5. 提供される API アノテーション

group/version/kind(GVK) は Kubernetes API の一意の識別子です。Operator グループによって提供される GVK についての情報が **olm.providedAPIs** アノテーションに表示されます。アノテーションの値は、コンマで区切られた **<kind>.<version>.<group>** で設定される文字列です。Operator グループのすべてのアクティブメンバーの CSV によって提供される CRD および API サービスの GVK が含まれます。

PackageManifest リースを提供する単一のアクティブメンバー CSV を含む **OperatorGroup** オブジェクトの以下の例を確認してください。

```
apiVersion: operators.coreos.com/v1
kind: OperatorGroup
metadata:
  annotations:
    olm.providedAPIs: PackageManifest.v1alpha1.packages.apps.redhat.com
  name: olm-operators
  namespace: local
  ...
spec:
  selector: {}
  serviceAccount:
    metadata:
      creationTimestamp: null
  targetNamespaces:
```

```
  - local
status:
  lastUpdated: 2019-02-19T16:18:28Z
  namespaces:
  - local
```

2.4.5.6. ロールベースのアクセス制御

Operator グループの作成時に、3 つのクラスター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レクターがラベルに一致するように設定された単一の集計ルールが含まれます。

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

以下の RBAC リソースは、CSV が **AllNamespaces** インストールモードのあるすべての namespace を監視しており、理由が **InterOperatorGroupOwnerConflict** の失敗状態にない限り、CSV が Operator グループのアクティブメンバーになる際に生成されます。

- CRD からの各 API リソースのクラスターロール
- API サービスからの各 API リソースのクラスターロール
- 追加のロールおよびロールバインディング

表2.5 CRD からの各 API リソース用に生成されたクラスターロール

クラスターロール	設定
<kind>.<group>-<version>-admin	**<kind>** の動詞 - * 集計ラベル: - **rbac.authorization.k8s.io/aggregate-to-admin: true** - **olm.opgroup.permissions/aggregate-to-admin: <operatorgroup_name>**

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

表2.6 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
<kind>.<group>-<version>-admin	**<kind>** の動詞 • ***** 集計ラベル: • **rbac.authorization.k8s.io/aggregate-to-admin: true** • **olm.opgroup.permissions/aggregate-to-admin: <operatorgroup_name>**
<kind>.<group>-<version>-edit	**<kind>** の動詞 • **create** • **update** • **patch** • **delete** 集計ラベル: • **rbac.authorization.k8s.io/aggregate-to-edit: true** • **olm.opgroup.permissions/aggregate-to-edit: <operatorgroup_name>**
<kind>.<group>-<version>-view	**<kind>** の動詞 • **get** • **list** • **watch** 集計ラベル: • **rbac.authorization.k8s.io/aggregate-to-view: true** • **olm.opgroup.permissions/aggregate-to-view: <operatorgroup_name>**

追加のロールおよびロールバインディング

- CSV が * が含まれる 1 つのターゲット namespace を定義する場合、クラスターロールと対応するクラスターロールバインディングが CSV の **permissions** フィールドに定義されるパーミッションごとに生成されます。生成されたすべてのリソースには **olm.owner: <csv_name>** および **olm.owner.namespace: <csv_namespace>** ラベルが付与されます。
- CSV が * が含まれる 1 つのターゲット namespace を定義 **しない** 場合、**olm.owner:**

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

2.4.5.7. コピーされる CSV

OLM は、それぞれの Operator グループのターゲット namespace の Operator グループのすべてのアクティブな CSV のコピーを作成します。コピーされる CSV の目的は、ユーザーに対して、特定の Operator が作成されるリソースを監視するように設定されたターゲット namespace について通知することにあります。

コピーされる CSV にはステータスの理由 **Copied** があり、それらのソース CSV のステータスに一致するように更新されます。**olm.targetNamespaces** アノテーションは、クラスター上でコピーされる CSV が作成される前に取られます。ターゲット namespace 選択を省略すると、テナント間のターゲット namespace の重複が回避されます。

コピーされる CSV はそれらのソース CSV が存在しなくなるか、またはそれらのソース CSV が属する Operator グループが、コピーされた CSV の namespace をターゲットに設定しなくなると削除されます。

2.4.5.8. 静的 Operator グループ

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

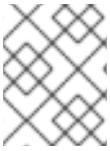
以下は、**something.cool.io/cluster-monitoring: "true"** アノテーションのあるすべての namespace の **Prometheus** リソースを保護する Operator グループの例です。

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

2.4.5.9. Operator グループの交差部分

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

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



注記

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

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

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

注記

Operator グループによって生じる失敗状態は非終了状態です。

以下のアクションは、Operator グループが同期するたびに実行されます。

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

- クラスターセットは **olm.providedAPIs** と比較され、**olm.providedAPIs** に追加の API が含まれる場合は、それらの API がプルーニングされます。

- すべての namespace で同じ API を提供するすべての CSV は再びキューに入れられます。これにより、交差部分のあるグループ間の競合する CSV に対して、それらの競合が競合する CSV のサイズ変更または削除のいずれかによって解決されている可能性があることが通知されます。

2.4.5.10. マルチテナント Operator 管理の制限

OpenShift Container Platform では、クラスターに異なる Operator のバージョンを同時にインストールする場合のサポートは限定されます。Operator はコントロールプレーンの拡張機能です。すべてのテナントまたは namespace がクラスターの同じコントロールプレーンを共有します。そのため、マルチテナント環境のテナントも Operator を共有する必要があります。

Operator Lifecycle Manager(OLM) は、複数の異なる namespace に Operator を複数回インストールします。その 1 つの制約として、Operator の API バージョンは同じである必要があります。

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

2.4.5.10.1. 関連情報

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

2.4.5.11. Operator グループのトラブルシューティング

メンバーシップ

- 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。

2.4.6. Operator Lifecycle Manager メトリクス

2.4.6.1. 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2.7 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。

名前	説明
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。
install_plan_count	インストール計画の数。
subscription_count	サブスクリプションの数。
subscription_sync_total	サブスクリプション同期の単調 (monotonic) カウント。**channel**、**installed** CSV、およびサブスクリプション **name** ラベルが含まれます。

2.4.7. Operator Lifecycle Manager での Webhook の管理

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

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

2.4.7.1. 追加リソース

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

2.5. OPERATORHUB について

2.5.1. OperatorHub について

OperatorHub は OpenShift Container Platform の Web コンソールインターフェイスであり、これを使用してクラスター管理者は Operator を検出し、インストールします。1 回のクリックで、Operator をクラスター外のソースからプルし、クラスター上でインストールおよびサブスクライブして、エンジニアリングチームが Operator Lifecycle Manager (OLM) を使用してデプロイメント環境全体で製品をセルフサービスで管理される状態にすることができます。

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

カテゴリー	説明
Red Hat Operator	Red Hat によってパッケージ化され、出荷される Red Hat 製品。Red Hat によってサポートされます。
認定 Operator	大手独立系ソフトウェアベンダー (ISV) の製品。Red Hat は ISV とのパートナーシップにより、パッケージ化および出荷を行います。ISV によってサポートされます。
Red Hat Marketplace	Red Hat Marketplace から購入できる認定ソフトウェア。
コミュニティー Operator	operator-framework/community-operators GitHub リポジトリーで関連するエンティティーによってメンテナーンスされる、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 を追加していない場合、**カスタム** カテゴリーは Web コンソールの OperatorHub 上に表示されません。

OperatorHub の Operator は OLM で実行されるようにパッケージ化されます。これには、Operator のインストールおよびセキュアな実行に必要なすべての CRD、RBAC ルール、デプロイメント、およびコンテナーイメージが含まれるクラスターサービスバージョン (CSV) という YAML ファイルが含まれます。また、機能の詳細やサポートされる Kubernetes バージョンなどのユーザーに表示される情報も含まれます。

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

2.5.2. OperatorHub アーキテクチャー

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

2.5.2.1. OperatorHub カスタムリソース

Marketplace Operator は、OperatorHub で提供されるデフォルトの **CatalogSource** オブジェクトを管理する **cluster** という名前の **OperatorHub** カスタムリソース (CR) を管理します。このリソースを変更して、デフォルトのカタログを有効または無効にすることができます。これは、ネットワークが制限された環境で OpenShift Container Platform を設定する際に役立ちます。

OperatorHub カスタムリースの例

```
apiVersion: config.openshift.io/v1
```

```
kind: OperatorHub
metadata:
  name: cluster
spec:
  disableAllDefaultSources: true (1)
  sources: [ (2)
    {
      name: "community-operators",
      disabled: false
    }
  ]
```

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

(2) ソースごとに **disabled** パラメーター値を変更して、デフォルトのカタログを個別に無効にします。

2.5.3. 関連情報

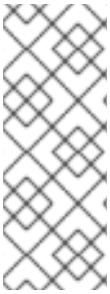
- カタログソース
- Operator SDK の使用を開始する
- ClusterServiceVersion (CSV) の生成
- OLM での Operator のインストールおよびアップグレードのワークフロー
- Red Hat Partner Connect
- Red Hat Marketplace

2.6. RED HAT が提供する OPERATOR カタログ

2.6.1. Operator カタログについて

Operator カタログは、Operator Lifecycle Manager (OLM) がクエリーを行い、Operator およびそれらの依存関係をクラスターで検出し、インストールできるメタデータのリポジトリーです。OLM は最新バージョンのカタログから Operator を常にインストールします。OpenShift Container Platform 4.6 の時点で、Red Hat が提供するカタログは **インデックスイメージ** を使用して提供されています。

Operator Bundle Format に基づくインデックスイメージは、カタログのコンテナー化されたスナップショットです。これは、Operator マニフェストコンテンツのセットへのポインターのデータベースが含まれるイミュータブルなアーティファクトです。カタログはインデックスイメージを参照し、クラスター上の OLM のコンテンツを調達できます。



注記

OpenShift Container Platform 4.6 以降では、Red Hat が提供するインデックスイメージは、以前のバージョンの OpenShift Container Platform 4 用に配布される非推奨の Package Manifest Format に基づいて、App Registry カタログイメージを置き換えます。OpenShift Container Platform 4.6 以降については、App Registry カタログイメージは Red Hat によって提供されませんが、Package Manifest Format に基づくカスタムカタログイメージは引き続きサポートされます。

カタログが更新されると、Operator の最新バージョンが変更され、それ以前のバージョンが削除または変更される可能性があります。さらに OLM がネットワークが制限された環境の OpenShift Container Platform クラスターで実行される場合、最新のコンテンツをプルするためにインターネットからカタログに直接アクセスすることはできません。

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

重要

カスタムカタログイメージを作成する場合、OpenShift Container Platform 4 の以前のバージョンでは、複数のリリースで非推奨となった **oc adm catalog build** コマンドの使用が必要でした。OpenShift Container Platform 4.6 以降の Red Hat が提供するインデックスイメージの可用性により、カタログビルダーは、**oc adm catalog build** コマンドが今後のリリースで削除される前に、**opm index** コマンドを使用してインデックスイメージを管理できるように切り換える必要があります。

関連情報

- カスタムカタログの管理
- ネットワークが制限された環境での Operator Lifecycle Manager の使用

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

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

カタログ	インデックスイメージ	説明
redhat-operators	**registry.redhat.io/redhat/redhat-operator-index:v4.6**	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。

カタログ	インデックスイメージ	説明
certified-operators	**registry.redhat.io/redhat/certified-operator-index:v4.6**	大手独立系ソフトウェアベンダー (ISV) の製品。Red Hat は ISV とのパートナーシップにより、パッケージ化および出荷を行います。ISV によってサポートされます。
redhat-marketplace	**registry.redhat.io/redhat/redhat-marketplace-index:v4.6**	Red Hat Marketplace から購入できる認定ソフトウェア。
community-operators	**registry.redhat.io/redhat/community-operator-index:v4.6**	operator-framework/community-operators GitHub リポジトリーで関連するエンティティーによってメンテナーンスされるソフトウェア。正式なサポートはありません。

2.7. CRD

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

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。

2.7.1.1. カスタムリソース定義

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

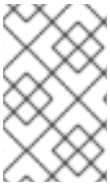
カスタムリソース定義 (CRD) オブジェクトは、クラスター内に新規の固有オブジェクト **kind** を定義し、Kubernetes API サーバーにそのライフサイクル全体を処理させます。

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

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

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

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



注記

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

2.7.1.2. カスタムリソース定義の作成

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

前提条件

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

手順

CRD を作成するには、以下を実行します。

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

CRD の YAML ファイルの例

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

1. **apiextensions.k8s.io/v1** API を使用します。
2. 定義の名前を指定します。これは **group** および **plural** フィールドの値を使用する **<plural-name>.<group>** 形式である必要があります。
3. API のグループ名を指定します。API グループは、論理的に関連付けられるオブジェクトのコレクションです。たとえば、**Job** または **ScheduledJob** などのすべてのバッチオブジェクトはバッチ API グループ (**batch.api.example.com** など) である可能性があります。組織の完全修飾ドメイン名 (FQDN) を使用することが奨励されます。

4. URL で使用されるバージョン名を指定します。それぞれの API グループは複数バージョンに存在させることができます (例: **v1alpha**、**v1beta**、 **v1**)。

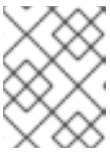
5. カスタムオブジェクトがクラスター (**Cluster**) の 1 つのプロジェクト (**Namespaced**) またはすべてのプロジェクトで利用可能であるかどうかを指定します。

6. URL で使用される複数形の名前を指定します。**plural** フィールドは API URL のリソースと同じになります。

7. CLI および表示用にエイリアスとして使用される単数形の名前を指定します。

8. 作成できるオブジェクトの種類を指定します。タイプは CamelCase にすることができます。

9. CLI でリソースに一致する短い文字列を指定します。



注記

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

2. CRD オブジェクトを作成します。

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

新規の RESTful API エンドポイントは以下のように作成されます。

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

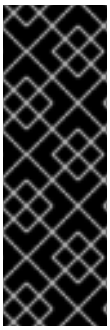
たとえば、サンプルファイルを使用すると、以下のエンドポイントが作成されます。

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

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

2.7.1.3. カスタムリソース定義のクラスターロールの作成

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



重要

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

前提条件

- CRD を作成します。

手順

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

カスタムロール定義の YAML ファイルサンプル

```
kind: ClusterRole
apiVersion: rbac.authorization.k8s.io/v1 (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

2. クラスターロールを作成します。

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

2.7.1.4. ファイルからのカスタムリソースの作成

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

前提条件

- CRD がクラスター管理者によってクラスターに追加されている。

手順

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

   CR の YAML ファイルサンプル

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

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

   2 CRD にタイプを指定します。

   3 オブジェクトの名前を指定します。

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

   5 オブジェクトのタイプに固有の条件を指定します。

2. ファイルの作成後に、オブジェクトを作成します。

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

2.7.1.5. カスタムリソースの検査

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

前提条件

- CR オブジェクトがアクセスできる namespace にあること。

手順

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

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

以下に例を示します。

```
$ oc get crontab
```

出力例

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

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

```
$ oc get crontabs
```

```
$ oc get crontab
```

```
$ oc get ct
```

2. CR の未加工の YAML データを確認することもできます。

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

以下に例を示します。

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

出力例

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

```
spec:
  cronSpec: '* * * * /5' 1
  image: my-awesome-cron-image 2
```

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

2.7.2. カスタムリソース定義からのリソースの管理

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

2.7.2.1. カスタムリソース定義

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

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

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

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

注記

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

2.7.2.2. ファイルからのカスタムリソースの作成

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

前提条件

- CRD がクラスター管理者によってクラスターに追加されている。

手順

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

   CR の YAML ファイルサンプル

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

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

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

2. ファイルの作成後に、オブジェクトを作成します。

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

2.7.2.3. カスタムリソースの検査

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

前提条件

- CR オブジェクトがアクセスできる namespace にあること。

手順

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

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

以下に例を示します。

```
$ oc get crontab
```

出力例

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

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

```
$ oc get crontabs
```

```
$ oc get crontab
```

```
$ oc get ct
```

2. CR の未加工の YAML データを確認することもできます。

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

以下に例を示します。

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

出力例

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

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

第3章 ユーザータスク

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

以下では、開発者を対象に、OpenShift Container Platform Web コンソールを使用して、インストールされた Operator からアプリケーションを作成する例を示します。

3.1.1. Operator を使用した etcd クラスターの作成

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

前提条件

- OpenShift Container Platform 4.6 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、etcd Operator をクリックして詳細情報および選択可能なアクションを表示します。
   Provided APIs に表示されているように、この Operator は 3 つの新規リソースタイプを利用可能にします。これには、**etcd クラスター** (**EtcdCluster** リソース) のタイプが含まれます。これらのオブジェクトは、**Deployment** または **ReplicaSet** などの組み込み済みのネイティブ Kubernetes オブジェクトと同様に機能しますが、これらには etcd を管理するための固有のロジックが含まれます。
4. 新規 etcd クラスターを作成します。
   a. **etcd Cluster** API ボックスで、**Create instance** をクリックします。
   b. 次の画面では、クラスターのサイズなど **EtcdCluster** オブジェクトのテンプレートを起動する最小条件への変更を加えることができます。ここでは **Create** をクリックして確定します。これにより、Operator がトリガーされ、Pod、サービス、および新規 etcd クラスターの他のコンポーネントが起動します。

5. **example** etcd クラスターをクリックしてから **Resources** タブをクリックして、プロジェクトに Operator によって自動的に作成され、設定された数多くのリソースが含まれることを確認します。
   Kubernetes サービスが作成され、プロジェクトの他の Pod からデータベースにアクセスできることを確認します。

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

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

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

3.2. NAMESPACE への OPERATOR のインストール

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

3.2.1. 前提条件

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

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

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。

更新チャネル

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

承認ストラテジー

自動 (Automatic) または手動 (Manual) のいずれかの更新を選択します。
インストールされた Operator について自動更新を選択する場合、Operator の新規バージョンが選択されたチャネルで利用可能になると、Operator Lifecycle Manager (OLM) は人の介入なしに、Operator の実行中のインスタンスを自動的にアップグレードします。

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

- OperatorHub について

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

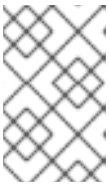
OpenShift Container Platform Web コンソールを使用して OperatorHub から Operator をインストールし、これをサブスクライブできます。

前提条件

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

手順

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



注記

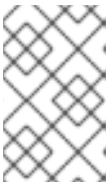
コミュニティー Operator を選択すると、Red Hat がコミュニティー Operator を認定していないことを警告します。続行する前に警告を確認する必要があります。

4. Operator についての情報を確認してから、**Install** をクリックします。
5. **Install Operator** ページで以下を行います。
   a. Operator をインストールする特定の単一 namespace を選択します。Operator は監視のみを実行し、この単一 namespace で使用されるように利用可能になります。
   b. **Update Channel** を選択します (複数を選択できる場合)。
   c. 前述のように、**自動 (Automatic)** または **手動 (Manual)** の承認ストラテジーを選択します。
6. **Install** をクリックし、Operator をこの OpenShift Container Platform クラスターの選択した namespace で利用可能にします。
   a. **手動** の承認ストラテジーを選択している場合、サブスクリプションのアップグレードステータスは、そのインストール計画を確認し、承認するまで **Upgrading** のままになります。

Install Plan ページでの承認後に、サブスクリプションのアップグレードステータスは **Up to date** に移行します。

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

7. サブスクリプションのアップグレードステータスが **Up to date** になった後に、**Operators → Installed Operators** を選択し、インストールされた Operator のクラスターサービスバージョン (CSV) が表示されることを確認します。その **Status** は最終的に関連する namespace で **InstallSucceeded** に解決するはずです。



注記

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

上記通りにならない場合、以下を実行します。

a. さらにトラブルシューティングを行うために問題を報告している **Workloads → Pods** ページで、**openshift-operators** プロジェクト (または **A specific namespace...** インストールモードが選択されている場合は他の関連の namespace) の Pod のログを確認します。

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

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

前提条件

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

手順

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

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

出力例

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

```
jaeger                 Community Operators   91m
kubefed                Community Operators   91m
...
```

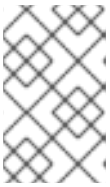
必要な Operator のカタログをメモします。

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

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

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

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



注記

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

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

OperatorGroup オブジェクトのサンプル

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

b. **OperatorGroup** オブジェクトを作成します。

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

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

Subscription オブジェクトの例

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

```
  name: <subscription_name>
  namespace: openshift-operators (1)
spec:
  channel: <channel_name> (2)
  name: <operator_name> (3)
  source: redhat-operators (4)
  sourceNamespace: openshift-marketplace (5)
  config:
    env: (6)
    - name: ARGS
      value: "-v=10"
    envFrom: (7)
    - secretRef:
        name: license-secret
    volumes: (8)
    - name: <volume_name>
      configMap:
        name: <configmap_name>
    volumeMounts: (9)
    - mountPath: <directory_name>
      name: <volume_name>
    tolerations: (10)
    - operator: "Exists"
    resources: (11)
      requests:
        memory: "64Mi"
        cpu: "250m"
      limits:
        memory: "128Mi"
        cpu: "500m"
    nodeSelector: (12)
      foo: bar
```

(1) **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ネルの名前。

(3) サブスクライブする Operator の名前。

(4) Operator を提供するカタログソースの名前。

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

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

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

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

9 **volumeMounts** パラメーターは、OLM によって作成される Pod のすべてのコンテナーに存在する必要があるボリュームマウントの一覧を定義します。**volumeMount** が存在しな

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

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

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

5. **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
- チャネル名

3.2.5. Operator の特定バージョンのインストール

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

前提条件

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

手順

1. **startingCSV** フィールドを設定し、特定バージョンの Operator に namespace をサブスクライブする **Subscription** オブジェクト YAML ファイルを作成します。**installPlanApproval** フィールドを **Manual** に設定し、Operator の新しいバージョンがカタログに存在する場合に Operator が自動的にアップグレードされないようにします。
たとえば、以下の **sub.yaml** ファイルを使用して、バージョン 3.4.0 に固有の Red Hat Quay Operator をインストールすることができます。

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

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

```
channel: quay-v3.4
installPlanApproval: Manual (1)
name: quay-operator
source: redhat-operators
sourceNamespace: openshift-marketplace
startingCSV: quay-operator.v3.4.0 (2)
```

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

(2) Operator CSV の特定バージョンを設定します。

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

第4章 管理者タスク

4.1. OPERATOR のクラスターへの追加

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

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

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。

更新チャネル

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

承認ストラテジー

自動 (Automatic) または手動 (Manual) のいずれかの更新を選択します。
インストールされた Operator について自動更新を選択する場合、Operator の新規バージョンが選択されたチャネルで利用可能になると、Operator Lifecycle Manager (OLM) は人の介入なしに、Operator の実行中のインスタンスを自動的にアップグレードします。

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

- OperatorHub について

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

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

前提条件

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

手順

1. Web コンソールで、**Operators → OperatorHub** ページに移動します。

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

3. Operator を選択して、追加情報を表示します。

注記

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

4. Operator についての情報を確認してから、**Install** をクリックします。

5. **Install Operator** ページで以下を行います。

   a. 以下のいずれかを選択します。

      - **All namespaces on the cluster (default)**は、デフォルトの **openshift-operators** namespace で Operator をインストールし、クラスターのすべての namespace を監視し、Operator をこれらの namespace に対して利用可能にします。このオプションは常に選択可能です。

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

   b. Operator をインストールする特定の単一 namespace を選択します。Operator は監視のみを実行し、この単一 namespace で使用されるように利用可能になります。

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

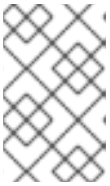
   d. 前述のように、**自動 (Automatic)** または **手動 (Manual)** の承認ストラテジーを選択します。

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

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

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

7. サブスクリプションのアップグレードステータスが **Up to date** になった後に、**Operators → Installed Operators** を選択し、インストールされた Operator のクラスターサービスバージョン (CSV) が表示されることを確認します。その **Status** は最終的に関連する namespace で **InstallSucceeded** に解決するはずです。



注記

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

上記通りにならない場合、以下を実行します。

a. さらにトラブルシューティングを行うために問題を報告している **Workloads → Pods** ページで、**openshift-operators** プロジェクト (または **A specific namespace...** インストールモードが選択されている場合は他の関連の namespace) の Pod のログを確認します。

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

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

前提条件

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

手順

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

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

出力例

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

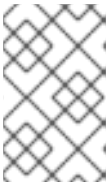
必要な Operator のカタログをメモします。

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

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

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

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

   注記

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

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

      OperatorGroup オブジェクトのサンプル

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

   b. **OperatorGroup** オブジェクトを作成します。

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

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

   Subscription オブジェクトの例

   ```
   apiVersion: operators.coreos.com/v1alpha1
   kind: Subscription
   metadata:
     name: <subscription_name>
     namespace: openshift-operators 1
   spec:
     channel: <channel_name> 2
     name: <operator_name> 3
     source: redhat-operators 4
     sourceNamespace: openshift-marketplace 5
     config:
       env: 6
       - name: ARGS
         value: "-v=10"
   ```

```
  envFrom: 7
  - secretRef:
      name: license-secret
  volumes: 8
  - name: <volume_name>
    configMap:
      name: <configmap_name>
  volumeMounts: 9
  - mountPath: <directory_name>
    name: <volume_name>
  tolerations: 10
  - operator: "Exists"
  resources: 11
    requests:
      memory: "64Mi"
      cpu: "250m"
    limits:
      memory: "128Mi"
      cpu: "500m"
  nodeSelector: 12
    foo: bar
```

1. **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ネルの名前。
3. サブスクライブする Operator の名前。
4. Operator を提供するカタログソースの名前。
5. カタログソースの namespace。デフォルトの OperatorHub カタログソースには **openshift-marketplace** を使用します。
6. **env** パラメーターは、OLM によって作成される Pod のすべてのコンテナーに存在する必要がある環境変数の一覧を定義します。
7. **envFrom** パラメーターは、コンテナーの環境変数に反映するためのソースの一覧を定義します。
8. **volumes** パラメーターは、OLM によって作成される Pod に存在する必要があるボリュームの一覧を定義します。
9. **volumeMounts** パラメーターは、OLM によって作成される Pod のすべてのコンテナーに存在する必要があるボリュームマウントの一覧を定義します。**volumeMount** が存在しない **ボリューム** を参照する場合、OLM は Operator のデプロイに失敗します。
10. **tolerations** パラメーターは、OLM によって作成される Pod の容認の一覧を定義します。
11. **resources** パラメーターは、OLM によって作成される Pod のすべてのコンテナーのリソース制約を定義します。
12. **nodeSelector** パラメーターは、OLM によって作成される Pod の **ノードセレクター** を定義します。

5. **Subscription** オブジェクトを作成します。

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

この時点で、OLM は選択した Operator を認識します。Operator のクラスターサービスバージョン (CSV) はターゲット namespace に表示され、Operator で指定される API は作成用に利用可能になります。

関連情報

- Operator グループについて

4.1.4. Operator の特定バージョンのインストール

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

前提条件

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

手順

1. **startingCSV** フィールドを設定し、特定バージョンの Operator に namespace をサブスクライブする **Subscription** オブジェクト YAML ファイルを作成します。**installPlanApproval** フィールドを **Manual** に設定し、Operator の新しいバージョンがカタログに存在する場合に Operator が自動的にアップグレードされないようにします。
たとえば、以下の **sub.yaml** ファイルを使用して、バージョン 3.4.0 に固有の Red Hat Quay Operator をインストールすることができます。

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

```
apiVersion: operators.coreos.com/v1alpha1
kind: Subscription
metadata:
  name: quay-operator
  namespace: quay
spec:
  channel: quay-v3.4
  installPlanApproval: Manual (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、承認ストラテジーを **Manual** に設定します。これにより、新しいバージョンへの自動アップグレードが阻止され、最初の CSV のインストールが完了する前に手動での承認が必要となります。

(2) Operator CSV の特定バージョンを設定します。

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

4.2. 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。

4.2.1. Operator の更新チャネルの変更

インストールされた Operator のサブスクリプションは、Operator の更新を追跡し、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、サブスクリプションで更新チャネルを変更できます。

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

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ストラテジーが **Automatic** に設定されている場合、アップグレードプロセスは、選択したチャネルで新規 Operator バージョンが利用可能になるとすぐに開始します。承認ストラテジーが **Manual** に設定されている場合、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ネルの名前をクリックします。
5. 変更する新しい更新チャネルをクリックし、**Save** をクリックします。

6. **Automatic** 承認ストラテジーのあるサブスクリプションの場合、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サブスクリプションの場合、**Subscription** タブからアップグレードを手動で承認できます。

4.2.2. 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。

4.3. クラスターからの OPERATOR の削除

以下では、OpenShift Container Platform クラスターで Operator Lifecycle Manager (OLM) を使用して、以前にインストールされた Operator をアップグレードする方法を説明します。

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

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

前提条件

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

手順

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

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

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

   このアクションにより、Operator および Operator のデプロイメントおよび Pod が削除されます (ある場合)。CRD および CR を含む Operator によって管理される Operand およびリソースは削除されません。Web コンソールは、一部の Operator のダッシュボードおよびナビゲーションアイテムを有効にします。Operator のアンインストール後にこれらを削除するには、Operator CRD を手動で削除する必要があります。

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

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

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

前提条件

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

手順

1. サブスクライブされた 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
   ```

2. サブスクリプション (例: **jaeger**) を削除します。

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

   出力例

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

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

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

   出力例

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

4.3.3. 障害のあるサブスクリプションの更新

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

出力例

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

出力例

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

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

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

前提条件

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

手順

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

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

   出力例

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

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

2. サブスクリプションを削除します。

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

3. クラスターサービスバージョンを削除します。

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

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

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

出力例

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

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

5. ジョブを削除します。

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

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

6. 設定マップを削除します。

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

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

検証

- Operator が正常に再インストールされていることを確認します。

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

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

グローバルプロキシーが OpenShift 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ーで管理する Operator を自動的に設定します。ただし、インストールされた Operator をグローバルプロキシーを上書きするか、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ーの設定
- カスタム PKI の設定 (カスタム CA 証明書)

4.4.1. Operator のプロキシー設定の上書き

クラスター全体の egress プロキシーが設定されている場合、Operator Lifecycle Manager (OLM) を使用して実行する Operator は、デプロイメントでクラスター全体のプロキシー設定を継承します。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ー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 でのプロキシー設定の環境変数の設定を処理する必要があります。

前提条件

- **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。

   プロキシー設定の上書きのある Subscription オブジェクト

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

注記

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

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

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

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

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

出力例

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

4.4.2. カスタム CA 証明書の挿入

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 プロキシー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 1
  labels:
    config.openshift.io/inject-trusted-cabundle: "true" 2
```

1 設定マップの名前。

2 Cluster Network Operator に対してマージされたバンドルを挿入するように要求します。

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

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

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

1 **config** セクションがない場合に、これを追加します。

2 Operator が所有する Pod に一致するラベルを指定します。

3 **trusted-ca** ボリュームを作成します。

4 **ca-bundle.crt** は設定マップキーとして必要になります。

5 **tls-ca-bundle.pem** は設定マップパスとして必要になります。

6 **trusted-ca** ボリュームマウントを作成します。

4.5. OPERATOR ステータスの表示

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

4.5.1. Operator サブスクリプションの状態のタイプ

サブスクリプションは状態についての以下のタイプを報告します。

表4.1 サブスクリプションの状態のタイプ

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

注記

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

関連情報

- 障害のあるサブスクリプションの更新

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

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

前提条件

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

手順

1. Operator サブスクリプションを一覧表示します。

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

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

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

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

出力例

```
Conditions:
  Last Transition Time:  2019-07-29T13:42:57Z
  Message:               all available catalogsources are healthy
  Reason:                AllCatalogSourcesHealthy
  Status:                False
  Type:                  CatalogSourcesUnhealthy
```

注記

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

4.5.3. CLI を使った Operator カタログソースのステータス表示

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

前提条件

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

手順

1. namespace のカタログソースを一覧表示します。例えば、クラスター全体のカタログソースに使用されている **openshift-marketplace**namespace を確認することができます。

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

出力例

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

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

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

出力例

```
Name:         example-catalog
Namespace:    openshift-marketplace
...
Status:
  Connection State:
    Address:              example-catalog.openshift-marketplace.svc:50051
    Last Connect:         2021-09-09T17:07:35Z
    Last Observed State:  TRANSIENT_FAILURE
  Registry Service:
    Created At:         2021-09-09T17:05:45Z
    Port:               50051
    Protocol:           grpc
    Service Name:       example-catalog
    Service Namespace:  openshift-marketplace
```

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

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

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

出力例

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

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

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

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

出力例

```
Name:         example-catalog-bwt8z
Namespace:    openshift-marketplace
Priority:     0
Node:         ci-ln-jyryyg2-f76d1-ggdbq-worker-b-vsxjd/10.0.128.2
...
Events:
  Type     Reason          Age              From               Message
  ----     ------          ----             ----               -------
  Normal   Scheduled       48s              default-scheduler  Successfully assigned openshift-
```

```
marketplace/example-catalog-bwt8z to ci-ln-jyryyf2-f76d1-fgdbq-worker-b-vsxjd
  Normal   AddedInterface  47s                multus            Add eth0 [10.131.0.40/23] from
openshift-sdn
  Normal   BackOff         20s (x2 over 46s)  kubelet           Back-off pulling image
"quay.io/example-org/example-catalog:v1"
  Warning  Failed          20s (x2 over 46s)  kubelet           Error: ImagePullBackOff
  Normal   Pulling         8s (x3 over 47s)   kubelet           Pulling image "quay.io/example-
org/example-catalog:v1"
  Warning  Failed          8s (x3 over 47s)   kubelet           Failed to pull image
"quay.io/example-org/example-catalog:v1": rpc error: code = Unknown desc = reading
manifest v1 in quay.io/example-org/example-catalog: unauthorized: access to the requested
resource is not authorized
  Warning  Failed          8s (x3 over 47s)   kubelet           Error: ErrImagePull
```

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

関連情報

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

4.6. 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。

4.6.1. 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。

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

4.6.1.2. 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。

4.6.2. 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 を制限する必要の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 ですべてのことを実行できるパーミッションを付与します。実稼働環境では、より粒度の細かいパーミッションセットを作成する必要があります。

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

```
  apiGroup: rbac.authorization.k8s.io
  kind: Role
  name: scoped
subjects:
- kind: ServiceAccount
  name: scoped
  namespace: scoped
EOF
```

3. 指定された namespace に **OperatorGroup** オブジェクトを作成します。この Operator グループは指定された namespace をターゲットにし、そのテナンシーがこれに制限されるようにします。
さらに、Operator グループはユーザーがサービスアカウントを指定できるようにします。直前の手順で作成したサービスアカウントを指定します。

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

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

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

```
$ cat <<EOF | oc create -f -
apiVersion: operators.coreos.com/v1alpha1
kind: Subscription
metadata:
  name: etcd
  namespace: scoped
spec:
  channel: singlenamespace-alpha
  name: etcd
  source: <catalog_source_name> 1
  sourceNamespace: <catalog_source_namespace> 2
EOF
```

1. 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。

2. カタログソースが作成された namespace を指定します。

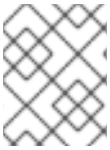
この Operator グループに関連付けられる Operator は、指定されたサービスアカウントに付与されるパーミッションに制限されます。Operator がサービスアカウントの範囲外のパーミッションを要求する場合、インストールは関連するエラーを出して失敗します。

4.6.2.1. 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。

4.6.3. 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。

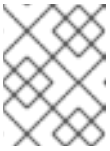
```
apiVersion: operators.coreos.com/v1
kind: Subscription
metadata:
  name: etcd
  namespace: scoped
status:
  installPlanRef:
    apiVersion: operators.coreos.com/v1
    kind: InstallPlan
    name: install-4plp8
    namespace: scoped
    resourceVersion: "117359"
    uid: 2c1df80e-afea-11e9-bce3-5254009c9c23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

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。

4.7. カスタムカタログの管理

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

関連情報

- Red Hat が提供する Operator カタログ

4.7.1. Bundle Format を使用したカスタムカタログ

4.7.1.1. 前提条件

- **opm** CLI をインストールします。

4.7.1.2. インデックスイメージの作成

opm CLI を使用してインデックスイメージを作成できます。

前提条件

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

重要

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

手順

1. 新しいインデックスを開始します。

```
$ opm index add \
    --bundles <registry>/<namespace>/<bundle_image_name>:<tag> \ 1
    --tag <registry>/<namespace>/<index_image_name>:<tag> \ 2
    [--binary-image <registry_base_image>] 3
```

1. インデックスに追加するバンドルイメージのコンマ区切りの一覧。
2. インデックスイメージで使用するイメージタグ。
3. オプション: カタログを提供するために使用する代替レジストリーベースイメージ。

2. インデックスイメージをレジストリーにプッシュします。
   a. 必要な場合は、ターゲットレジストリーで認証します。

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

   b. インデックスイメージをプッシュします。

```
$ podman push <registry>/<namespace>/test-catalog:latest
```

4.7.1.3. インデックスイメージからのカタログの作成

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

前提条件

- レジストリーにビルドされ、プッシュされるインデックスイメージ。

手順

1. インデックスイメージを参照する **CatalogSource** オブジェクトを作成します。
   a. 仕様を以下のように変更し、これを **catalogSource.yaml** ファイルとして保存します。

```
apiVersion: operators.coreos.com/v1alpha1
kind: CatalogSource
metadata:
  name: my-operator-catalog
  namespace: openshift-marketplace 1
spec:
  sourceType: grpc
  image: <registry>:<port>/<namespace>/redhat-operator-index:v4.6 2
  displayName: My Operator Catalog
  publisher: <publisher_name> 3
  updateStrategy:
    registryPoll: 4
      interval: 30m
```

1. カタログソースを全 namespace のユーザーがグローバルに利用できるようにする場合は、**openshift-marketplace** namespace を指定します。それ以外の場合は、そのカタログの別の namespace を対象とし、その namespace のみが利用できるように指定できます。
2. インデックスイメージを指定します。
3. カタログを公開する名前または組織名を指定します。
4. カタログソースは新規バージョンの有無を自動的にチェックし、最新の状態を維持します。

b. このファイルを使用して **CatalogSource** オブジェクトを作成します。

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

2. 以下のリソースが正常に作成されていることを確認します。

a. Pod を確認します。

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

出力例

```
NAME                                  READY   STATUS    RESTARTS  AGE
my-operator-catalog-6njx6             1/1     Running   0         28s
marketplace-operator-d9f549946-96sgr  1/1     Running   0         26h
```

b. カタログソースを確認します。

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

出力例

```
NAME                  DISPLAY               TYPE PUBLISHER  AGE
my-operator-catalog   My Operator Catalog   grpc           5s
```

c. パッケージマニフェストを確認します。

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

出力例

```
NAME                     CATALOG               AGE
jaeger-product           My Operator Catalog   93s
```

OpenShift Container Platform Web コンソールで、**OperatorHub** ページから Operator をインストールできるようになりました。

4.7.1.4. インデックスイメージの更新

カスタムインデックスイメージを参照するカタログソースを使用するように OperatorHub を設定した後に、クラスター管理者はバンドルイメージをインデックスイメージに追加して、クラスターで利用可能な Operator を最新の状態に維持することができます。

opm index add コマンドを使用して既存インデックスイメージを更新できます。

前提条件

- **opm** version 1.12.3+
- **podman** version 1.9.3+
- レジストリーにビルドされ、プッシュされるインデックスイメージ。
- インデックスイメージを参照する既存のカタログソース。

手順

1. バンドルイメージを追加して、既存のインデックスを更新します。

```
$ opm index add \
   --bundles <registry>/<namespace>/<new_bundle_image>@sha256:<digest> \ 1
   --from-index <registry>/<namespace>/<existing_index_image>:<existing_tag> \ 2
   --tag <registry>/<namespace>/<existing_index_image>:<updated_tag> \ 3
   --pull-tool podman 4
```

1. **--bundles**フラグは、インデックスに追加する他のバンドルイメージのコンマ区切りリストを指定します。
2. **--from-index**フラグは、以前にプッシュされたインデックスを指定します。
3. **--tag**フラグは、更新されたインデックスイメージに適用するイメージタグを指定します。
4. **--pull-tool**フラグは、コンテナーイメージのプルに使用されるツールを指定します。

ここでは、以下のようになります。

<registry>
: **quay.io**や**mirror.example.com**などのレジストリーのホスト名を指定します。

<namespace>
: **ocs-dev**や**abc**など、レジストリーの namespace を指定します。

<new_bundle_image>
: **ocs-operator**など、レジストリーに追加する新しいバンドルイメージを指定します。

<digest>
: **c7f11097a628f092d8bad148406aa0e0951094a03445fd4bc0775431ef683a41**などのバンドルイメージの SHA イメージ ID またはダイジェストを指定します。

<existing_index_image>
: **abc-redhat-operator-index**など、以前にプッシュされたイメージを指定します。

<existing_tag>
: **4.6** など、以前にプッシュされたイメージタグを指定します。

<updated_tag>

4.6.1 など、更新されたインデックスイメージに適用するイメージタグを指定します。

コマンドの例

```
$ opm index add \
    --bundles quay.io/ocs-dev/ocs-
operator@sha256:c7f11097a628f092d8bad148406aa0e0951094a03445fd4bc0775431ef683a
41 \
    --from-index mirror.example.com/abc/abc-redhat-operator-index:4.6 \
    --tag mirror.example.com/abc/abc-redhat-operator-index:4.6.1 \
    --pull-tool podman
```

2. 更新されたインデックスイメージをプッシュします。

```
$ podman push <registry>/<namespace>/<existing_index_image>:<updated_tag>
```

3. Operator Lifecycle Manager (OLM) がカタログソースで参照されるインデックスイメージを一定間隔で自動的にポーリングした後に、新規パッケージが正常に追加されたことを確認します。

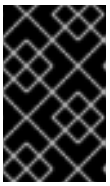
```
$ oc get packagemanifests -n openshift-marketplace
```

4.7.1.5. インデックスイメージのプルーニング

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

前提条件

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



重要

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

手順

1. ターゲットレジストリーで認証します。

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

2. プルーニングされたインデックスに追加するパッケージの一覧を判別します。

    a. コンテナーでプルーニングするソースインデックスイメージを実行します。以下に例を示します。

```
$ podman run -p50051:50051 \
    -it registry.redhat.io/redhat/redhat-operator-index:v4.6
```

出力例

```
Trying to pull registry.redhat.io/redhat/redhat-operator-index:v4.6...
Getting image source signatures
Copying blob ae8a0c23f5b1 done
...
INFO[0000] serving registry                 database=/database/index.db port=50051
```

b. 別のターミナルセッションで、**grpcurl** コマンドを使用して、インデックスが提供するパッケージの一覧を取得します。

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

c. **packages.out** ファイルを検査し、プルーニングされたインデックスに保持したいパッケージ名をこの一覧から特定します。以下に例を示します。

パッケージ一覧のスニペットの例

```
...
{
  "name": "advanced-cluster-management"
}
...
{
  "name": "jaeger-product"
}
...
{
{
  "name": "quay-operator"
}
...
```

d. **podman run** コマンドを実行したターミナルセッションで、**Ctrl** と **C** を押してコンテナープロセスを停止します。

3. 以下のコマンドを実行して、指定したパッケージ以外のすべてのパッケージのソースインデックスをプルーニングします。

```
$ opm index prune \
    -f registry.redhat.io/redhat/redhat-operator-index:v4.6 \ 1
    -p advanced-cluster-management,jaeger-product,quay-operator \ 2
    [-i registry.redhat.io/openshift4/ose-operator-registry:v4.6] \ 3
    -t <target_registry>:<port>/<namespace>/redhat-operator-index:v4.6 4
```

1 プルーニングするインデックス。

2 保持するパッケージのコンマ区切りリスト。

3 IBM Power Systems および IBM Z イメージのみに必要です: ターゲット OpenShift Container Platform クラスターのメジャーバージョンおよびマイナーバージョンに一致す

るタグを使用する Operator レジストリーのベースイメージです。

4 ビルドされる新規インデックスイメージのカスタムタグ。

4. 以下のコマンドを実行して、新規インデックスイメージをターゲットレジストリーにプッシュします。

```
$ podman push <target_registry>:<port>/<namespace>/redhat-operator-index:v4.6
```

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

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

4.7.2.1. Package Manifest Format 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ー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 1.9.3+
- 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>
```

2. 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
```

3. 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:v4.6 \ 2
    --filter-by-os="linux/amd64" \ 3
    --to=<registry_host_name>:<port>/olm/redhat-operators:v1 \ 4
    [-a ${REG_CREDS}] \ 5
    [--insecure] \ 6
    [--auth-token "${AUTH_TOKEN}"] 7
```

1. App Registry インスタンスからのプルに使用する組織 (namespace)。
2. 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 Operator レジストリーの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設定します。使用できる値は、**linux/amd64**、**linux/ppc64le**、および **linux/s390x** です。
4. 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。
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

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、それらを無視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

4.7.2.2. Package Manifest Format カタログイメージのミラーリング

クラスター管理者は Package Manifest Format に基づいてカスタム Operator カタログイメージをレジストリーにミラーリングし、カタログソースを使用してコンテンツを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 Package Manifest Format に基づくカスタム Operator カタログイメージ
- **oc** version 4.3.5+
- **podman** version 1.9.3+
- 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

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。

- 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. **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> \ 2
  [-a ${REG_CREDS}] \ 3
  [--insecure] \ 4
  [--index-filter-by-os='<platform>/<arch>'] \ 5
  [--manifests-only] 6
```

1. Operator カタログイメージを指定します。
2. ターゲットレジストリーの完全修飾ドメイン名 (FQDN) を指定します。
3. オプション: 必要な場合は、レジストリー認証情報ファイルの場所を指定します。
4. オプション: ターゲットレジストリーの信頼を設定しない場合は、**--insecure** フラグを追加します。
5. オプション: 複数のバリアントが利用可能な場合に、選択するカタログイメージのプラットフォームおよびアーキテクチャーを指定します。イメージは **'<platform>/<arch>[/<variant>]'** として渡されます。これは、カタログイメージで参照されるイメージには適用されません。使用できる値は、**linux/amd64**、**linux/ppc64le**、および **linux/s390x** です。
6. オプション: ミラーリングに必要なマニフェストのみを生成し、実際にはイメージコンテンツをレジストリーにミラーリングしません。

出力例

```
using database path mapping: /:/tmp/190214037
wrote database to /tmp/190214037
using database at: /tmp/190214037/bundles.db 1
```

```
...
```

1 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、**manifests-<index_image_name>-<random_number>/** 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。

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

注記

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

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

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@sha256:6b4db07f6e6c962fc96473d86c44532c93b146bbefe311d0c348117bf759c506
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:aa4a8b2a00836d0e28aa6497ad90a3c116f135f382d8211e3c55f34fb36dfe61=<registry_host_name>:<port>/openshift4-ose-logging-kibana5:a767c8f0
registry.redhat.io/openshift4/ose-oauth-
proxy@sha256:6b4db07f6e6c962fc96473d86c44532c93b146bbefe311d0c348117bf759c506=<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 ./manifests-redhat-operators-<random_number>/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** オブジェクトを作成します。

```
$ oc create -f ./manifests-redhat-operators-
<random_number>/imageContentSourcePolicy.yaml
```

ミラーリングされたコンテンツを参照するように **CatalogSource** オブジェクトを作成できるようになりました。

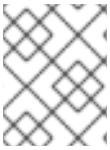
関連情報

- 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

4.7.2.3. Package Manifest Format 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。

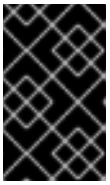


注記

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 1.9.3+
- 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



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。

- 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>
```

2. 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
```

3. 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 \ 1
    --from=registry.redhat.io/openshift4/ose-operator-registry:v4.6 \ 2
    --filter-by-os="linux/amd64" \ 3
    --to=<registry_host_name>:<port>/olm/redhat-operators:v2 \ 4
    [-a ${REG_CREDS}] \ 5
    [--insecure] \ 6
    [--auth-token "${AUTH_TOKEN}"] 7
```

1. App Registry インスタンスからのプルに使用する組織 (namespace)。
2. 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 Operator レジストリーの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設定します。使用できる値は、**linux/amd64**、**linux/ppc64le**、および **linux/s390x** です。
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
```

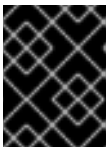
4. カタログのコンテンツをターゲットレジストリーに対してミラーリングします。以下の **oc adm catalog mirror** コマンドは、カスタム Operator カタログイメージのコンテンツを抽出し、ミラーリングに必要なマニフェストを生成し、イメージをレジストリーにミラーリングします。

```
$ oc adm catalog mirror \
    <registry_host_name>:<port>/olm/redhat-operators:v2 \ 1
    <registry_host_name>:<port> \ 2
    [-a ${REG_CREDS}] \ 3
    [--insecure] \ 4
    [--index-filter-by-os='<platform>/<arch>'] 5
```

1. 新規の Operator カタログイメージを指定します。
2. ターゲットレジストリーの完全修飾ドメイン名 (FQDN) を指定します。
3. オプション: 必要な場合は、レジストリー認証情報ファイルの場所を指定します。
4. オプション: ターゲットレジストリーの信頼を設定しない場合は、**--insecure** フラグを追加します。
5. オプション: 複数のバリアントが利用可能な場合に、選択するカタログイメージのプラットフォームおよびアーキテクチャーを指定します。イメージは **'<platform>/<arch>[/<variant>]'** として渡されます。これは、カタログイメージで参照されるイメージには適用されません。使用できる値は、**linux/amd64**、**linux/ppc64le**、および **linux/s390x** です。

5. 新たに生成されたマニフェストを適用します。

```
$ oc replace -f ./manifests-redhat-operators-<random_number>
```



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。

6. 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

4.7.2.4. Package Manifest Format 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、これをコンテナーとして実行し、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、カタログソースでイメージを参照して Operator Lifecycle Manager (OLM) で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 Package Manifest Format カタログイメージ
- **podman** version 1.9.3+
- **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

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。

- **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-ossm","channelName":"stable"}' localhost:50051
api.Registry/GetBundleForChannel
```

出力例

```
{
  "csvName": "kiali-operator.v1.0.7",
  "packageName": "kiali-ossm",
  "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"
```

4.7.3. デフォルトの OperatorHub ソースの無効化

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

手順

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

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

ヒント

または、Web コンソールを使用してカタログソースを管理できます。**Administration → Cluster Settings → Global Configuration → OperatorHub** ページから、**Sources** タブをクリックして、個別のソースを作成し、削除し、無効にし、有効にすることができます。

4.7.4. カスタムカタログの削除

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

手順

1. Web コンソールの **Administrator** パースペクティブで、**Administration → Cluster Settings** に移動します。
2. **Global Configuration** タブをクリックしてから、**OperatorHub** をクリックします。
3. **Sources** タブをクリックします。
4. 削除するカタログの **Options** メニュー を選択し、**Delete CatalogSource** をクリックします。

4.8. 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) はリモートレジストリーでホストされる Red Hat が提供する OperatorHub ソースにアクセスできません。それらのリモートソースには完全なインターネット接続が必要であるためです。

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

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

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

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

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

重要

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

- 関連するイメージ、または Operator がそれらの機能を実行するために必要となる可能性のある他のコンテナーイメージを **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。

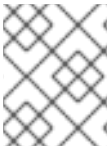
https://access.redhat.com/articles/4740011

関連情報

- Red Hat が提供する Operator カタログ
- ネットワークが制限された環境についての Operator の有効化

4.8.1. 前提条件

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



注記

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

4.8.2. デフォルトの OperatorHub ソースの無効化

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

手順

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

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

ヒント

または、Web コンソールを使用してカタログソースを管理できます。**Administration → Cluster Settings → Global Configuration → OperatorHub** ページから、**Sources** タブをクリックして、個別のソースを作成し、削除し、無効にし、有効にすることができます。

4.8.3. インデックスイメージのプルーニング

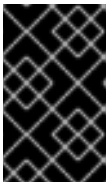
Operator Bundle Format に基づくインデックスイメージは、Operator カタログのコンテナー化されたスナップショットです。パッケージの指定された一覧以外のすべてのインデックスをプルーニングできます。これにより、必要な Operator のみが含まれるソースインデックスのコピーを作成できます。

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

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

前提条件

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



重要

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

手順

1. **registry.redhat.io** で認証します。

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

2. ターゲットレジストリーで認証します。

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

3. プルーニングされたインデックスに追加するパッケージの一覧を判別します。

   a. コンテナーでプルーニングするソースインデックスイメージを実行します。以下に例を示します。

```
$ podman run -p50051:50051 \
    -it registry.redhat.io/redhat/redhat-operator-index:v4.6
```

出力例

```
Trying to pull registry.redhat.io/redhat/redhat-operator-index:v4.6...
Getting image source signatures
Copying blob ae8a0c23f5b1 done
...
INFO[0000] serving registry                 database=/database/index.db port=50051
```

b. 別のターミナルセッションで、**grpcurl** コマンドを使用して、インデックスが提供するパッケージの一覧を取得します。

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

c. **packages.out** ファイルを検査し、プルーニングされたインデックスに保持したいパッケージ名をこの一覧から特定します。以下に例を示します。

パッケージ一覧のスニペットの例

```
...
{
  "name": "advanced-cluster-management"
}
...
{
  "name": "jaeger-product"
}
...
{
{
  "name": "quay-operator"
}
...
```

d. **podman run** コマンドを実行したターミナルセッションで、**Ctrl** と **C** を押してコンテナープロセスを停止します。

4. 以下のコマンドを実行して、指定したパッケージ以外のすべてのパッケージのソースインデックスをプルーニングします。

```
$ opm index prune \
    -f registry.redhat.io/redhat/redhat-operator-index:v4.6 \ 1
    -p advanced-cluster-management,jaeger-product,quay-operator \ 2
    [-i registry.redhat.io/openshift4/ose-operator-registry:v4.6] \ 3
    -t <target_registry>:<port>/<namespace>/redhat-operator-index:v4.6 4
```

1 プルーニングするインデックス。

2 保持するパッケージのコンマ区切りリスト。

3 IBM Power Systems および IBM Z イメージのみに必要です: ターゲット OpenShift Container Platform クラスターのメジャーバージョンおよびマイナーバージョンに一致す

るタグを使用する Operator レジストリーのベースイメージです。

4 ビルドされる新規インデックスイメージのカスタムタグ。

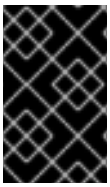
5. 以下のコマンドを実行して、新規インデックスイメージをターゲットレジストリーにプッシュします。

```
$ podman push <target_registry>:<port>/<namespace>/redhat-operator-index:v4.6
```

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

4.8.4. Operator カタログのミラーリング

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



重要

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

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

前提条件

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

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

手順

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

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

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

- **オプション A: ミラーレジストリーがネットワークアクセスが無制限のワークステーションと同じネットワーク上にある** 場合、ワークステーションで以下のアクションを実行します。

a. ミラーレジストリーに認証が必要な場合は、以下のコマンドを実行してレジストリーにログインします。

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

b. 以下のコマンドを実行してコンテンツをミラーリングします。

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

1. ミラーリングするカタログのインデックスイメージを指定します。たとえば、これは以前に作成したプルーニングされたインデックスイメージ、または **registry.redhat.io/redhat/redhat-operator-index:v4.6** などのデフォルトカタログのソースインデックスイメージのいずれかである可能性があります。

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

3. オプション: 必要な場合は、レジストリー認証情報ファイルの場所を指定します。**registry.redhat.io** には、**{REG_CREDS}** が必要です。

4. オプション: ターゲットレジストリーの信頼を設定しない場合は、**--insecure** フラグを追加します。

5. オプション: 複数のバリアントが利用可能な場合に、選択するインデックスイメージのプラットフォームおよびアーキテクチャーを指定します。イメージは **'<platform>/<arch>[/<variant>]'** として渡されます。これはインデックスで参照されるイメージには適用されません。使用できる値

は、**linux/amd64**、**linux/ppc64le**、および **linux/s390x** です。

6 オプション: ミラーリングに必要なマニフェストのみを生成し、実際にはイメージコンテンツをレジストリーにミラーリングしません。このオプションは、ミラーリングする内容を確認するのに役立ちます。また、パッケージのサブセットのみが必要な場合に、マッピングの一覧に変更を加えることができます。次に、**mapping.txt** ファイルを **oc image mirror** コマンドで使用し、後のステップでイメージの変更済みの一覧をミラーリングできます。このフラグは、カタログからのコンテンツの高度で選択可能なミラーリングにのみ使用することが意図されています。**opm index prune** をインデックスイメージをプルーニングするために以前にしている場合、これはほとんどのカタログ管理のユースケースに適しています。

出力例

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

1 コマンドで生成された一時的な **index.db** データベースのディレクトリー。

2 生成される manifests ディレクトリー名を記録します。このディレクトリー名は、後の手順で使用されます。

注記

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

- **オプション B: ミラーレジストリーが非接続ホストにある場合** は、以下のアクションを実行します。

  a. ネットワークアクセスが無制限のワークステーションで以下のコマンドを実行し、コンテンツをローカルファイルにミラーリングします。

```
$ oc adm catalog mirror \
    <index_image> \ 1
    file:///local/index \ 2
    [-a ${REG_CREDS}] \
    [--insecure]
```

1 ミラーリングするカタログのインデックスイメージを指定します。たとえば、これは以前に作成したプルーニングされたインデックスイメージ、または **registry.redhat.io/redhat/redhat-operator-index:v4.6** などのデフォルトカタログ

のソースインデックスイメージのいずれかである可能性があります。

2 現在のディレクトリーのローカルファイルにコンテンツをミラーリングします。

出力例

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

To upload local images to a registry, run:

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

1 生成される manifests ディレクトリー名を記録します。このディレクトリー名は、後の手順で使用されます。

2 提供されたインデックスイメージをベースとする、拡張された **file://** パスを記録します。このパスは、後のステップで使用されます。

b. 現在のディレクトリーに生成される **v2/** ディレクトリーをリムーバブルメディアにコピーします。

c. メディアを物理的に削除して、これをミラーレジストリーにアクセスできる非接続環境のホストに割り当てます。

d. ミラーレジストリーに認証が必要な場合は、非接続環境のホストで以下のコマンドを実行し、レジストリーにログインします。

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

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

```
$ oc adm catalog mirror \
    file://local/index/<repo>/<index_image>:<tag> \ 1
    <mirror_registry>:<port>/<namespace> \ 2
    [-a ${REG_CREDS}] \
    [--insecure]
```

1 直前のコマンド出力の **file://** パスを指定します。

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

注記

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

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

注記

manifests ディレクトリー名は後の手順で使用されます。

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

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

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

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

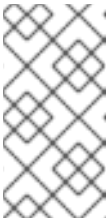
manifests ディレクトリーには以下のファイルが含まれており、これらの一部にはさらに変更が必要になる場合があります。

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

重要

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

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



注記

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

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

重要

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

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

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

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

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

関連情報

- 非接続インストールのイメージのミラーリング
- Operator のアーキテクチャーおよびオペレーティングシステムのサポート
- Package Manifest Format カタログイメージのミラーリング

4.8.5. インデックスイメージからのカタログの作成

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

前提条件

- レジストリーにビルドされ、プッシュされるインデックスイメージ。

手順

1. インデックスイメージを参照する **CatalogSource** オブジェクトを作成します。**oc adm catalog mirror** コマンドを使用してカタログをターゲットレジストリーにミラーリングする場

合、開始点として生成される **catalogSource.yaml** ファイルをそのまま使用することができます。

a. 仕様を以下のように変更し、これを **catalogSource.yaml** ファイルとして保存します。

```
apiVersion: operators.coreos.com/v1alpha1
kind: CatalogSource
metadata:
  name: my-operator-catalog (1)
  namespace: openshift-marketplace (2)
spec:
  sourceType: grpc
  image: <registry>:<port>/<namespace>/redhat-operator-index:v4.6 (3)
  displayName: My Operator Catalog
  publisher: <publisher_name> (4)
  updateStrategy:
    registryPoll: (5)
      interval: 30m
```

(1) レジストリーにアップロードする前にローカルファイルにコンテンツをミラーリングする場合は、**metadata.name** フィールドからバックスラッシュ (/) 文字を削除し、オブジェクトの作成時に invalid resource name エラーを回避します。

(2) カタログソースを全 namespace のユーザーがグローバルに利用できるようにする場合は、**openshift-marketplace** namespace を指定します。それ以外の場合は、そのカタログの別の namespace を対象とし、その namespace のみが利用できるように指定できます。

(3) インデックスイメージを指定します。

(4) カタログを公開する名前または組織名を指定します。

(5) カタログソースは新規バージョンの有無を自動的にチェックし、最新の状態を維持します。

b. このファイルを使用して **CatalogSource** オブジェクトを作成します。

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

2. 以下のリソースが正常に作成されていることを確認します。

a. Pod を確認します。

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

出力例

```
NAME                                READY   STATUS    RESTARTS  AGE
my-operator-catalog-6njx6           1/1     Running   0         28s
marketplace-operator-d9f549946-96sgr  1/1     Running   0         26h
```

b. カタログソースを確認します。

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

出力例

```
NAME                  DISPLAY               TYPE PUBLISHER  AGE
my-operator-catalog   My Operator Catalog   grpc           5s
```

c. パッケージマニフェストを確認します。

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

出力例

```
NAME                     CATALOG               AGE
jaeger-product           My Operator Catalog   93s
```

OpenShift Container Platform Web コンソールで、**OperatorHub** ページから Operator をインストールできるようになりました。

4.8.6. インデックスイメージの更新

カスタムインデックスイメージを参照するカタログソースを使用するように OperatorHub を設定した後に、クラスター管理者はバンドルイメージをインデックスイメージに追加して、クラスターで利用可能な Operator を最新の状態に維持することができます。

opm index add コマンドを使用して既存インデックスイメージを更新できます。ネットワークが制限された環境の場合、更新されたコンテンツもクラスターにミラーリングする必要があります。

前提条件

- **opm** version 1.12.3+
- **podman** version 1.9.3+
- レジストリーにビルドされ、プッシュされるインデックスイメージ。
- インデックスイメージを参照する既存のカタログソース。

手順

1. バンドルイメージを追加して、既存のインデックスを更新します。

```
$ opm index add \
   --bundles <registry>/<namespace>/<new_bundle_image>@sha256:<digest> \ 1
   --from-index <registry>/<namespace>/<existing_index_image>:<existing_tag> \ 2
   --tag <registry>/<namespace>/<existing_index_image>:<updated_tag> \ 3
   --pull-tool podman 4
```

1 **--bundles**フラグは、インデックスに追加する他のバンドルイメージのコンマ区切りリストを指定します。

2 **--from-index**フラグは、以前にプッシュされたインデックスを指定します。

3 **--tag**フラグは、更新されたインデックスイメージに適用するイメージタグを指定します。

4 **--pull-tool**フラグは、コンテナーイメージのプルに使用されるツールを指定します。

ここでは、以下のようになります。

<registry>

quay.ioや**mirror.example.com**などのレジストリーのホスト名を指定します。

<namespace>

ocs-devや**abc**など、レジストリーの namespace を指定します。

<new_bundle_image>

ocs-operatorなど、レジストリーに追加する新しいバンドルイメージを指定します。

<digest>

c7f11097a628f092d8bad148406aa0e0951094a03445fd4bc0775431ef683a41などのバンドルイメージの SHA イメージ ID またはダイジェストを指定します。

<existing_index_image>

abc-redhat-operator-indexなど、以前にプッシュされたイメージを指定します。

<existing_tag>

4.6 など、以前にプッシュされたイメージタグを指定します。

<updated_tag>

4.6.1 など、更新されたインデックスイメージに適用するイメージタグを指定します。

コマンドの例

```
$ opm index add \
    --bundles quay.io/ocs-dev/ocs-
operator@sha256:c7f11097a628f092d8bad148406aa0e0951094a03445fd4bc0775431ef683a
41 \
    --from-index mirror.example.com/abc/abc-redhat-operator-index:4.6 \
    --tag mirror.example.com/abc/abc-redhat-operator-index:4.6.1 \
    --pull-tool podman
```

2. 更新されたインデックスイメージをプッシュします。

```
$ podman push <registry>/<namespace>/<existing_index_image>:<updated_tag>
```

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

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

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

注記

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

4. Operator Lifecycle Manager (OLM) がカタログソースで参照されるインデックスイメージを一定間隔で自動的にポーリングした後に、新規パッケージが正常に追加されたことを確認します。

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

関連情報

- Operator カタログのミラーリング

第5章 OPERATOR の開発

5.1. OPERATOR SDK について

Operator Framework は **Operator** と呼ばれる Kubernetes ネイティブアプリケーションを効果的かつ自動化された拡張性のある方法で管理するためのオープンソースツールキットです。Operator は Kubernetes の拡張性を利用して、プロビジョニング、スケーリング、バックアップおよび復元などのクラウドサービスの自動化の利点を提供し、同時に Kubernetes が実行される場所であればどこでも実行することができます。

Operator により、Kubernetes の上部の複雑で、ステートフルなアプリケーションを管理することが容易になります。ただし、現時点での Operator の作成は、低レベルの API の使用、ボイラープレートの作成、モジュール化の欠如による重複の発生などの課題があるため、困難になる場合があります。

Operator Framework のコンポーネントである Operator SDK は、Operator 開発者が Operator のビルド、テストおよびデプロイに使用できるコマンドラインインターフェイス (CLI) ツールを提供します。

Operator SDK を使用する理由

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

Operator SDK は、controller-runtime ライブラリーを使用して、以下の機能を提供することで Operator を容易に作成するフレームワークです。

- 運用ロジックをより直感的に作成するための高レベルの API および抽象化
- 新規プロジェクトを迅速にブートストラップするためのスキャフォールディングツールおよびコード生成ツール
- Operator Lifecycle Manager (OLM) との統合による、クラスターでの Operator のパッケージング、インストール、および実行の単純化
- 共通する Operator ユースケースに対応する拡張機能
- Prometheus Operator がデプロイされているクラスターで使用できるように、生成された Go ベースの Operator にメトリクスが自動的にセットアップします。

Kubernetes ベースのクラスター (OpenShift Container Platform など) へのクラスター管理者のアクセスのある Operator の作成者は、Operator SDK CLI を使用して Go、Ansible、または Helm をベースに独自の Operator を開発できます。Kubebuilder は Go ベースの Operator のスキャフォールディングソリューションとして Operator SDK に組み込まれます。つまり、既存の Kubebuilder プロジェクトは Operator SDK でそのまま使用でき、引き続き機能します。

注記

OpenShift Container Platform 4.6 は Operator SDK v0.19.4 をサポートします。

5.1.1. Operator について

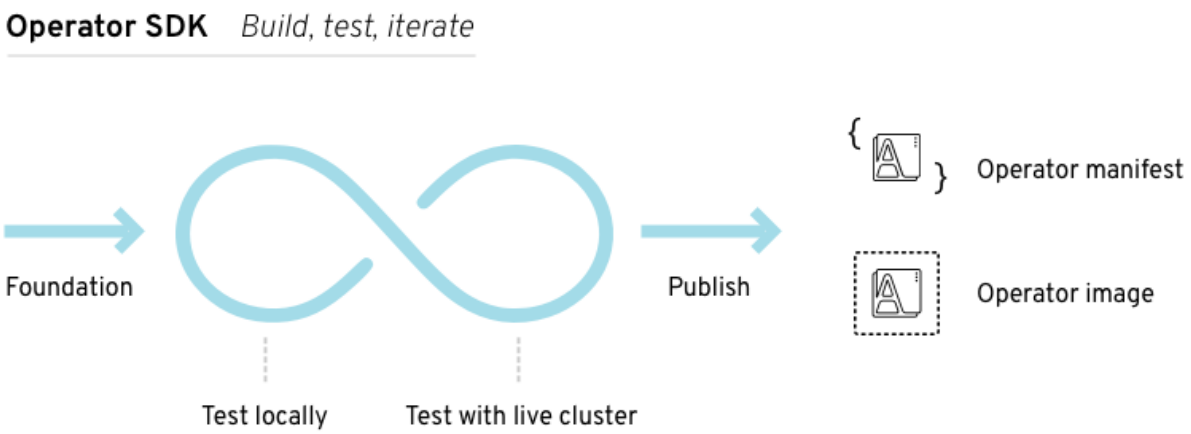
基本的な Operator の概念および用語の概要については、Operator について を参照してください。

5.1.2. 開発ワークフロー

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

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

図5.1 Operator SDK ワークフロー



高次元では、Operator SDK を使用する Operator は Operator の作成者が定義するハンドラーで監視対象のリソースについてのイベントを処理し、アプリケーションの状態を調整するための動作を実行します。

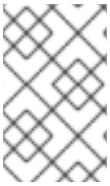
5.1.3. 関連情報

- 認定 Operator ビルドガイド

5.2. OPERATOR SDK CLI のインストール

Operator SDK は、Operator 開発者が Operator のビルド、テストおよびデプロイに使用できるコマンドラインインターフェイス (CLI) ツールを提供します。ワークステーションに Operator SDK CLI をインストールして、独自の Operator のオーサリングを開始する準備を整えることができます。

OpenShift Container Platform 4.6 は Operator SDK v0.19.4 をサポートします。これは、アップストリームソースからインストールすることができます。



注記

OpenShift Container Platform 4.7 以降、Operator SDK は完全にサポートされ、公式の Red Hat 製品ソースから利用できます。回避策については、OpenShift Container Platform 4.7 リリースノート を参照してください。

5.2.1. GitHub リリースからの Operator SDK CLI 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** v1.9.3+、または **buildah** v1.7+
- OpenShift CLI (**oc**) v4.6+ (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
- コンテナーレジストリーへのアクセス

手順

1. リリースバージョン変数を設定します。

```
$ RELEASE_VERSION=v0.19.4
```

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> 1
$ gpg: Can't check signature: No public key
```

1 RSA キー文字列。

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

```
$ gpg [--keyserver keys.gnupg.net] --recv-key "<key_id>" 1
```

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

4. リリースバイナリーを **PATH** にインストールします。

- Linux の場合

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

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

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

- macOS の場合

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

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

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

5. CLI ツールが正しくインストールされていることを確認します。

```
$ operator-sdk version
```

5.2.2. Homebrew からの Operator SDK CLI 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** v1.9.3+、または **buildah** v1.7+
- OpenShift CLI (**oc**) v4.6+ (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
- コンテナーレジストリーへのアクセス

手順

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

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

2. CLI ツールが正しくインストールされていることを確認します。

```
$ operator-sdk version
```

5.2.3. ソースからの Operator SDK CLI の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** v1.9.3+、または **buildah** v1.7+
- OpenShift CLI (**oc**) v4.6+ (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
- コンテナーレジストリーへのアクセス

手順

1. **operator-sdk** リポジトリーのクローンを作成します。

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

2. クローンしたリポジトリーのディレクトリーに移動します。

```
$ cd operator-sdk
```

3. v0.19.4 リリースをチェックアウトします。

```
$ git checkout tags/v0.19.4 -b v0.19.4
```

4. 依存関係を更新します。

```
$ make tidy
```

5. SDK CLI ツールをコンパイルし、インストールします。

```
$ 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。

6. 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
```

5.3. GO ベースの OPERATOR の作成

Operator 開発者は、Operator SDK での Go プログラミング言語のサポートを利用して、Go ベースの Memcached Operator のサンプルをビルドして、分散キー/値のストアを作成し、そのライフサイクルを管理することができます。

注記

Kubebuilder は Go ベース Operator のスキャフォールディングソリューションとして、Operator SDK に組み込まれます。

5.3.1. Operator SDK を使用した Go ベース Operator の作成

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ー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 を作成する例を示します。

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 v0.19.4 CLI
- OpenShift Container Platform 4.6 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**) v4.6+ (インストール済み)

手順

1. Operator プロジェクトを作成します。
   a. プロジェクトのディレクトリーを作成します。

```
$ mkdir -p $HOME/projects/memcached-operator
```

   b. ディレクトリーに切り替えます。

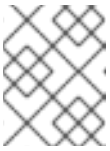
```
$ cd $HOME/projects/memcached-operator
```

   c. Go モジュールのサポートをアクティブにします。

```
$ export GO111MODULE=on
```

   d. **operator-sdk init** コマンドを実行してプロジェクトを初期化します。

```
$ operator-sdk init \
    --domain=example.com \
    --repo=github.com/example-inc/memcached-operator
```



注記

operator-sdk init コマンドは、デフォルトで **go.kubebuilder.io/v2** プラグインを使用します。

2. サポートされるイメージを使用するよう Operator を更新します。
   a. プロジェクトのルートレベルの Dockerfile で、デフォルトのランナーイメージの参照フォームを変更します。

```
FROM gcr.io/distroless/static:nonroot
```

   以下のように変更してください。

```
FROM registry.access.redhat.com/ubi8/ubi-minimal:latest
```

   b. Go プロジェクトのバージョンによっては、 Dockerfile に **USER 65532:65532** または **USER nonroot:nonroot** ディレクティブが含まれる可能性があります。いずれの場合も、サポートされるランナーイメージでは必要ないため、その行を削除します。

   c. **config/default/manager_auth_proxy_patch.yaml** ファイルで、以下の **image** の値を変更します。

```
gcr.io/kubebuilder/kube-rbac-proxy:<tag>
```

   サポートされるイメージを使用するには、以下へ変更します。

```
registry.redhat.io/openshift4/ose-kube-rbac-proxy:v4.6
```

3. 以下の行を置き換えて、Makefile の **test** ターゲットを更新し、後続のビルドで必要な依存関係をインストールできるようにします。

例5.1 既存の test ターゲット

```
test: generate fmt vet manifests
	go test ./... -coverprofile cover.out
```

以下の行を使用します。

例5.2 更新された test ターゲット

```
ENVTEST_ASSETS_DIR=$(shell pwd)/testbin
test: manifests generate fmt vet ## Run tests.
 mkdir -p ${ENVTEST_ASSETS_DIR}
 test -f ${ENVTEST_ASSETS_DIR}/setup-envtest.sh || curl -sSLo
${ENVTEST_ASSETS_DIR}/setup-envtest.sh
https://raw.githubusercontent.com/kubernetes-sigs/controller-runtime/v0.7.2/hack/setup-envtest.sh
 source ${ENVTEST_ASSETS_DIR}/setup-envtest.sh; fetch_envtest_tools
$(ENVTEST_ASSETS_DIR); setup_envtest_env $(ENVTEST_ASSETS_DIR); go test ./...
-coverprofile cover.out
```

4. カスタムリソース定義 (CRD) API およびコントローラーを作成します。

   a. 以下のコマンドを実行して、グループ **cache**、バージョン **v1**、および種類 **Memcached** を指定して API を作成します。

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

   b. プロンプトが表示されたら **y** を入力し、リソースとコントローラーの両方を作成します。

   ```
   Create Resource [y/n]
   y
   Create Controller [y/n]
   y
   ```

   出力例

   ```
   Writing scaffold for you to edit...
   api/v1/memcached_types.go
   controllers/memcached_controller.go
   ...
   ```

   このプロセスでは、**api/v1/memcached_types.go** で Memcached リソース API が生成され、**controllers/memcached_controller.go** でコントローラーが生成されます。

   c. **api/v1/memcached_types.go** で Go タイプの定義を変更し、以下の **spec** および **status** を追加します。

   ```
   // MemcachedSpec defines the desired state of Memcached
   ```

```
type MemcachedSpec struct {
 // +kubebuilder:validation:Minimum=0
 // Size is the size of the memcached deployment
 Size int32 `json:"size"`
}

// MemcachedStatus defines the observed state of Memcached
type MemcachedStatus struct {
 // Nodes are the names of the memcached pods
 Nodes []string `json:"nodes"`
}
```

d. **+kubebuilder:subresource:status** マーカーを追加し、**status** サブリソースを CRD マニフェストに追加します。

```
// Memcached is the Schema for the memcacheds API
// +kubebuilder:subresource:status (1)
type Memcached struct {
 metav1.TypeMeta   `json:",inline"`
 metav1.ObjectMeta `json:"metadata,omitempty"`

 Spec   MemcachedSpec   `json:"spec,omitempty"`
 Status MemcachedStatus `json:"status,omitempty"`
}
```

(1) この行を追加します。

これにより、コントローラーは残りの CR オブジェクトを変更せずに CR ステータスを更新できます。

e. リソースタイプ用に生成されたコードを更新します。

```
$ make generate
```

ヒント

***_types.go** ファイルの変更後は、**make generate** コマンドを実行し、該当するリソースタイプ用に生成されたコードを更新する必要があります。

上記の Makefile ターゲットは **controller-gen** ユーティリティーを呼び出して、**api/v1/zz_generated.deepcopy.go** ファイルを更新します。これにより、API Go タイプの定義は、すべての **Kind** タイプが実装する必要のある **runtime.Object** インターフェイスが実装されます。

5. CRD マニフェストを生成して更新します。

```
$ make manifests
```

この Makefile ターゲットは **controller-gen** ユーティリティーを呼び出し、**config/crd/bases/cache.example.com_memcacheds.yaml** ファイルに CRD マニフェストを生成します。

a. 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 の作成または更新時に **Memcached** カスタムリソース (CR) のプロパティーを検証できます。

Operator の作成者は Kubebuilder **markers** と呼ばれるアノテーションのような、単一行のコメントを使用して 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 OpenAPI 検証セクションを更新します。

```
$ make manifests
```

6. 新規 API およびコントローラーの作成後に、コントローラーロジックを実装することができます。この例では、生成されたコントローラーファイル **controllers/memcached_controller.go** を以下の実装例に置き換えます。

例5.3 memcached_controller.go の例

```
/*
Licensed under the Apache License, Version 2.0 (the "License");
you may not use this file except in compliance with the License.
You may obtain a copy of the License at

    http://www.apache.org/licenses/LICENSE-2.0

Unless required by applicable law or agreed to in writing, software
distributed under the License is distributed on an "AS IS" BASIS,
WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
See the License for the specific language governing permissions and
limitations under the License.
*/

package controllers

import (
 "context"
 "reflect"

 "github.com/go-logr/logr"
 appsv1 "k8s.io/api/apps/v1"
 corev1 "k8s.io/api/core/v1"
 "k8s.io/apimachinery/pkg/api/errors"
 metav1 "k8s.io/apimachinery/pkg/apis/meta/v1"
 "k8s.io/apimachinery/pkg/runtime"
 "k8s.io/apimachinery/pkg/types"
 ctrl "sigs.k8s.io/controller-runtime"
```

```
 "sigs.k8s.io/controller-runtime/pkg/client"

 cachev1 "github.com/example-inc/memcached-operator/api/v1"
)

// MemcachedReconciler reconciles a Memcached object
type MemcachedReconciler struct {
 client.Client
 Log    logr.Logger
 Scheme *runtime.Scheme
}

//
+kubebuilder:rbac:groups=cache.example.com,resources=memcacheds,verbs=get;list;watch
;create;update;patch;delete
//
+kubebuilder:rbac:groups=cache.example.com,resources=memcacheds/status,verbs=get;up
date;patch
//
+kubebuilder:rbac:groups=apps,resources=deployments,verbs=get;list;watch;create;update;
patch;delete
// +kubebuilder:rbac:groups=core,resources=pods,verbs=get;list;

func (r *MemcachedReconciler) Reconcile(req ctrl.Request) (ctrl.Result, error) {
 ctx := context.Background()
 log := r.Log.WithValues("memcached", req.NamespacedName)

 // Fetch the Memcached instance
 memcached := &cachev1.Memcached{}
 err := r.Get(ctx, req.NamespacedName, memcached)
 if err != nil {
  if errors.IsNotFound(err) {
   // Request object not found, could have been deleted after reconcile request.
   // Owned objects are automatically garbage collected. For additional cleanup logic use
finalizers.
   // Return and don't requeue
   log.Info("Memcached resource not found. Ignoring since object must be deleted")
   return ctrl.Result{}, nil
  }
  // Error reading the object - requeue the request.
  log.Error(err, "Failed to get Memcached")
  return ctrl.Result{}, err
 }

 // Check if the deployment already exists, if not create a new one
 found := &appsv1.Deployment{}
 err = r.Get(ctx, types.NamespacedName{Name: memcached.Name, Namespace:
memcached.Namespace}, found)
 if err != nil && errors.IsNotFound(err) {
  // Define a new deployment
  dep := r.deploymentForMemcached(memcached)
  log.Info("Creating a new Deployment", "Deployment.Namespace", dep.Namespace,
"Deployment.Name", dep.Name)
  err = r.Create(ctx, dep)
  if err != nil {
   log.Error(err, "Failed to create new Deployment", "Deployment.Namespace",
```

```
dep.Namespace, "Deployment.Name", dep.Name)
    return ctrl.Result{}, err
   }
   // Deployment created successfully - return and requeue
   return ctrl.Result{Requeue: true}, nil
  } else if err != nil {
   log.Error(err, "Failed to get Deployment")
   return ctrl.Result{}, err
  }

  // Ensure the deployment size is the same as the spec
  size := memcached.Spec.Size
  if *found.Spec.Replicas != size {
   found.Spec.Replicas = &size
   err = r.Update(ctx, found)
   if err != nil {
    log.Error(err, "Failed to update Deployment", "Deployment.Namespace",
found.Namespace, "Deployment.Name", found.Name)
    return ctrl.Result{}, err
   }
   // Spec updated - return and requeue
   return ctrl.Result{Requeue: true}, nil
  }

  // Update the Memcached status with the pod names
  // List the pods for this memcached's deployment
  podList := &corev1.PodList{}
  listOpts := []client.ListOption{
   client.InNamespace(memcached.Namespace),
   client.MatchingLabels(labelsForMemcached(memcached.Name)),
  }
  if err = r.List(ctx, podList, listOpts...); err != nil {
   log.Error(err, "Failed to list pods", "Memcached.Namespace", memcached.Namespace,
"Memcached.Name", memcached.Name)
   return ctrl.Result{}, err
  }
  podNames := getPodNames(podList.Items)

  // Update status.Nodes if needed
  if !reflect.DeepEqual(podNames, memcached.Status.Nodes) {
   memcached.Status.Nodes = podNames
   err := r.Status().Update(ctx, memcached)
   if err != nil {
    log.Error(err, "Failed to update Memcached status")
    return ctrl.Result{}, err
   }
  }

  return ctrl.Result{}, nil
 }

 // deploymentForMemcached returns a memcached Deployment object
 func (r *MemcachedReconciler) deploymentForMemcached(m *cachev1.Memcached)
 *appsv1.Deployment {
  ls := labelsForMemcached(m.Name)
  replicas := m.Spec.Size
```

```
 dep := &appsv1.Deployment{
  ObjectMeta: metav1.ObjectMeta{
   Name:      m.Name,
   Namespace: m.Namespace,
  },
  Spec: appsv1.DeploymentSpec{
   Replicas: &replicas,
   Selector: &metav1.LabelSelector{
    MatchLabels: ls,
   },
   Template: corev1.PodTemplateSpec{
    ObjectMeta: metav1.ObjectMeta{
     Labels: ls,
    },
    Spec: corev1.PodSpec{
     Containers: []corev1.Container{{
      Image:   "memcached:1.4.36-alpine",
      Name:    "memcached",
      Command: []string{"memcached", "-m=64", "-o", "modern", "-v"},
      Ports: []corev1.ContainerPort{{
       ContainerPort: 11211,
       Name:          "memcached",
      }},
     }},
    },
   },
  },
 }
 // Set Memcached instance as the owner and controller
 ctrl.SetControllerReference(m, dep, r.Scheme)
 return dep
}

// labelsForMemcached returns the labels for selecting the resources
// belonging to the given memcached CR name.
func labelsForMemcached(name string) map[string]string {
 return map[string]string{"app": "memcached", "memcached_cr": name}
}

// getPodNames returns the pod names of the array of pods passed in
func getPodNames(pods []corev1.Pod) []string {
 var podNames []string
 for _, pod := range pods {
  podNames = append(podNames, pod.Name)
 }
 return podNames
}

func (r *MemcachedReconciler) SetupWithManager(mgr ctrl.Manager) error {
 return ctrl.NewControllerManagedBy(mgr).
  For(&cachev1.Memcached{}).
  Owns(&appsv1.Deployment{}).
  Complete(r)
}
```

コントローラーのサンプルは、それぞれの **Memcached** CR について以下の調整 (reconciliation) ロジックを実行します。

- Memcached デプロイメントを作成します (ない場合)。
- デプロイメントのサイズが、**Memcached** CR 仕様で指定されたものと同じであることを確認します。
- **Memcached** CR ステータスを **memcached** Pod の名前に置き換えます。

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

a. **controllers/memcached_controller.go** ファイルでコントローラーの実装を検査し、コントローラーのリソースの監視方法を確認します。
SetupWithManager() 関数は、CR およびコントローラーによって所有され、管理される他のリソースを監視するようにコントローラーがビルドされる方法を指定します。

例5.4 SetupWithManager() 関数

```
import (
 ...
 appsv1 "k8s.io/api/apps/v1"
 ...
)

func (r *MemcachedReconciler) SetupWithManager(mgr ctrl.Manager) error {
 return ctrl.NewControllerManagedBy(mgr).
  For(&cachev1.Memcached{}).
  Owns(&appsv1.Deployment{}).
  Complete(r)
}
```

NewControllerManagedBy() は、さまざまなコントローラー設定を可能にするコントローラービルダーを提供します。

For(&cachev1.Memcached{}) は、監視するプライマリーリソースとして **Memcached** タイプを指定します。**Memcached** タイプのそれぞれの Add、Update、または Delete イベントの場合、reconcile ループに **Memcached** オブジェクトの (namespace および name キーから成る) reconcile **Request** 引数が送られます。

Owns(&appsv1.Deployment{}) は、監視するセカンダリーリソースとして **Deployment** タイプを指定します。Add、Update、または Delete イベントの各 **Deployment** タイプの場合、イベントハンドラーは各イベントを、デプロイメントのオーナーの reconcile request にマップします。この場合、デプロイメントが作成された **Memcached** オブジェクトがオーナーです。

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

例5.5 reconcile ループ

```
import (
 ctrl "sigs.k8s.io/controller-runtime"

 cachev1 "github.com/example-inc/memcached-operator/api/v1"
 ...
)

func (r *MemcachedReconciler) Reconcile(ctx context.Context, req ctrl.Request)
(ctrl.Result, error) {
  // Lookup the Memcached instance for this reconcile request
  memcached := &cachev1.Memcached{}
  err := r.Get(ctx, req.NamespacedName, memcached)
  ...
}
```

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。

例5.6 再キューロジック

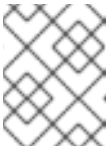
```
// Reconcile successful - don't requeue
return reconcile.Result{}, nil
// Reconcile failed due to error - requeue
return reconcile.Result{}, err
// Requeue for any reason other than error
return reconcile.Result{Requeue: true}, nil
```

Result.RequeueAfter を設定して、猶予期間後に要求を再びキューに入れることができます。

例5.7 猶予期間後の再キュー

```
import "time"

// Reconcile for any reason other than an error after 5 seconds
return ctrl.Result{RequeueAfter: time.Second*5}, nil
```



注記

RequeueAfter を定期的な CR の調整に設定している **Result** を返すことができます。

reconciler、クライアント、およびリソースイベントとの対話に関する詳細は、Controller Runtime Client API のドキュメントを参照してください。

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。

5.3.2. Operator の実行

Operator SDK CLI を使用して Operator をビルドし、実行する方法は 2 つあります。

- クラスター外で Go プログラムとしてローカルに実行します。
- クラスター上のデプロイメントとして実行します。

前提条件

- Operator SDK を使用した Go ベース Operator の作成 で説明されているようにな Go ベース Operator プロジェクトがある。

5.3.2.1. クラスター外でローカルに実行する。

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

手順

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

```
$ make install run
```

例5.8 出力例

```
...
2021-01-10T21:09:29.016-0700 INFO controller-runtime.metrics metrics server is starting to listen {"addr": ":8080"}
2021-01-10T21:09:29.017-0700 INFO setup starting manager
2021-01-10T21:09:29.017-0700 INFO controller-runtime.manager starting metrics server {"path": "/metrics"}
2021-01-10T21:09:29.018-0700 INFO controller-runtime.manager.controller.memcached Starting EventSource {"reconciler group": "cache.example.com", "reconciler kind": "Memcached", "source": "kind source: /, Kind="}
2021-01-10T21:09:29.218-0700 INFO controller-runtime.manager.controller.memcached Starting Controller {"reconciler group": "cache.example.com", "reconciler kind": "Memcached"}
2021-01-10T21:09:29.218-0700 INFO controller-runtime.manager.controller.memcached Starting workers {"reconciler group": "cache.example.com", "reconciler kind": "Memcached", "worker count": 1}
```

5.3.2.2. デプロイメントとしての実行

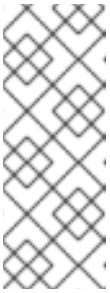
Go ベースの Operator プロジェクトの作成後に、Operator をクラスター内でデプロイメントとしてビルドし、実行できます。

手順

1. 以下の **make** コマンドを実行して Operator イメージをビルドし、プッシュします。以下の手順の **IMG** 引数を変更して、アクセス可能なリポジトリーを参照します。Quay.io などのリポジトリーサイトにコンテナーを保存するためのアカウントを取得できます。

   a. イメージをビルドします。

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

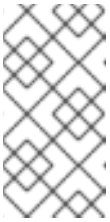
   

   注記

   Operator の SDK によって生成される Dockerfile は、**go build** について **GOARCH=amd64** を明示的に参照します。これは、AMD64 アーキテクチャー以外の場合は **GOARCH=$TARGETARCH** に修正できます。Docker は、**-platform** で指定された値に環境変数を自動的に設定します。Buildah では、そのために **-build-arg** を使用する必要があります。詳細は、Multiple Architectures を参照してください。

   b. イメージをリポジトリーにプッシュします。

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

   注記

   両方のコマンドのイメージの名前とタグ (例: **IMG=<registry>/<user>/<image_name>:<tag>**) を Makefile に設定することもできます。**IMG ?= controller:latest** の値を変更して、デフォルトのイメージ名を設定します。

2. 以下のコマンドを実行して Operator をデプロイします。

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

   デフォルトで、このコマンドは **<project_name>-system** の形式で Operator プロジェクトの名前で namespace を作成し、デプロイメントに使用します。このコマンドは、**config/rbac** から RBAC マニフェストもインストールします。

3. Operator が実行されていることを確認します。

   ```
   $ oc get deployment -n <project_name>-system
   ```

   出力例

   ```
   NAME                                    READY   UP-TO-DATE   AVAILABLE   AGE
   <project_name>-controller-manager       1/1     1            1           8m
   ```

5.3.3. カスタムリソースの作成

Operator のインストール後に、Operator によってクラスターに提供されるカスタムリソース (CR) を作成して、これをテストできます。

前提条件

- クラスターにインストールされている **Memcached** CR を提供する Memcached Operator の例

手順

1. Operator がインストールされている namespace へ変更します。たとえば、**make deploy** コマンドを使用して Operator をデプロイした場合は、以下のようになります。

```
$ oc project memcached-operator-system
```

2. **config/samples/cache_v1_memcached.yaml** で **Memcached** CR マニフェストのサンプルを編集し、以下の仕様が含まれるようにします。

```
apiVersion: cache.example.com/v1
kind: Memcached
metadata:
  name: memcached-sample
...
spec:
...
  size: 3
```

3. CR を作成します。

```
$ oc apply -f config/samples/cache_v1_memcached.yaml
```

4. **Memcached** Operator が、正しいサイズで CR サンプルのデプロイメントを作成することを確認します。

```
$ oc get deployments
```

出力例

```
NAME                                  READY   UP-TO-DATE   AVAILABLE   AGE
memcached-operator-controller-manager   1/1     1            1           8m
memcached-sample                      3/3     3            3           1m
```

5. ステータスが Memcached Pod 名で更新されていることを確認するために、Pod および CR ステータスを確認します。

   a. Pod を確認します。

```
$ oc get pods
```

   出力例

```
NAME                                  READY     STATUS    RESTARTS   AGE
memcached-sample-6fd7c98d8-7dqdr      1/1       Running   0          1m
memcached-sample-6fd7c98d8-g5k7v      1/1       Running   0          1m
memcached-sample-6fd7c98d8-m7vn7      1/1       Running   0          1m
```

   b. CR ステータスを確認します。

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

出力例

```
apiVersion: cache.example.com/v1
kind: Memcached
metadata:
...
  name: memcached-sample
...
spec:
  size: 3
status:
  nodes:
  - memcached-sample-6fd7c98d8-7dqdr
  - memcached-sample-6fd7c98d8-g5k7v
  - memcached-sample-6fd7c98d8-m7vn7
```

6. デプロイメントサイズを更新します。

   a. **config/samples/cache_v1_memcached.yaml** ファイルを更新し、**Memcached** CR の **spec.size** フィールドを **3** から **5** に変更します。

```
$ oc patch memcached memcached-sample \
    -p '{"spec":{"size": 5}}' \
    --type=merge
```

   b. Operator がデプロイメントサイズを変更することを確認します。

```
$ oc get deployments
```

出力例

```
NAME                                  READY   UP-TO-DATE   AVAILABLE   AGE
memcached-operator-controller-manager   1/1     1            1           10m
memcached-sample                        5/5     5            5           3m
```

5.3.4. 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

5.4. 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。

5.4.1. 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。

5.4.1.1. 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5.1 カスタムリソースフィールド

フィールド	説明
apiVersion	作成される CR のバージョン。
kind	作成される CR の種類。
metadata	作成される Kubernetes 固有のメタデータ。
spec (オプション)	Ansible に渡される変数のキーと値の一覧。このフィールドは、デフォルトでは空です。
status	オブジェクトの現在の状態の概要を示します。Ansible ベースの Operator の場合、**status** サブリソース はデフォルトで CRD について有効にされ、**operator_sdk.util.k8s_status** Ansible モジュールによって管理されます。 これには、CR の **status** に対する **condition** 情報が含まれます。
annotations	CR に付加する Kubernetes 固有のアノテーション。

CR アノテーションの以下の一覧は Operator の動作を変更します。

表5.2 Ansible ベースの Operator アノテーション

アノテーション	説明
ansible.operator-sdk/reconcile-period	CR の調整間隔を指定します。この値は標準的な Golang パッケージ **time** を使用して解析されます。とくに、**ParseDuration** は、**s** のデフォルト接尾辞を適用し、秒単位で値を指定します。

Ansible ベースの Operator アノテーションの例

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

```
  name: "example"
annotations:
  ansible.operator-sdk/reconcile-period: "30s"
```

5.4.1.2. 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5.3 watches.yaml ファイルのマッピング

フィールド	説明
group	監視する CR のグループ。
version	監視する CR のバージョン。
kind	監視する CR の種類。
role (デフォルト)	コンテナーに追加される Ansible ロールへのパスです。たとえば、 **roles** ディレクトリーが **/opt/ansible/roles/** にあり、ロールの名前が **busybox** の場合、この値は **/opt/ansible/roles/busybox** になります。このフィールドは **playbook** フィールドと相互に排他的です。
playbook	コンテナーに追加される Ansible Playbook へのパスです。この Playbook の使用はロールを呼び出す方法になります。このフィールドは **role** フィールドと相互に排他的です。
reconcilePeriod (オプション)	ロールまたは Playbook が特定の CR について実行される調整期間および頻度。
manageStatus (オプション)	**true** (デフォルト) に設定されると、Operator は CR のステータスを汎用的に管理します。 **false** に設定されると、指定されたロール、または別のコントローラーの Playbook により、CR のステータスは他の場所で管理されます。

watches.yaml ファイルの例

```
- version: v1alpha1 (1)
  group: test1.example.com
  kind: Test1
  role: /opt/ansible/roles/Test1

- version: v1alpha1 (2)
  group: test2.example.com
  kind: Test2
  playbook: /opt/ansible/playbook.yml

- version: v1alpha1 (3)
```

```
group: test3.example.com
kind: Test3
playbook: /opt/ansible/test3.yml
reconcilePeriod: 0
manageStatus: false
```

1. **Test1** の **test1** ロールへの単純なマッピングの例。
2. **Test2** の Playbook への単純なマッピングの例。
3. **Test3** の種類についてのより複雑な例。Playbook での CR ステータスを再度キューに入れるタスクまたはその管理を無効にします。

5.4.1.2.1. 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5.4 高度な watches.yaml ファイルのオプション

機能	YAML キー	説明	上書きのアノテーション	デフォルト値
調整期間	**reconcilePeriod**	特定の CR についての調整実行の間隔。	**ansible.operator-sdk/reconcile-period**	**1m**
ステータスの管理	**manageStatus**	Operator は各 CR の **status** セクションの **conditions** セクションを管理できます。		**true**
依存するリソースの監視	**watchDependentResources**	Operator は Ansible によって作成されるリソースを動的に監視できます。		**true**
クラスタースコープのリソースの監視	**watchClusterScopedResources**	Operator は Ansible によって作成されるクラスタースコープのリソースを監視できます。		**false**
最大 Runner アーティファクト	**maxRunnerArtifacts**	Ansible Runner が各リソースについて Operator コンテナーに保持する アーティファクトディレクトリー の数を管理します。	**ansible.operator-sdk/max-runner-artifacts**	**20**

高度なオプションを含む watches.yml ファイルの例

```
- version: v1alpha1
  group: app.example.com
  kind: AppService
  playbook: /opt/ansible/playbook.yml
  maxRunnerArtifacts: 30
  reconcilePeriod: 5s
  manageStatus: False
  watchDependentResources: False
```

5.4.1.3. 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 }}"
```

5.4.1.4. Ansible Runner ディレクトリー

Ansible Runner はコンテナーに Ansible 実行についての情報を維持します。これは **/tmp/ansible-operator/runner/<group>/<version>/<kind>/<namespace>/<name>** に置かれます。

関連情報

- **runner** ディレクトリーについての詳細は、Ansible Runner ドキュメント を参照してください。

5.4.2. 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 v0.19.4 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 4.6 など) へのアクセス
- OpenShift CLI (**oc**) v4.6+ (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ーリングのための 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/v1apha1** および Kind **Memcached** の **Memcached** リソースを監視するための **memcached-operator** プロジェクトが作成されます。

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

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

ヒント

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

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

```
size: 1
```

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

ここでの目的は、Ansible で **memcached:1.4.36-alpine** イメージを実行する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。

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

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

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

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

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

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

   iii. **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
   ```

   iv. **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
   ```

b. **クラスター外で実行します**。この方法は、デプロイメントおよびテストの速度を上げるために開発サイクル時に優先される方法です。
   Ansible Runner および Ansible Runner HTTP プラグインがインストールされていることを確認します。 インストールされていない場合、CR の作成時に Ansible Runner から予想しないエラーが発生します。

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

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

   ```
   $ operator-sdk 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-7cc7cfdf86-vvjqk   1/1       Running   0          2m
   ```

7. **サイズを更新します。**

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

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

   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
```

5.4.3. 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。

5.4.3.1. 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
```

5.4.3.2. 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 (1)
```

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

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
```

5.4.3.3. 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。

5.4.3.3.1. 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
```

5.4.3.3.2. 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 1
```

1 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
```

5.4.4. 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、新たにスキャ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。

5.4.5. 追加リソース

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

5.5. 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。

5.5.1. Operator SDK での Helm チャートのサポート

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

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

Operator の主な機能として、アプリケーションインスタンスを表すカスタムオブジェクトから読み取り、必要な状態を実行されている内容に一致させることができます。Helm ベース Operator の場合、オブジェクトの **spec** フィールドは、通常 Helm の **values.yaml** ファイルに記述される設定オプションの一覧です。Helm CLI を使用してフラグ付きの値を設定する代わりに (例: **helm install -f values.yaml**)、これらをカスタムリソース (CR) 内で表現することができます。 これにより、ネイティブ Kubernetes オブジェクトとして、適用される RBAC および監査証跡の利点を活用できます。

Tomcat という単純な CR の例:

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

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

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

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

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

Helm CLI を使用したり、Tiller をインストールしたりする必要はありません。Helm ベースの Operator はコードを Helm プロジェクトからインポートします。Operator のインスタンスを実行状態にし、カスタムリソース定義 (CRD) で CR を登録することのみが必要になります。これは RBAC に準拠するため、実稼働環境の変更を簡単に防止することができます。

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

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

ヒント

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

前提条件

- 開発ワークステーションにインストールされる Operator SDK v0.19.4 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 4.6 など) へのアクセス
- OpenShift CLI (**oc**) v4.6+ (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/v1apha1** および Kind **Nginx** の Nginx リソースを監視する目的で **nginx-operator** プロジェクトが作成されます。

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。

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

ii. 同様に、デフォルトのサービスポートは **80** に設定されます。**8080** を代わりに使用するには、サービスポートの上書きを追加して
deploy/crds/example.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。

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

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

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

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

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

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

iii. **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
```

iv. **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
```

5.5.3. 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

5.6. 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 コンテナーイメージに伴うメタデータです。CSV は、Operator が必要とする RBAC ルールやそれが管理したり、依存したりするカスタムリソース (CR) などの Operator の実行に必要な技術情報の情報源でもあります。

Operator SDK には、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。

5.6.1. 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 の生成に必要になります。

- **operator.yaml**
- ***_{crd,cr}.yaml**
- RBAC ロールファイル (例: **role.yaml**)

Operator の作者にはこれらのファイルについてそれぞれ異なるバージョン管理の要件がある場合があり、**deploy/olm-catalog/csv-config.yaml** ファイルに組み込む特定のファイルを設定できます。

ワークフロー
検出される既存の CSV に応じて、またすべての設定のデフォルト値が使用されることを仮定すると、**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。

5.6.2. 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]**

5.6.3. 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、Operator および各種のカスタムリソース定義 (CRD) に関する人間が作成するメタデータです。

Operator 作成者はそれらのクラスターサービスバージョン (CSV) YAML ファイルを直接変更する必要があり、パーソナライズ設定されたデータを以下の必須フィールドに追加します。Operator SDK は、必須フィールドのいずれかにデータが欠落していることが検出されると、CSV 生成時に警告を送信します。

以下の表は、手動で定義された CSV フィールドのうち、必須フィールドとオプションフィールドについて詳細に示しています。

表5.5 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。

フィールド	説明
spec.description	Operator の機能についての簡単な説明。
spec.keywords	Operator について記述するキーワード。
spec.maintainers	**name** および **email** を持つ、Operator を維持する人または組織上のエンティティー
spec.provider	**name** を持つ、Operator のプロバイダー (通常は組織)。
spec.labels	Operator 内部で使用されるキー/値のペア。
spec.version	Operator のセマンティクスバージョン。 例: **0.1.1**。
spec.customresourcedefinitions	Operator が使用する任意の CRD。このフィールドは、CRD YAML ファイルが **deploy/** にある場合に Operator SDK によって自動的に設定されます。ただし、CRD マニフェスト仕様にない複数のフィールドでは、ユーザーの入力が必要です。 ● **description**: description of the CRD. ● **resources**: CRD によって利用される任意の Kubernetes リソース (例:**Pod** および **StatefulSet** オブジェクト)。 ● **specDescriptors**: Operator の入力および出力についての UI ヒント。

表5.6 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

5.6.3.1. Operator メタデータアノテーション

Operator 開発者は、クラスターサービスバージョン (CSV) のメタデータで特定のアノテーションを手動で定義し、OperatorHub などのユーザーインターフェイス (UI) の機能を有効にしたり、機能を強調したりできます。

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

表5.7 アノテーション

フィールド	説明
alm-examples	カスタムリソース定義 (CRD) テンプレートに最低限の設定セットを指定します。互換性のある UI は、ユーザーがさらにカスタマイズできるようにこのテンプレートの事前入力を行います。
operatorframework.io/initialization-resource	Operator のインストール時に作成する必要のある単一の必要なカスタムリソースを指定します。完全な YAML 定義が含まれるテンプレートを含める必要があります。
operatorframework.io/suggested-namespace	Operator をデプロイする必要のある推奨 namespace を設定します。

フィールド	説明
operators.openshift.io/infrastructure-features	Operator によってサポートされるインフラストラクチャー機能。ユーザーは、Web コンソールで OperatorHub を使用して Operator を検出する際に、これらの機能で表示してフィルターを実行できます。有効で、大文字と小文字が区別される値は以下のとおりです。 ● **disconnected**: Operator はすべての依存関係を含む非接続カタログにミラーリングされるため、インターネットへのアクセスは必要ありません。ミラーリングに必要なすべての関連イメージが Operator によって一覧表示されます。 ● **cnf**: Operator は Cloud-native Network Functions (CNF) Kubernetes プラグインを提供します。 ● **CNI**: Operator は Container Network Interface (CNI) Kubernetes プラグインを提供します。 ● **csi**: Operator は Container Storage Interface (CSI) Kubernetes プラグインを提供します。 ● **fips**: Operator は基礎となるプラットフォームの FIPS モードを受け入れ、FIPS モードで起動されるノードで機能します。 **重要** FIPS 検証済み/進行中のモジュール (Modules in Process) 暗号ライブラリーの使用は、**x86_64** アーキテクチャーの OpenShift Container Platform デプロイメントでのみサポートされています。 ● **proxy-aware**: Operator は、プロキシーの背後にあるクラスターでの実行をサポートします。Operator は、クラスターがプロキシーを使用するように設定される際に Operator Lifecycle Manager (OLM) が Operator に自動的に提供する標準のプロキシー環境変数の **HTTP_PROXY** および **HTTPS_PROXY** を受け入れます。必要な環境変数は、管理されているワークロード用にオペランドに渡されます。
operators.openshift.io/valid-subscription	Operator を使用するために必要とされる特定のサブスクリプションを一覧表示するための自由形式の配列です。例: **['"3Scale Commercial License", "Red Hat Managed Integration"]'**

フィールド	説明
operators.operatorframework.io/internal-objects	ユーザーの操作を目的としていない UI の CRD を非表示にします。

使用例

Operator は非接続およびプロキシー対応をサポートします

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

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

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

Operator には 3scale ライセンスが必要です

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

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

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

関連情報

- CRD テンプレート
- 必要なカスタムリソースの初期化
- 推奨される namespace の設定
- ネットワークが制限された環境についての Operator の有効化(非接続モード)
- 内部オブジェクトの非表示
- FIPS 暗号のサポート

5.6.4. 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。

5.6.5. 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、Operator がネットワークが制限された環境、または非接続の環境で適切に実行されるよう追加要件を満たすことを確認する必要があります。

非接続モードをサポートするための Operator の要件

- Operator のクラスターサービスバージョン (CSV) で以下を行います。
  - Operator がそれらの機能を実行するために必要となる可能性のある **関連イメージ** または他のコンテナーを一覧表示します。
  - 指定されたすべてのイメージを、タグではなくダイジェスト (SHA) で参照します。
- Operator のすべての依存関係は、非接続モードでの実行もサポートする必要があります。
- Operator にはクラスター外のリソースは必要ありません。

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

前提条件

- CSV を含む Operator プロジェクト

手順

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

```
...
spec:
  relatedImages: 1
    - name: etcd-operator 2
      image: quay.io/etcd-
operator/operator@sha256:d134a9865524c29fcf75bbc4469013bc38d8a15cb5f41acfddb6
b9e492f556e4 3
    - name: etcd-image
      image: quay.io/etcd-
operator/etcd@sha256:13348c15263bd8838ec1d5fc4550ede9860fcbb0f843e48cbccec07
810eebb68
...
```

1. **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。
2. イメージの一意の識別子を指定します。
3. 各イメージを、イメージタグでなく、ダイジェスト (SHA) で指定します。

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

```
spec:
  install:
    spec:
      deployments:
      - name: etcd-operator-v3.1.1
        spec:
          replicas: 1
          selector:
            matchLabels:
              name: etcd-operator
          strategy:
            type: Recreate
          template:
            metadata:
              labels:
                name: etcd-operator
            spec:
              containers:
              - args:
                - /opt/etcd/bin/etcd_operator_run.sh
                env:
                - name: WATCH_NAMESPACE
                  valueFrom:
                    fieldRef:
                      fieldPath: metadata.annotations['olm.targetNamespaces']
                - name: ETCD_OPERATOR_DEFAULT_ETCD_IMAGE (1)
                  value: quay.io/etcd-
operator/etcd@sha256:13348c15263bd8838ec1d5fc4550ede9860fcbb0f843e48cbccec07
810eebb68 (2)
                - name: ETCD_LOG_LEVEL
                  value: INFO
                image: quay.io/etcd-
operator/operator@sha256:d134a9865524c29fcf75bbc4469013bc38d8a15cb5f41acfddb6
b9e492f556e4 (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 コンテナーイメージを参照します。

注記

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

2. **disconnected** アノテーションを追加します。これは、Operator が非接続環境で機能することを示します。

```
metadata:
  annotations:
    operators.openshift.io/infrastructure-features: '["disconnected"]'
```

Operator は、このインフラストラクチャー機能によって OperatorHub でフィルターされます。

5.6.6. 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 以外のバリアントをサポートする場合、サポートされるバリアントを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

デフォルトチャネルのチャネルヘッドにあるラベルのみが、パッケージマニフェストをラベルでフィルターする場合に考慮されます。たとえば、デフォルト以外のチャネルで Operator の追加アーキテクチャーを提供することは可能ですが、そのアーキテクチャーは **PackageManifest** API でのフィルターには使用できません。

CSV に **os** ラベルが含まれていない場合、これはデフォルトで以下の Linux サポートラベルが設定されているかのように処理されます。

```
labels:
    operatorframework.io/os.linux: supported
```

CSV に **arch** ラベルが含まれていない場合、これはデフォルトで以下の AMD64 サポートラベルが設定されているかのように処理されます。

```
labels:
    operatorframework.io/arch.amd64: supported
```

Operator が複数のノードアーキテクチャーまたはオペレーティングシステムをサポートする場合、複数のラベルを追加することもできます。

前提条件

- CSV を含む Operator プロジェクト
- 複数のアーキテクチャーおよびオペレーティングシステムの一覧表示をサポートするには、CSV で参照される Operator イメージはマニフェスト一覧イメージである必要があります。
- Operator がネットワークが制限された環境または非接続環境で適切に機能できるようにするには、参照されるイメージは、タグではなくダイジェスト (SHA) を使用して指定される必要もあります。

手順

- Operator がサポートするサポートされるアーキテクチャーおよびオペレーティングシステムのそれぞれについて CSV の **metadata.labels** にラベルを追加します。

```
labels:
  operatorframework.io/arch.s390x: supported
  operatorframework.io/os.zos: supported
  operatorframework.io/os.linux: supported (1)
  operatorframework.io/arch.amd64: supported (2)
```

(1) (2) 新規のアーキテクチャーまたはオペレーティングシステムを追加したら、デフォルトの **os.linux** および **arch.amd64** バリアントも明示的に組み込む必要があります。

関連情報

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

5.6.6.1. 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

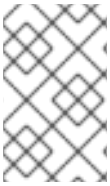
以下の文字列は、複数のアーキテクチャーおよびオペレーティングシステムをサポートする Operator のラベル付けまたはフィルター時に OpenShift Container Platform の Operator Lifecycle Manager (OLM) でサポートされます。

表5.8 OpenShift Container Platform でサポートされるアーキテクチャー

アーキテクチャー	文字列
AMD64	**amd64**
64 ビット PowerPC little-endian	**ppc64le**
IBM Z	**s390x**

表5.9 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。

5.6.7. 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> 1
```

1 提案された namespace を設定します。

5.6.8. Webhook の定義

Webhook により、リソースがオブジェクトストアに保存され、Operator コントローラーによって処理される前に、Operator の作成者はリソースのインターセプト、変更、許可、および拒否を実行するこ

とができます。Operator Lifecycle Manager (OLM) は、Operator と共に提供される際にこれらの Webhook のライフサイクルを管理できます。

Operator のクラスターサービスバージョン (CSV) リソースには、以下のタイプの Webhook を定義するために **webhookdefinitions** セクションを含めることができます。

- 受付 Webhook (検証および変更用)
- 変換 Webhook

手順

- **webhookdefinitions** セクションを Operator の CSV の **spec** セクションに追加し、**type** として **ValidatingAdmissionWebhook**、 **MutatingAdmissionWebhook**、または **ConversionWebhook** を使用して Webhook 定義を追加します。以下の例には、3 つのタイプの Webhook がすべて含まれます。

  Webhook が含まれる CSV

```
apiVersion: operators.coreos.com/v1alpha1
kind: ClusterServiceVersion
metadata:
  name: webhook-operator.v0.0.1
spec:
  customresourcedefinitions:
    owned:
    - kind: WebhookTest
      name: webhooktests.webhook.operators.coreos.io 1
      version: v1
  install:
    spec:
      deployments:
      - name: webhook-operator-webhook
        ...
        ...
        ...
    strategy: deployment
  installModes:
  - supported: false
    type: OwnNamespace
  - supported: false
    type: SingleNamespace
  - supported: false
    type: MultiNamespace
  - supported: true
    type: AllNamespaces
  webhookdefinitions:
  - type: ValidatingAdmissionWebhook 2
    admissionReviewVersions:
    - v1beta1
    - v1
    containerPort: 443
    targetPort: 4343
    deploymentName: webhook-operator-webhook
    failurePolicy: Fail
    generateName: vwebhooktest.kb.io
```

```
    rules:
    - apiGroups:
      - webhook.operators.coreos.io
      apiVersions:
      - v1
      operations:
      - CREATE
      - UPDATE
      resources:
      - webhooktests
    sideEffects: None
    webhookPath: /validate-webhook-operators-coreos-io-v1-webhooktest
  - type: MutatingAdmissionWebhook (3)
    admissionReviewVersions:
    - v1beta1
    - v1
    containerPort: 443
    targetPort: 4343
    deploymentName: webhook-operator-webhook
    failurePolicy: Fail
    generateName: mwebhooktest.kb.io
    rules:
    - apiGroups:
      - webhook.operators.coreos.io
      apiVersions:
      - v1
      operations:
      - CREATE
      - UPDATE
      resources:
      - webhooktests
    sideEffects: None
    webhookPath: /mutate-webhook-operators-coreos-io-v1-webhooktest
  - type: ConversionWebhook (4)
    admissionReviewVersions:
    - v1beta1
    - v1
    containerPort: 443
    targetPort: 4343
    deploymentName: webhook-operator-webhook
    generateName: cwebhooktest.kb.io
    sideEffects: None
    webhookPath: /convert
    conversionCRDs:
    - webhooktests.webhook.operators.coreos.io (5)
...
```

(1) 変換 Webhook がターゲットとする CRD がここに存在している必要があります。

(2) 検証用の受付 Webhook。

(3) 変更用の受付 Webhook。

(4) 変換 Webhook。

(5) 各 CRD の **spec.PreserveUnknownFields** プロパティーは **false** または **nil** に設定される必要があります。

必要があります。

関連情報

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

5.6.8.1. OLM についての Webhook の考慮事項

Operator Lifecycle Manager (OLM) を使用して Webhook で Operator をデプロイする場合、以下を定義する必要があります。

- **type** フィールドは **ValidatingAdmissionWebhook**、**MutatingAdmissionWebhook**、または **ConversionWebhook** のいずれかに設定する必要があります。そうでないと、CSV は失敗フェーズに置かれます。
- CSV には、**webhookdefinition** の **deploymentName** フィールドに指定される値に等しい名前のデプロイメントが含まれる必要があります。

Webhook が作成されると、OLM は、Operator がデプロイされる Operator グループに一致する namespace でのみ Webhook が機能するようにします。

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

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

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

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

変換 Webhook の制約
OLM は、変換 Webhook 定義が以下の制約に準拠しない場合に、失敗フェーズに CSV を配置します。

- 変換 Webhook と特長とする CSV は、**AllNamespaces** インストールモードのみをサポートできます。
- 変換 Webhook がターゲットとする CRD では、**spec.preserveUnknownFields** フィールドを **false** または **nil** に設定する必要があります。
- CSV で定義される変換 Webhook は所有 CRD をターゲットにする必要があります。
- 特定の CRD には、クラスター全体で 1 つの変換 Webhook のみを使用できます。

5.6.9. カスタムリソース定義 (CRD) について

Operator が使用できる以下の 2 つのタイプのカスタムリソース定義 (CRD) があります。1 つ目は Operator が所有する **所有** タイプと、もう 1 つは Operator が依存する **必須** タイプです。

5.6.9.1. 所有 CRD (Owned CRD)

Operator が所有するカスタムリソース定義 (CRD) は CSV の最も重要な部分です。これは Operator と必要な RBAC ルール間のリンク、依存関係の管理、および他の Kubernetes の概念を設定します。

Operator は通常、複数の CRD を使用して複数の概念を結び付けます (あるオブジェクトの最上位のデータベース設定と別のオブジェクトのレプリカセットの表現など)。それぞれは CSV ファイルに一覧表示される必要があります。

表5.10 所有 CRD フィールド

フィールド	説明	必須/オプション
Name	CRD のフルネーム。	必須
Version	オブジェクト API のバージョン。	必須
Kind	CRD の機械可読名。	必須
DisplayName	CRD 名の人間が判読できるバージョン (例: **MongoDB Standalone**)。	必須
説明	Operator がこの CRD を使用する方法についての短い説明、または CRD が提供する機能の説明。	必須
Group	この CRD が所属する API グループ (例: **database.example.com**)。	オプション

フィールド	説明	必須/オプション
Resources	CRD が 1 つ以上の Kubernetes オブジェクトのタイプを所有する。これらは、トラブルシューティングが必要になる可能性のあるオブジェクトや、データベースを公開するサービスまたは Ingress ルールなどのアプリケーションに接続する方法についてユーザーに知らせるために **resources** セクションに一覧表示されます。 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**	これらの記述子は、エンドユーザーにとって最も重要な Operator の入力および出力で UI にヒントを提供する手段になります。CRD にユーザーが指定する必要のあるシークレットまたは設定マップの名前が含まれる場合は、それをここに指定できます。これらのアイテムはリンクされ、互換性のある UI で強調表示されます。 記述子には、3 つの種類があります。 ● **SpecDescriptors**: オブジェクトの **spec** ブロックのフィールドへの参照。 ● **StatusDescriptors**: オブジェクトの **status** ブロックのフィールドへの参照。 ● **ActionDescriptors**: オブジェクトで実行できるアクションへの参照。 すべての記述子は以下のフィールドを受け入れます。 ● **DisplayName**: **Spec**、**Status**、または **Action** の人間が判読できる名前。 ● **Description**: **Spec**、**Status**、または **Action**、およびそれが Operator によって使用される方法についての短い説明。 ● **Path**: この記述子が記述するオブジェクトのフィールドのドットで区切られたパス。 ● **X-Descriptors**: この記述子が持つ機能および使用する UI コンポーネントを判別するために使用されます。OpenShift Container Platform の正規の React UI X-Descriptor の一覧については、**openshift/console** プロジェクトを参照してください。 記述子 一般についての詳細は、**openshift/console** プロジェクトも参照してください。	オプション

以下の例は、シークレットおよび設定マップ でユーザー入力を必要とし、サービス、ステートフルセット、Pod および設定マップのオーケストレーションを行う **MongoDB Standalone** CRD を示しています。

所有 CRD の例

```
    - displayName: MongoDB Standalone
      group: mongodb.com
      kind: MongoDbStandalone
      name: mongodbstandalones.mongodb.com
      resources:
        - kind: Service
          name: ''
          version: v1
        - kind: StatefulSet
          name: ''
          version: v1beta2
        - kind: Pod
          name: ''
          version: v1
        - kind: ConfigMap
          name: ''
          version: v1
      specDescriptors:
        - description: Credentials for Ops Manager or Cloud Manager.
          displayName: Credentials
          path: credentials
          x-descriptors:
            - 'urn:alm:descriptor:com.tectonic.ui:selector:core:v1:Secret'
        - description: Project this deployment belongs to.
          displayName: Project
          path: project
          x-descriptors:
            - 'urn:alm:descriptor:com.tectonic.ui:selector:core:v1:ConfigMap'
        - description: MongoDB version to be installed.
          displayName: Version
          path: version
          x-descriptors:
            - 'urn:alm:descriptor:com.tectonic.ui:label'
      statusDescriptors:
        - description: The status of each of the pods for the MongoDB cluster.
          displayName: Pod Status
          path: pods
          x-descriptors:
            - 'urn:alm:descriptor:com.tectonic.ui:podStatuses'
      version: v1
      description: >-
        MongoDB Deployment consisting of only one host. No replication of
        data.
```

5.6.9.2. 必須 CRD (Required CRD)

他の必須 CRD の使用は完全にオプションであり、これらは個別 Operator のスコープを縮小し、エンドツーエンドのユースケースに対応するために複数の Operator を一度に作成するために使用できます。

一例として、Operator がアプリケーションをセットアップし、分散ロックに使用する (etcd Operator からの) etcd クラスター、およびデータストレージ用に (Postgres Operator からの) Postgres データベースをインストールする場合があります。

Operator Lifecycle Manager (OLM) は、これらの要件を満たすためにクラスター内の利用可能な CRD および Operator に対してチェックを行います。適切なバージョンが見つかると、Operator は必要な namespace 内で起動し、サービスアカウントが各 Operator が必要な Kubernetes リソースを作成し、監視し、変更できるようにするために作成されます。

表5.11 必須 CRD フィールド

フィールド	説明	必須/オプション
Name	必要な CRD のフルネーム。	必須
Version	オブジェクト API のバージョン。	必須
Kind	Kubernetes オブジェクトの種類。	必須
DisplayName	CRD の人間による可読可能なバージョン。	必須
説明	大規模なアーキテクチャーにおけるコンポーネントの位置付けについてのサマリー。	必須

必須 CRD の例

```
required:
- name: etcdclusters.etcd.database.coreos.com
  version: v1beta2
  kind: EtcdCluster
  displayName: etcd Cluster
  description: Represents a cluster of etcd nodes.
```

5.6.9.3. CRD のアップグレード

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

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

5.6.9.3.1. 新規 CRD バージョンの追加

手順

CRD の新規バージョンを Operator に追加するには、以下を実行します。

1. CSV の **versions** セクションに CRD リソースの新規エントリーを追加します。
たとえば、現在の CRD にバージョン **v1alpha1** があり、新規バージョン **v1beta1** を追加し、これを新規のストレージバージョンとしてマークをする場合に、**v1beta1** の新規エントリーを追加します。

```
versions:
```

```
  - name: v1alpha1
    served: true
    storage: false
  - name: v1beta1 (1)
    served: true
    storage: true
```

(1) 新規エントリー。

2. CSV が新規バージョンを使用する場合、CSV の **owned** セクションの CRD の参照バージョンが更新されていることを確認します。

```
customresourcedefinitions:
  owned:
  - name: cluster.example.com
    version: v1beta1 (1)
    kind: cluster
    displayName: Cluster
```

(1) **version** を更新します。

3. 更新された CRD および CSV をバンドルにプッシュします。

5.6.9.3.2. CRD バージョンの非推奨または削除

Operator Lifecycle Manager (OLM) では、カスタムリソース定義 (CRD) の提供バージョンをすぐに削除できません。その代わりに、CRD の非推奨バージョンを CRD の **served** フィールドを **false** に設定して無効にする必要があります。その後に、無効にされたバージョンではないバージョンを後続の CRD アップグレードで削除できます。

手順

特定バージョンの CRD を非推奨にし、削除するには、以下を実行します。

1. 非推奨バージョンを non-serving (無効にされたバージョン) とマークして、このバージョンが使用されなくなり、後続のアップグレードで削除される可能性があることを示します。以下に例を示します。

```
versions:
  - name: v1alpha1
    served: false (1)
    storage: true
```

(1) **false** に設定します。

2. 非推奨となるバージョンが現在 **storage** バージョンの場合、**storage** バージョンを有効にされたバージョンに切り替えます。以下に例を示します。

```
versions:
  - name: v1alpha1
    served: false
    storage: false (1)
```

```
  - name: v1beta1
    served: true
    storage: true 2
```

1 2 **storage** フィールドを適宜更新します。

注記

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

3. 上記の変更内容で CRD をアップグレードします。
4. 後続のアップグレードサイクルでは、無効にされたバージョンを CRD から完全に削除できます。以下に例を示します。

```
versions:
  - name: v1beta1
    served: true
    storage: true
```

5. 該当バージョンが CRD から削除される場合、CSV の **owned** セクションにある CRD の参照バージョンも更新されていることを確認します。

5.6.9.4. CRD テンプレート

Operator のユーザーは、どのオプションが必須またはオプションであるかを認識している必要があります。**alm-examples** という名前のアノテーションとして、設定の最小セットを使用して、各カスタムリソース定義 (CRD) のテンプレートを提供できます。互換性のある UI は、ユーザーがさらにカスタマイズできるようにこのテンプレートの事前入力を行います。

アノテーションは、Kind の一覧で設定されます (例: CRD 名および Kubernetes オブジェクトの対応する **metadata** および **spec**)。

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

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

5.6.9.5. 内部オブジェクトの非表示

Operator がタスクを実行するためにカスタムリソース定義 (CRD) を内部で使用する方法は一般的な方法です。これらのオブジェクトはユーザーが操作することが意図されていません。オブジェクトの操作により Operator のユーザーにとって混乱を生じさせる可能性があります。たとえば、データベース Operator には、ユーザーが **replication: true** で Database オブジェクトを作成する際に常に作成される **Replication** CRD が含まれる場合があります。

Operator の作成者は、**operators.operatorframework.io/internal-objects** アノテーションを Operator のクラスターサービスバージョン (CSV) に追加して、ユーザー操作を目的としていないユーザーインターフェイスの CRD を非表示にすることができます。

手順

1. CRD のいずれかに internal のマークを付ける前に、アプリケーションの管理に必要となる可能性のあるデバッグ情報または設定が CR のステータスまたは **spec** ブロックに反映されていることを確認してください (使用する Opearator に該当する場合)。

2. **operators.operatorframework.io/internal-objects** アノテーションを Operator の CSV に追加し、ユーザーインターフェイスで非表示にする内部オブジェクトを指定します。

   内部オブジェクトのトアノテーション

```
apiVersion: operators.coreos.com/v1alpha1
kind: ClusterServiceVersion
metadata:
  name: my-operator-v1.2.3
  annotations:
    operators.operatorframework.io/internal-objects:
'["my.internal.crd1.io","my.internal.crd2.io"]' (1)
...
```

   (1) 内部 CRD を文字列の配列として設定します。

5.6.9.6. 必要なカスタムリソースの初期化

Operator では、ユーザーが Operator が完全に機能する前にカスタムリソースをインスタンス化する必要がある場合があります。ただし、ユーザーが必要な内容やリソースの定義方法を判断することが困難な場合があります。

Operator 開発者は、**operatorframework.io/initialization-resource** アノテーションをクラスターサービスバージョン (CSV) に追加して Operator のインストール時に作成する必要のある単一の必要なカスタムリソースを指定できます。アノテーションには、インストール時にリソースを初期化するために必要な完全な YAML 定義が含まれるテンプレートが含まれている必要があります。

このアノテーションが定義されている場合、OpenShift Container Platform Web コンソールから Operator をインストールすると、ユーザーには CSV で提供されるテンプレートを使用してリソースを作成することを求めるプロンプトが出されます。

手順

- **operatorframework.io/initialization-resource** アノテーションを Operator の CSV に追加し、必要なカスタムリソースを指定します。たとえば、以下のアノテーションでは **StorageCluster** リソースの作成が必要であり、これは完全な YAML 定義を提供します。

  初期化リソースアノテーション

```
apiVersion: operators.coreos.com/v1alpha1
kind: ClusterServiceVersion
metadata:
  name: my-operator-v1.2.3
  annotations:
    operatorframework.io/initialization-resource: |-
        {
            "apiVersion": "ocs.openshift.io/v1",
            "kind": "StorageCluster",
            "metadata": {
                "name": "example-storagecluster"
            },
            "spec": {
                "manageNodes": false,
                "monPVCTemplate": {
                    "spec": {
                        "accessModes": [
                            "ReadWriteOnce"
                        ],
                        "resources": {
                            "requests": {
                                "storage": "10Gi"
                            }
                        },
                        "storageClassName": "gp2"
                    }
                },
                "storageDeviceSets": [
                    {
                        "count": 3,
                        "dataPVCTemplate": {
                            "spec": {
                                "accessModes": [
                                    "ReadWriteOnce"
                                ],
                                "resources": {
                                    "requests": {
                                        "storage": "1Ti"
                                    }
                                },
                                "storageClassName": "gp2",
                                "volumeMode": "Block"
                            }
                        },
                        "name": "example-deviceset",
                        "placement": {},
                        "portable": true,
                        "resources": {}
                    }
                ]
            }
        }
...
```

5.6.10. API サービスについて

CRD の場合のように、Operator が使用できる API サービスの 2 つのタイプ (**所有 (owned)** および **必須 (required)**) があります。

5.6.10.1. 所有 API サービス

CSV が API サービスを所有する場合、CSV は API サービスおよびこれが提供する group/version/kind (GVK) をサポートする拡張 **api-server** のデプロイメントを記述します。

API サービスはこれが提供する group/version によって一意に識別され、提供することが予想される複数の種類を示すために複数回一覧表示できます。

表5.12 所有 API サービスフィールド

フィールド	説明	必須/オプション
Group	API サービスが提供するグループ (**database.example.com** など)。	必須
Version	API サービスのバージョン (**v1alpha1** など)。	必須
Kind	API サービスが提供することが予想される種類。	必須
Name	指定された API サービスの複数形の名前	必須
DeploymentName	API サービスに対応する CSV で定義されるデプロイメントの名前 (所有 API サービスに必要)。CSV の保留フェーズに、OLM Operator は CSV の **InstallStrategy** で一致する名前を持つ **Deployment** 仕様を検索し、これが見つからない場合には、CSV をインストールの準備完了フェーズに移行しません。	必須
DisplayName	API サービス名の人間が判読できるバージョン (例: **MongoDB Standalone**)。	必須
説明	Operator がこの API サービスを使用する方法についての短い説明、または API サービスが提供する機能の説明。	必須
Resources	API サービスは 1 つ以上の Kubernetes オブジェクトのタイプを所有します。これらは、トラブルシューティングが必要になる可能性のあるオブジェクトや、データベースを公開するサービスまたは Ingress ルールなどのアプリケーションに接続する方法についてユーザーに知らせるためにリソースセクションに一覧表示されます。 この場合、オーケストレーションするすべての一覧ではなく、重要なオブジェクトのみを一覧表示することが推奨されます。たとえば、ユーザーが変更できない内部状態を保存する設定マップを一覧表示しないでください。	オプション

フィールド	説明	必須/オプション
SpecDescriptors、**StatusDescriptors**、および **ActionDescriptors**	所有 CRD と基本的に同じです。	オプション

5.6.10.1.1. API サービスリソースの作成

Operator Lifecycle Manager (OLM) はそれぞれ固有の所有 API サービスについてサービスおよび API サービスリソースを作成するか、またはこれらを置き換えます。

- サービス Pod セレクターは API サービスの記述の **DeploymentName** フィールドに一致する CSV デプロイメントからコピーされます。
- 新規の CA キー/証明書ペアが各インストールについて生成され、base64 でエンコードされた CA バンドルがそれぞれの API サービスリソースに組み込まれます。

5.6.10.1.2. API サービス提供証明書

OLM は、所有 API サービスがインストールされるたびに、提供するキー/証明書のペアの生成を処理します。提供証明書には、生成される **Service** リソースのホスト名が含まれる一般名 (CN) が含まれ、これは対応する API サービスリソースに組み込まれた CA バンドルのプライベートキーによって署名されます。

証明書は、デプロイメント namespace の **kubernetes.io/tls** タイプのシークレットとして保存され、**apiservice-cert** という名前のボリュームは、 API サービスの記述の **DeploymentName** フィールドに一致する CSV のデプロイメントのボリュームセクションに自動的に追加されます。

存在していない場合、一致する名前を持つボリュームマウントもそのデプロイメントのすべてのコンテナーに追加されます。これにより、ユーザーは、カスタムパスの要件に対応するために、予想される名前のボリュームマウントを定義できます。生成されるボリュームマウントのパスは **/apiserver.local.config/certificates** にデフォルト設定され、同じパスの既存のボリュームマウントが置き換えられます。

5.6.10.2. 必要な API サービス

OLM は、必要なすべての CSV に利用可能な API サービスがあり、すべての予想される GVK がインストールの試行前に検出可能であることを確認します。これにより、CSV は所有しない API サービスによって提供される特定の種類に依存できます。

表5.13 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

フィールド	説明	必須/オプション
DisplayName	API サービス名の人間が判読できるバージョン (例: **MongoDB Standalone**)。	必須
説明	Operator がこの API サービスを使用する方法についての短い説明、または API サービスが提供する機能の説明。	必須

5.7. バンドルイメージの使用

Bundle Format を使用し、Operator SDK で Operator をパッケージ化できます。

5.7.1. 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 0.19.4
- **podman** version 1.9.3+
- Operator プロジェクトが Operator SDK を使用して生成されている
- Docker v2-2 をサポートするレジストリーへのアクセス

重要

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

手順

1. 以下の **make** コマンドを Operator プロジェクトディレクトリーで実行し、Operator イメージをビルドし、プッシュします。以下の手順の **IMG** 引数を変更して、アクセス可能なリポジトリーを参照します。Quay.io などのリポジトリーサイトにコンテナーを保存するためのアカウントを取得できます。

   a. イメージをビルドします。

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

注記

Operator の SDK によって生成される Dockerfile は、**go build** について **GOARCH=amd64** を明示的に参照します。これは、AMD64 アーキテクチャー以外の場合は **GOARCH=$TARGETARCH** に修正できます。Docker は、**-platform** で指定された値に環境変数を自動的に設定します。Buildah では、そのために **-build-arg** を使用する必要があります。詳細は、Multiple Architectures を参照してください。

b. イメージをリポジトリーにプッシュします。

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

2. プッシュした Operator イメージ名およびタグに **IMG** URL を設定し、 **Makefile** を更新します。

```
$ # Image URL to use all building/pushing image targets
IMG ?= <registry>/<user>/<operator_image_name>:<tag>
```

この値は、後続の操作に使用されます。

3. Operator SDK **generate bundle** および **bundle validate** のサブコマンドを含む複数のコマンドを呼び出す **make bundle** コマンドを実行し、Operator バンドルマニフェストを作成します。

```
$ make bundle
```

Operator のバンドルマニフェストは、アプリケーションを表示し、作成し、管理する方法を説明します。**make bundle** コマンドは、以下のファイルおよびディレクトリーを Operator プロジェクトに作成します。

- **ClusterServiceVersion** オブジェクトを含む **bundle/manifests** という名前のバンドルマニフェストディレクトリー
- **bundle/metadata** という名前のバンドルメタデータディレクトリー
- **config/crd** ディレクトリー内のすべてのカスタムリソース定義 (CRD)
- Dockerfile **bundle.Dockerfile**

続いて、これらのファイルは **operator-sdk bundle validate** を使用して自動的に検証され、ディスク上のバンドル表現が正しいことを確認します。

4. 以下のコマンドを実行し、バンドルイメージをビルドしてプッシュします。OLM は、1 つ以上のバンドルイメージを参照するインデックスイメージを使用して Operator バンドルを使用します。

a. バンドルイメージをビルドします。イメージをプッシュしようとするレジストリー、ユーザー namespace、およびイメージタグの詳細で **BUNDLE_IMG** を設定します。

```
$ make bundle-build BUNDLE_IMG=<registry>/<user>/<bundle_image_name>:<tag>
```

b. バンドルイメージをプッシュします。

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

5.7.2. 関連情報

- Bundle Format の詳細は、Operator Framework パッケージ形式 を参照してください。
- **opm** コマンドを使用してバンドルイメージをインデックスイメージに追加する方法の詳細は、カスタムカタログの管理 を参照してください。
- インストールされた Operator のアップグレードの仕組みについての詳細は、Operator Lifecycle Manager ワークフロー を参照してください。

5.8. 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 4.6 は Operator SDK v0.19.4 をサポートします。

5.8.1. 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、その後に 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ーコンテナーを 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。

5.8.2. 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。

5.8.2.1. 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: 1
        cr-manifest:
          - "deploy/crds/cache.example.com_v1alpha1_memcached_cr.yaml"
          - "deploy/crds/cache.example.com_v1alpha1_memcachedrs_cr.yaml"
    - olm: 2
        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"
```

1. 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。
2. 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。

5.8.2.2. 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5.14 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レクター。
--list, **-L**	bool	**true** の場合、セレ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。

5.8.2.3. 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5.15 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。
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。

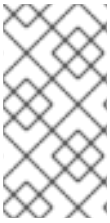
5.8.2.3.1. 基本的なプラグインおよび OLM プラグイン

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5.16 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)。

オプション	型	説明
crds-dir	string	クラスターにデプロイする必要のある CRD が含まれるディレクトリーへのパス。
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。

5.8.3. 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)'
```

5.8.3.1. 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5.17 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ーのログを読み取り、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。	**writingintocrshaseffecttest**

5.8.3.2. 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5.18 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**
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**

テスト	説明	短縮名
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。	**statusdescriptorstest**

追加リソース

- 所有 CRD (Owned CRD)

5.8.4. 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ー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。

5.8.5. 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 にプロキシーコンテナーが必要になります。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** を含むプロキシー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ーの 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":"'<namespace>
'"}' | base64 -w 0 (1)
```

(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ーに挿入します。

```
spec:
  install:
    spec:
      deployments:
      - name: memcached-operator
        spec:
          ...
          template:
            ...
            spec:
              containers:
              - name: container1
```

```
          ...
          volumeMounts:
          - name: scorecard-kubeconfig 1
            mountPath: /scorecard-secret
          env:
          - name: KUBECONFIG 2
            value: /scorecard-secret/config
        - name: container2 3
          ...
```

1 スコアカードの **kubeconfig** ボリュームマウント。

2 スコアカードの **kubeconfig** 環境変数。

3 これと同じ手順を他のコンテナーについても繰り返します。

c. スコアカードプロキシーコンテナーを 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 1
                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ーコンテナー。

- **自動的な方法:**
  community-operators リポジトリーには、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. プロキシーコンテナーを挿入します。

```
$ insert_proxy_container "<csv_file>" "quay.io/operator-framework/scorecard-
proxy:master"
```

2. プロキシーコンテナーの挿入後に、**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
```

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

5.9.1. Prometheus Operator のサポート

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

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

5.9.2. 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。

5.9.2.1. メトリクスポートの変更

Operator の作成者は、メトリクスが公開されるポートを変更できます。

前提条件

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

手順

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

```
var metricsPort int32 = 8383
```

5.9.3. 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。

5.9.3.1. サービスモニターの作成

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

前提条件

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

手順

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

```
import(
   "k8s.io/api/core/v1"
   "github.com/operator-framework/operator-sdk/pkg/metrics"
```

```
    "machine.openshift.io/controller-runtime/pkg/client/config"
)
func main() {

    ...
    // Populate below with the Service(s) for which you want to create ServiceMonitors.
    services := []*v1.Service{}
    // Create one ServiceMonitor per application per namespace.
    // Change the below value to name of the Namespace you want the ServiceMonitor to be
created in.
    ns := "default"
    // restConfig is used for talking to the Kubernetes apiserver
    restConfig := config.GetConfig()

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

5.10. 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。

5.10.1. Operator リーダー選出の例

次の例では、Operator のリーダー選出オプション (Leader-for-life と Leader-with-lease) 2 つの使用方法を説明します。

5.10.1.1. Leader-for-life 選択

Leader-for-life 選択の実装の場合、**leader.Become()** の呼び出しは、**memcached-operator-lock** という名前の設定マップを作成して、リーダー選択までの再試行中に Operator をブロックします。

```
import (
  ...
  "github.com/operator-framework/operator-sdk/pkg/leader"
)

func main() {
  ...
  err = leader.Become(context.TODO(), "memcached-operator-lock")
  if err != nil {
    log.Error(err, "Failed to retry for leader lock")
    os.Exit(1)
  }
  ...
}
```

Operator がクラスター内で実行されていない場合、 **leader.Become()** はエラーなしに返し、Operator の名前を検出できないことからリーダー選択をスキップします。

5.10.1.2. Leader-with-lease 選択

Leader-with-lease 実装は、リーダー選択について Manager オプション を使用して有効にできます。

```
import (
  ...
  "sigs.k8s.io/controller-runtime/pkg/manager"
)

func main() {
  ...
  opts := manager.Options{
    ...
    LeaderElection: true,
    LeaderElectionID: "memcached-operator-lock"
  }
  mgr, err := manager.New(cfg, opts)
  ...
}
```

Operator がクラスターで実行されていない場合、Manager はリーダー選択用の設定マップ を作成するための 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。

5.11. 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>]
```

5.11.1. alpha

operator-sdk alpha コマンドは、alpha サブコマンドを実行するために使用されます。

5.11.1.1. scorecard

alpha scorecard サブコマンドは、スコアカードツールを実行して Operator バンドルを検証し、改善のための提案を提供します。このコマンドは、バンドルイメージまたはマニフェストおよびメタデータを含むディレクトリーのいずれかの引数を取ります。引数がイメージタグを保持する場合は、イメージはリモートに存在する必要があります。

表5.19 scorecard フラグ

フラグ	説明
-c, **--config** (文字列)	スコアカード設定ファイルへのパス。
-h、**--help**	**scorecard** コマンドのヘルプ出力。
--kubeconfig (文字列)	**kubeconfig** ファイルへのパス。
-L, **--list**	実行可能なテストを一覧表示します。
-n, **--namespace** (文字列)	テストイメージを実行する namespace。デフォルト: **default**。
-o, **--output** (文字列)	結果の出力形式。利用可能な値は **text** と **json** です。デフォルト: **text**。
-l, **--selector** (文字列)	実行されるテストを決定するラベルセレクター。
-s, **--service-account** (文字列)	テストに使用するサービスアカウント。デフォルト: **default**。
-x, **--skip-cleanup**	テストの実行後にリソースクリーンアップを無効にします。
-w, **--wait-time <duration>**	テストが完了するのを待つ秒数 (例: **35s**)。デフォルト: **30s**

5.11.2. 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ー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5.20 build 引数

引数	説明
<image>	ビルドされるコンテナーイメージ (例: **quay.io/example/operator:v0.0.1**)。

表5.21 **build** フラグ

フラグ	説明
--go-build-args (文字列)	追加の Go ビルド引数。
--image-build-args (文字列)	1 つの文字列としての追加のイメージビルド引数。
--image-builder (文字列)	OCI イメージを構築するためのツール。利用可能なオプションは、**docker**、**podman**、または **buildah** です。デフォルト: **docker**。
-h, --help	使用方法についてのヘルプの出力。

5.11.3. bundle

operator-sdk bundle コマンドは Operator バンドルメタデータを管理します。

5.11.3.1. validate

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

表5.22 **bundle validate** フラグ

フラグ	説明
-h, **--help**	**bundle validate** サブコマンドのヘルプ出力。
-b、**--image-builder** (文字列)	バンドルイメージをプルおよび展開するためのツール。バンドルイメージを検証する場合にのみ使用されます。利用可能なオプションは **docker**、**podman**、または **none** です。デフォルト: **docker**。

5.11.4. cleanup

operator-sdk cleanup コマンドは、**run** コマンドでデプロイされた Operator 用に作成されたリソースを破棄し、削除します。

5.11.4.1. packagemanifests

cleanup packagemanifests サブコマンドは、**run packagemanifests** コマンドを使用して OLM でデプロイされた Operator を破棄します。

表5.23 **packagemanifests** 引数

引数	説明
--include (文字列)	ロールオブジェクトやサブスクリプションオブジェクトなどの Kubernetes リソースマニフェストへのファイルパス。これらは、**run** または **cleanup** によって生成されたデフォルトを補足またはオーバーライドします。

引数	説明
--install-mode (文字列)	**OperatorGroup** は、指定された **InstallMode** で作成されます。形式: **InstallModeType[=ns1,ns2[, …]]`**
--kubeconfig (文字列)	Kubernetes 設定ファイルへのファイルパス。デフォルト: **$KUBECONFIG** で指定された場所、または設定されていない場合はデフォルトのファイルルール。
--olm-namespace (文字列)	OLM がインストールされている namespace。デフォルト: **olm**。
--operator-namespace (文字列)	Operator リソースが作成される名前空間。名前空間は、クラスターに既に存在するか、**--include** に渡されるマニフェストで定義されている必要があります。
--operator-version	デプロイされる Operator のバージョン。
--timeout <duration>	コマンドが失敗する前に完了するのを待機する時間。デフォルト: **2m0s**。
-h, --help	使用方法についてのヘルプの出力。

5.11.5. completion

operator-sdk completion コマンドは、CLI コマンドをより迅速に、より容易に実行できるようにシェル補完を生成します。

表5.24 completion サブコマンド

サブコマンド	説明
bash	bash 補完を生成します。
zsh	zsh 補完を生成します。

表5.25 completion フラグ

フラグ	説明
-h, --help	使用方法についてのヘルプの出力。

以下に例を示します。

```
$ operator-sdk completion bash
```

出力例

```
# bash completion for operator-sdk                    -*- shell-script -*-
...
# ex: ts=4 sw=4 et filetype=sh
```

5.11.6. create

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

5.11.6.1. api

create api サブコマンドは Kubernetes API をスキャフォールディングします。サブコマンドは、 **init** コマンドで初期化されたプロジェクトで実行する必要があります。

表5.26 create api フラグ

フラグ	説明
-h, **--help**	**run bundle** サブコマンドのヘルプ出力。

5.11.6.2. webhook

create webhook サブコマンドは、API リソースの Webhook をスキャフォールディングします。サブコマンドは、**init** コマンドで初期化されたプロジェクトで実行する必要があります。

表5.27 create webhook フラグ

フラグ	説明
-h, **--help**	**run bundle** サブコマンドのヘルプ出力。

5.11.7. generate

operator-sdk generate コマンドは特定のジェネレーターを起動して、必要に応じてコードを生成します。

5.11.7.1. bundle

generate bundle サブコマンドは、Operator プロジェクトのバンドルマニフェスト、メタデータ、および **bundle.Dockerfile** ファイルのセットを生成します。

注記

通常は、最初に **generate kustomize manifests** サブコマンドを実行して、**generate bundle** サブコマンドで使用される入力された Kustomize ベースを生成します。ただし、初期化されたプロジェクトで **make bundle** コマンドを使用して、これらのコマンドの順次の実行を自動化できます。

表5.28 generate bundle フラグ

フラグ	説明
--channels (文字列)	バンドルが属するチャネルのコンマ区切りリスト。デフォルト値は **alpha** です。
--crds-dir (文字列)	**CustomResoureDefinition** マニフェストのルートディレクトリー。
--default-channel (文字列)	バンドルのデフォルトチャネル。
--deploy-dir (文字列)	デプロイメントや RBAC などの Operator マニフェストのルートディレクトリー。このディレクトリーは、**--input-dir** フラグに渡されるディレクトリーとは異なります。
-h, **--help**	**generate bundle** のヘルプ
--input-dir (文字列)	既存のバンドルを読み取るディレクトリー。このディレクトリーは、バンドル **manifests** ディレクトリーの親であり、**--deploy-dir** ディレクトリーとは異なります。
--kustomize-dir (文字列)	バンドルマニフェストの Kustomize ベースおよび **kustomization.yaml** ファイルを含むディレクトリー。デフォルトのパスは **config/manifests** です。
--manifests	バンドルマニフェストを生成します。
--metadata	バンドルメタデータと Dockerfile を生成します。
--operator-name (文字列)	バンドルの Operator の名前。
--output-dir (文字列)	バンドルを書き込むディレクトリー。
--overwrite	バンドルメタデータおよび Dockerfile を上書きします (ある場合)。デフォルト値は **true** です。
-q、**--quiet**	quiet モードで実行します。
--stdout	バンドルマニフェストを標準出力に書き込みます。
--version (文字列)	生成されたバンドルの Operator のセマンティックバージョン。新規バンドルを作成するか、または Operator をアップグレードする場合にのみ設定します。

5.11.7.2. kustomize

generate kustomize サブコマンドには、Operator の Kustomize データを生成するサブコマンドが含まれます。

5.11.7.2.1. manifests

generate kustomize manifests は Kustomize ベースを生成または再生成し、 **kustomization.yaml** ファ

イルを **config/manifests** ディレクトリーに生成または再生成します。これは、他の Operator SDK コマンドでバンドルマニフェストをビルドするために使用されます。このコマンドは、ベースがすでに存在しない場合や **--interactive=false** フラグが設定されていない場合に、デフォルトでマニフェストベースの重要なコンポーネントである UI メタデータを対話的に要求します。

表5.29 generate kustomize manifests フラグ

フラグ	説明
--apis-dir (文字列)	API タイプ定義のルートディレクトリー。
-h, **--help**	**generate kustomize manifests** のヘルプ。
--input-dir (文字列)	既存の Kustomize ファイルを含むディレクトリー。
--interactive	**false** に設定すると、Kustomize ベースが存在しない場合は、対話式コマンドプロンプトがカスタムメタデータを受け入れるように表示されます。
--operator-name (文字列)	Operator の名前
--output-dir (文字列)	Kustomize ファイルを書き込むディレクトリー。
-q、**--quiet**	quiet モードで実行します。

5.11.7.3. packagemanifests

generate packagemanifests サブコマンドの実行は、Operator をカタログに公開し、OLM またはその両方でデプロイするための最初のステップです。このコマンドは、Operator のバージョン管理されたディレクトリーおよびパッケージマニフェストファイルに一連のマニフェストを生成します。このコマンドによって消費される Kustomize ベースを再生成するには、まず **generate kustomize manifests** を実行する必要があります。

表5.30 generate packagemanifests フラグ

フラグ	説明
--channel (文字列)	生成されたパッケージのチャネル名。
--crds-dir (文字列)	カスタムリソース定義 (CRD) マニフェストのルートディレクトリー。
--default-channel	**--channel** に渡されたチャネルをパッケージマニフェストファイルのデフォルトチャネルとして使用します。
--deploy-dir (文字列)	デプロイや RBAC などの Operator マニフェストのルートディレクトリー (例: **deploy**)。このディレクトリーは、**--input-dir** に渡されるディレクトリーとは異なります。
--from-version (文字列)	アップグレード元の Operator のセマンティックバージョン。

フラグ	説明
-h、--help	**generate kustomize manifests** のヘルプ。
--input-dir (文字列)	既存のパッケージマニフェストを読み取るディレクトリー。このディレクトリーは、個々のバージョン管理されたパッケージディレクトリーの親であり、**--deploy-dir** とは異なります。
--kustomize-dir (文字列)	Kustomize ベースと **operator-framework** マニフェスト用の **kustomization.yaml** を含むディレクトリー。デフォルト: **config/manifests**。
--operator-name (文字列)	パッケージ化された Operator の名前。
--output-dir (文字列)	パッケージマニフェストを書き込むディレクトリー。
-q、--quiet	quiet モードで実行します。
--stdout	パッケージを **stdout** に書き込みます。
--update-crds	このパッケージのカスタムリソース定義 (CRD) マニフェストを更新します。Default: **true**
-v, --version (文字列)	パッケージ化された Operator のセマンティックバージョン。

5.11.8. init

operator-sdk init コマンドは Operator プロジェクトを初期化し、指定されたプラグインのデフォルトのプロジェクトディレクトリーレイアウトを生成または **スキャフォールド** します。

このコマンドは、以下のファイルを作成します。

- ボイラープレートライセンスファイル
- ドメインおよびリポジトリーを含む **PROJECT** ファイル
- プロジェクトをビルドする **Makefile**
- プロジェクト依存関係のある **go.mod** ファイル
- マニフェストをカスタマイズするための **kustomization.yaml** ファイル
- マネージャーマニフェストのイメージをカスタマイズするためのパッチファイル
- Prometheus メトリクスを有効にするためのパッチファイル
- 実行する **main.go** ファイル

表5.31 init フラグ

フラグ	説明
--help, -h	**init** コマンドのヘルプ出力。
--plugins (文字列)	プロジェクトを初期化するプラグインの名前およびオプションのバージョン。利用可能なプラグインは **ansible.sdk.operatorframework.io/v1**、**go.kubebuilder.io/v2**、**go.kubebuilder.io/v3**、および **helm.sdk.operatorframework.io/v1** です。
--project-version	プロジェクトのバージョン。使用できる値は **2** および **3-alpha** (デフォルト) です。

5.11.9. 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。

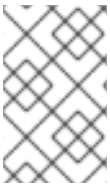
表5.32 new 引数

引数	説明
<project_name>	新規プロジェクトの名前。

表5.33 new フラグ

フラグ	説明
--api-version	**<group_name>/<version>** 形式の Kubernetes API バージョン (例: **app.example.com/v1alpha1**)。.
--crd-version	生成する CRD バージョン。デフォルト: **v1**。
--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。
--helm-chart <string>	既存の Helm チャートで Helm Operator を初期化します。**<url>**、**<repo>/<name>** 、またはローカルパス。
--helm-chart-repo <string>	要求される Helm チャートのチャートリポジトリー URL。
--helm-chart-version <string>	Helm チャートの特定バージョン。**helm** タイプのみ使用。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**)。

フラグ	説明
--skip-generation	deepcopy と OpenAPI コード、および OpenAPI CRD 仕様の生成をスキップします。
--type	初期化する Operator のタイプ: **ansible** または **helm**。



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
```

5.11.10. olm

operator-sdk olm コマンドは、クラスター内の Operator Lifecycle Manager (OLM) インストールを管理します。

5.11.10.1. install

olm install サブコマンドは、クラスターに OLM をインストールします。

表5.34 install 引数

引数	説明
--olm-namespace 文字列	OLM がインストールされている名前空間。デフォルト: **olm**。
--timeout <duration>	コマンドが失敗する前に完了するのを待機する時間。デフォルト: **2m0s**。
--version 文字列	インストールする OLM リソースのバージョン。デフォルト: **latest**。

引数	説明
-h, --help	使用方法についてのヘルプの出力。

5.11.10.2. status

olm status サブコマンドは、クラスター内の Operator Lifecycle Manager (OLM) インストールのステータスを取得します。

表5.35 status 引数

引数	説明
--olm-namespace 文字列	OLM のインストール元の namespace。デフォルト: **olm**。
--timeout <duration>	コマンドが失敗する前に完了するのを待機する時間。デフォルト: **2m0s**。
--version 文字列	クラスターにインストールされている OLM のバージョン。設定されていない場合、**operator-sdk** はバージョンの自動検出を試みます。
-h, --help	使用方法についてのヘルプの出力。

5.11.10.3. uninstall

olm uninstall サブコマンドは、クラスターから OLM をアンインストールします。

表5.36 uninstall 引数

引数	説明
--olm-namespace (文字列)	OLM をアンインストールする namespace。デフォルト: **olm**。
--timeout <duration>	コマンドが失敗する前に完了するのを待機する時間。デフォルト: **2m0s**。
--version (文字列)	アンインストールする OLM リソースのバージョン。
-h, --help	使用方法についてのヘルプの出力。

5.11.11. run

operator-sdk run コマンドは、さまざまな環境で Operator を起動できるオプションを提供します。

5.11.11.1. packagemanifests

run packagemanifests サブコマンドは、Operator Litecycle Manager (OLM) を使用して Operator のパッケージマニフェストをデプロイします。コマンド引数は、有効なパッケージマニフェストルートディレクトリー (**<project_root>/packagemanifests** など) に設定する必要があります。

表5.37 packagemanifests 引数

引数	説明
--include (文字列)	ロールオブジェクトやサブスクリプションオブジェクトなどの Kubernetes リソースマニフェストへのファイルパス。これらは、**run** または **cleanup** によって生成されたデフォルトを補足またはオーバーライドします。
--install-mode (文字列)	**OperatorGroup** は、指定された **InstallMode** で作成されます。形式: **InstallModeType[=ns1,ns2[, ...]]**。
--kubeconfig (文字列)	Kubernetes 設定ファイルへのファイルパス。デフォルト: **$KUBECONFIG** で指定された場所、または環境変数が設定されていない場合はデフォルトのファイルルール。
--olm-namespace (文字列)	OLM がインストールされている名前空間。デフォルト: **olm**。
--operator-namespace (文字列)	Operator リソースが作成される名前空間。名前空間は、クラスターに既に存在するか、**--include** に渡されるマニフェストで定義されている必要があります。
--operator-version (文字列)	デプロイする Operator のバージョン。
--timeout <duration>	コマンドが失敗する前に完了するのを待機する時間。デフォルト: **2m0s**。
-h, --help	使用方法についてのヘルプの出力。

5.12. 付録

5.12.1. Operator プロジェクトのスキャフォールディングレイアウト

operator-sdk CLI は、それぞれの Operator プロジェクトに多数のパッケージを生成します。以下のセクションには、生成される各ファイルおよびディレクトリーの基本的な要約が含まれます。

5.12.1.1. Ansible ベースのプロジェクト

operator-sdk new --type ansible コマンドを使用して生成される Ansible ベースの Operator プロジェクトには、以下のディレクトリーおよびファイルが含まれます。

ファイル/フォルダー	目的
molecule/	Ansible ロールのテストに使用されるファイルが含まれています。
roles/	プロジェクトの作成中に使用される Helm チャートが含まれています。

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。
requirements.yml	インストールが必要な Ansible コンテンツが含まれています。
watches.yaml	グループ、バージョン、種類、およびロールが含まれます。

5.12.1.2. 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、および Helm チャートの場所が含まれます。

第6章 RED HAT OPERATOR

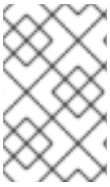
6.1. CLOUD CREDENTIAL OPERATOR

目的
Cloud Credential Operator (CCO) は、クラウドプロバイダーの認証情報を Kubernetes カスタムリソース定義 (CRD) として管理します。CCO は **credentialsRequest** カスタムリソース (CR) で同期し、OpenShift Container Platform コンポーネントが、クラスターの実行に必要な特定のパーミッションと共にクラウドプロバイダーの認証情報を要求できるようにします。

install-config.yaml ファイルで **credentialsMode** パラメーターに異なる値を設定すると、CCO は複数の異なるモードで動作するように設定できます。モードが指定されていない場合や、**credentialsMode** パラメーターが空の文字列 ("") に設定されている場合、CCO はデフォルトモードで動作します。

デフォルトの動作
複数のモードがサポートされるプラットフォーム (AWS、Azure、および GCP) の場合、CCO がデフォルトモードで動作する際に、これは指定される認証情報を動的にチェックし、**credentialsRequest** CR を処理するのに十分なモードを判別します。

デフォルトで、CCO は推奨される操作モードの mint モードに十分な認証情報があるかどうかを判別し、これらの認証情報を使用してクラスター内のコンポーネントの適切な認証情報を作成します。mint モードに十分な認証情報がない場合は、passthrough モードに十分な認証情報があるかどうかを判別します。passthrough モードに十分な認証情報がない場合、CCO は **credentialsRequest** CR を適切に処理できません。



注記

CCO は、Azure 認証情報が passthrough モードに十分であるかどうかを検証できません。Azure 認証情報が mint モードに不十分な場合、CCO は認証情報が passthrough モードには十分であることを前提として動作します。

インストール時に提供された認証情報が不十分であると判別される場合、インストールは失敗します。AWS では、インストーラーはプロセスの早期の段階で失敗し、どの必須パーミッションが欠落しているかを示します。他のプロバイダーは、エラーが発生するまでエラーの原因についての具体的な情報を提供しない場合があります。

認証情報が正常なインストールの後に変更され、CCO が新規の認証情報が不十分であると判別する場合に、CCO は新規の **credentialsRequest** CR に条件を追加し、認証情報が不十分であるためにそれらを処理できないことを示唆します。

不十分な認証情報についての問題を解決するために、適切なパーミッションで認証情報を指定します。エラーがインストール時に発生した場合は、再度インストールを試行します。新規の **credentialsRequest** CR 関連の問題については、CCO が再び CR の処理を試行するのを待機します。別の方法として、AWS、Azure、または GCP の IAM を手動で作成できます。詳細は、AWS、Azure、または GCP のインストールコンテンツの **IAM の手動作成** についてのセクションを参照してください。

モード
install-config.yaml ファイルに **credentialsMode** パラメーターの異なる値を設定することで、CCO を **mint**、**passthrough**、または **manual** モードで動作するように設定できます。これらのオプションにより、CCO がクラウド認証情報を使用してクラスターで **credentialsRequest** CR を処理し、CCO を組織のセキュリティー要件に対応するように設定する方法において透明性と柔軟性が提供されます。すべてのクラウドプロバイダーですべての CCO モードがサポートされているわけではありません。

Mint モード
mint モードは AWS、Azure、および GCP でサポートされます。

mint モードは、CCO が使用するデフォルトかつ推奨されるベストプラクティスの設定です。このモードでは、CCO は提供される管理レベルのクラウド認証情報を使用してクラスターを実行します。

インストール後に認証情報が削除されない場合、これは CCO によって保存され、クラスター内のコンポーネントの **credentialsRequest** CR を処理し、必要な特定のパーミッションのみでそれぞれの新規の認証情報を作成するために使用されます。mint モードでクラウド認証情報を継続的に調整することで、アップグレードなどの追加の認証情報またはパーミッションを必要とするアクションを続行できます。

mint モードがクラスターの **kube-system** namespace に管理者レベルの認証情報を保存する要件は、すべての組織のセキュリティー要件に適さない可能性があります。

mint モードで CCO を使用する場合、指定する認証情報が OpenShift Container Platform を実行し、インストールしているクラウドの各種要件を満たしていることを確認してください。指定される認証情報が mint モードで不十分な場合、CCO は IAM ユーザーを作成できません。

表6.1 mint モードの認証情報の要件

クラウド	パーミッション
AWS	• **iam:CreateAccessKey** • **iam:CreateUser** • **iam:DeleteAccessKey** • **iam:DeleteUser** • **iam:DeleteUserPolicy** • **iam:GetUser** • **iam:GetUserPolicy** • **iam:ListAccessKeys** • **iam:PutUserPolicy** • **iam:TagUser** • **iam:SimulatePrincipalPolicy**
Azure	**Azure アカウントの設定** コンテンツの **サービスプリンシパルの作成** のセクションで指定されるパーミッションを持つサービスプリンシパル

クラウド	パーミッション
GCP	• **resourcemanager.projects.get** • **serviceusage.services.list** • **iam.serviceAccountKeys.create** • **iam.serviceAccountKeys.delete** • **iam.serviceAccounts.create** • **iam.serviceAccounts.delete** • **iam.serviceAccounts.get** • **iam.roles.get** • **resourcemanager.projects.getIamPolicy** • **resourcemanager.projects.setIamPolicy**

管理者レベルの認証情報の削除またはローテーション機能を持つ mint モード

管理者レベルの認証情報の削除またはローテーション機能を持つ mint モードは、OpenShift Container Platform バージョン 4.4 以降で AWS でサポートされます。

このオプションでは、インストール時に管理者レベルの認証情報が必要ですが、認証情報はクラスターに永続的に保存されないため、有効期間を長くする必要はありません。

OpenShift Container Platform を mint モードでインストールした後に、管理者レベルの認証情報の Secret をクラスターから削除できます。Secret を削除する場合、CCO は、すべての **credentialsRequest** CR に必要なパーミッションがあるかどうかを確認することを可能にする、以前に作成された読み取り専用の認証情報を使用します。削除された後に、必要な場合は、関連付けられた認証情報は基礎となるクラウドで破棄できます。

管理者レベルの認証情報を必要とするものがアップグレード時などに変更される必要がない限り、管理者レベルの認証情報を必要はありません。それぞれのアップグレードの前に、認証情報の Secret を管理者レベルの認証情報と共に元に戻す必要があります。認証情報が存在しない場合は、アップグレードがブロックされる可能性があります。

passthrough モード

passthrough モードは、AWS、Azure、GCP、Red Hat OpenStack Platform (RHOSP)、Red Hat Virtualization (RHV)、および VMware vSphere でサポートされています。

passthrough モードでは、CCO はクラウド認証情報を要求するコンポーネントに、指定されたクラウド認証情報を渡します。認証情報には、インストールを実行し、クラスター内のコンポーネントで必要な操作を完了するためのパーミッションが必要ですが、認証情報を新たに作成する必要はありません。CCO は、passthrough モードで、追加の制限されたスコープの認証情報の作成を試行しません。

passthrough モードのパーミッション要件

passthrough モードで CCO を使用する場合、指定する認証情報が OpenShift Container Platform を実行し、インストールしているクラウドの各種要件を満たしていることを確認してください。CCO が **credentialsRequest** CR を作成するコンポーネントに渡す指定された認証情報が不十分な場合に、そのコンポーネントは、パーミッションがない API の呼び出しを試行する際にエラーを報告します。

AWS、Azure、または GCP の passthrough モードに指定する認証情報には、実行し、インストールしている OpenShift Container Platform のバージョンで必要なすべての **credentialsRequest** CR に必要なすべての必須パーミッションがなければなりません。クラウドプロバイダーに必要な **credentialsRequest** CR を見つけるには、AWS、Azure、または GCP のインストールコンテンツの **IAM の手動作成** についてのセクションを参照してください。

OpenShift Container Platform クラスターを Red Hat OpenStack Platform (RHOSP) にインストールするには、CCO では **member** ユーザーロールのパーミッションと共に認証情報が必要になります。

OpenShift Container Platform クラスターを Red Hat Virtualization (RHV) にインストールするには、CCO には、以下の権限と共に認証情報が必要になります。

- **DiskOperator**
- **DiskCreator**
- **UserTemplateBasedVm**
- **TemplateOwner**
- **TemplateCreator**
- OpenShift Container Platform デプロイメントにターゲットが設定される特定クラスターの **ClusterAdmin**

OpenShift Container Platform クラスターを VMware vSphere にインストールするには、CCO には以下の vSphere 権限と共に認証情報が必要になります。

表6.2 必要な vSphere 権限

カテゴリー	権限
データストア	**領域の割り当て**
フォルダー	**フォルダーの作成、フォルダーの削除**
vSphere タグ	すべての権限
ネットワーク	**ネットワークの割り当て**
リソース	**仮想マシンのリソースプールへの割り当て**
プロファイル駆動型ストレージ	すべての権限
vApp	すべての権限
仮想マシン	すべての権限

passthrough モードの認証情報のメンテナーンス

credentialsRequest CR がクラスターのアップグレード時に変更される場合、各種要件を満たすために passthrough モードの認証情報を手動で更新する必要があります。アップグレード時の認証情報の問題を回避するには、アップグレードの前に、新規バージョンの OpenShift Container Platform のリリース

イメージで **credentialsRequest** CR を確認します。クラウドプロバイダーに必要な **credentialsRequest** CR を見つけるには、AWS、Azure、または GCP のインストールコンテンツの **IAM の手動作成** についてのセクションを参照してください。

インストール後のパーミッションの縮小

passthrough モードを使用する場合、各コンポーネントには他のすべてのコンポーネントによって使用されるパーミッションと同じパーミッションがあります。インストール後にパーミッションを縮小しない場合、すべてのコンポーネントにはインストーラーの実行に必要な幅広いパーミッションが付与されます。

インストール後に、認証情報のパーミッションを、クラスターの実行に必要なパーミッションに制限できます。これは、使用している OpenShift Container Platform バージョンのリリースイメージの **credentialsRequest** CR で定義されます。

AWS、Azure、または GCP に必要な **credentialsRequest** CR を見つけ、CCO が使用するパーミッションを変更する方法については、AWS、Azure、または GCP のインストールコンテンツの **IAM の手動作成** についてのセクションを参照してください。

手動モード

手動モードは AWS でサポートされます。

手動モードでは、ユーザーは CCO の代わりにクラウド認証情報を管理します。このモードを使用するには、実行またはインストールしている OpenShift Container Platform バージョンのリリースイメージの **credentialsRequest** CR を検査し、基礎となるクラウドプロバイダーで対応する認証情報を作成し、クラスターのクラウドプロバイダーのすべての **credentialsRequest** CR に対応するために Kubernetes Secret を正しい namespace に作成する必要があります。

手動モードを使用すると、クラスターに管理者レベルの認証情報を保存する必要なく、各クラスターコンポーネントに必要なパーミッションのみを指定できます。このモードでは、AWS パブリック IAM エンドポイントへの接続も必要ありません。ただし、各アップグレードについて、パーミッションを新規リリースイメージを使用して手動で調整する必要があります。

手動モードを使用するように AWS を設定する方法についての詳細は、**AWS の IAM の手動作成** について参照してください。

無効にされた CCO

無効にされた CCO は Azure および GCP でサポートされます。

Azure または GCP の認証情報を手動で管理するには、CCO を無効にする必要があります。CCO を無効にする場合には、手動モードで CCO を実行する場合と同様の設定やメンテナーンスの要件が多数適用されますが、異なるプロセスでこれを実行できます。詳細は、Azure または GCP のインストールコンテンツの **IAM の手動作成**についてのセクションを参照してください。

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。

6.2. 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

6.3. CLUSTER AUTOSCALER OPERATOR

目的
Cluster Autoscaler Operator は **cluster-api** プロバイダーを使用して OpenShift Cluster Autoscaler のデプロイメントを管理します。

プロジェクト
cluster-autoscaler-operator

CRD

- **ClusterAutoscaler**: これは、クラスターの Autoscaler インスタンスの設定を制御するシングルトンリソースです。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。

6.4. 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

6.5. 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
```

6.6. 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。

6.7. OPENSHIFT CONTROLLER MANAGER OPERATOR

目的

OpenShift Controller Manager Operator は **OpenShiftControllerManager** カスタムリソースをクラスターにインストールし、これを維持します。これは、以下で表示できます。

```
$ oc get clusteroperator openshift-controller-manager -o yaml
```

カスタムリソース定義 (CRD) **openshiftcontrollermanagers.operator.openshift.io** は以下を使用してクラスターで確認できます。

```
$ oc get crd openshiftcontrollermanagers.operator.openshift.io -o yaml
```

プロジェクト
cluster-openshift-controller-manager-operator

6.8. 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** のクラススコープのオブジェクトです。

イメージストリームは、**registry.redhat.io** のイメージを参照する Red Hat Enterprise Linux CoreOS (RHCOS) ベースの OpenShift Container Platform イメージストリームです。同様に、テンプレートは OpenShift Container Platform テンプレートとして分類されます。

Cluster Samples Operator デプロイメントは **openshift-cluster-samples-operator** namespace 内に含まれます。起動時に、インストールプルシークレットは内部レジストリーおよび API サーバーのイメージストリームのインポートロジックによって使用され、**registry.redhat.io** で認証されます。管理者は、サンプルイメージストリームに使用されるレジストリーを変更する場合、追加のシークレットを **openshift** namespace に作成できます。これらのシークレットが作成される場合、これらには、イメージのインポートを容易にするために必要な **docker** の **config.json** のコンテンツが含まれます。

Cluster Samples Operator のイメージには、関連付けられた OpenShift Container Platform リリースのイメージストリームおよびテンプレートの定義が含まれます。Cluster Samples Operator がサンプルを作成した後に、互換性のある OpenShift Container Platform バージョンを示すアノテーションを追加します。Operator はこのアノテーションを使用して、各サンプルを互換性のあるリリースバージョンに一致させるようにします。このインベントリーの外にあるサンプルは省略されるサンプルであるために無視されます。

Operator によって管理されるサンプルへの変更は、バージョンのアノテーションが変更または削除されない限り許可されます。ただし、アップグレード時に、バージョンアノテーションが変更されると、サンプルが新しいバージョンで更新されるため、これらの変更は置き換えられる可能性があります。jenkins イメージはインストールからのイメージペイロードの一部であり、イメージストリームに直接タグ付けされます。

Samples Operator 設定リソースには、削除時に以下を消去するファイナライザーが含まれます。

- Operator 管理のイメージストリーム
- Operator 管理のテンプレート
- Operator が生成する設定リソース

- クラスターステータスのリソース

サンプルリソースの削除時に、Cluster Samples Operator はデフォルト設定を使用してリソースを再作成します。

プロジェクト
cluster-samples-operator

6.9. 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。

6.10. CLUSTER VERSION OPERATOR

目的
Cluster Operator は、クラスター機能の特定の領域を管理します。Cluster Version Operator (CVO) はクラスター Operator のライフサイクルを管理し、その多くはデフォルトで OpenShift Container Platform にインストールされます。

また、CVO は OpenShift Update Service をチェックして、現在のコンポーネントのバージョンとグラフの情報に基づいて、有効な更新と更新パスを確認します。

プロジェクト
cluster-version-operator

関連情報

- OpenShift Container Platform の Operator

6.11. 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

6.12. 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

6.13. 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
```

6.14. 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]
```

6.15. 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
```

6.16. 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

6.17. 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。

6.18. 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**

6.19. MACHINE CONFIG OPERATOR

目的
Machine Congig 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ーランタイムの設定および更新を管理し、適用します。

以下の 4 つのコンポーネントがあります。

- **machine-config-server**: クラスターに参加する新規マシンに Ignition 設定を提供します。
- **machine-config-controller**: マシンのアップグレードを **MachineConfig** オブジェクトで定義される必要な設定に調整します。マシンセットのアップグレードを個別に制御するオプションが提供されます。
- **machine-config-daemon**: 更新時に新規のマシン設定を適用します。マシンの状態を要求されたマシン設定に対して検証し、確認します。
- **machine-config**: インストール時のマシン設定の完全なソース、初回の起動、およびマシンの更新を提供します。

プロジェクト
openshift-machine-config-operator

6.20. 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

6.21. 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、コンテナー化された 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ー化された 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。

コンテナー化された 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ー化された Tuned デーモンが正常に終了する際にロールバックされます。

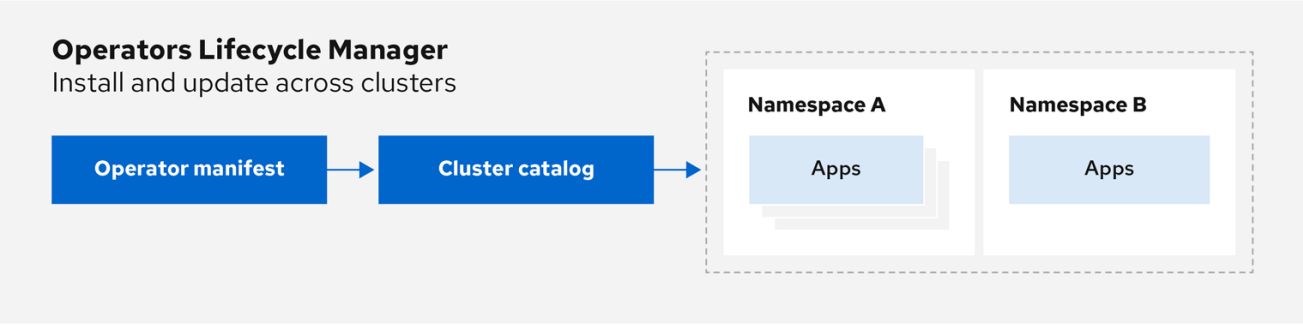
Node Tuning Operator は、バージョン 4.1 以降における標準的な OpenShift Container Platform インストールの一部となっています。

プロジェクト
cluster-node-tuning-operator

6.22. 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6.1 Operator Lifecycle Manager ワークフロー



OLM は OpenShift Container Platform 4.6 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。

CRD

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

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

表6.3 OLM およびカタログ Operator で管理される CRD

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

これらの Operator のそれぞれは以下のリソースの作成も行います。

表6.4 OLM およびカタログ Operator によって作成されるリソース

リソース	所有する Operator
Deployments	OLM
ServiceAccounts	
(Cluster)Role	
(Cluster)RoleBinding	
CustomResourceDefinitions (CRDs)	カタログ

リソース	所有する Operator
ClusterServiceVersions	

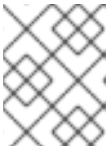
OLM Operator

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ネル内のパッケージへの更新の有無を確認し、必要な場合はそれらを利用可能な最新バージョンに自動的にアップグレードします。

チャネル内のパッケージを追跡するために、必要なパッケージ、チャネル、および更新のプルに使用する **CatalogSource** オブジェクトを設定して **Subscription** オブジェクトを作成できます。更新が見つかると、ユーザーに代わって適切な **InstallPlan** オブジェクトの namespace への書き込みが行われます。

カタログ Operator は以下のワークフローを使用します。

1. クラスターの各カタログソースに接続します。
2. ユーザーによって作成された未解決のインストール計画の有無を確認し、これがあった場合は以下を実行します。
    a. 要求される名前に一致する CSV を検索し、これを解決済みリソースとして追加します。
    b. 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。

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

パッケージマニフェスト は、パッケージアイデンティティーを CSV のセットに関連付けるカタログレジストリー内のエントリーです。パッケージ内で、チャネルは特定の CSV を参照します。CSV は置き換え対象の CSV を明示的に参照するため、パッケージマニフェストはカタログ Operator に対し、CSV をチャネル内の最新バージョンに更新するために必要なすべての情報を提供します (各中間バージョンをステップスルー)。

関連情報

- Operator Lifecycle Manager (OLM) について

6.23. OPENSHIFT 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

6.24. 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

6.25. WINDOWS MACHINE CONFIG OPERATOR

目的

Windows Machine Config Operator (WMCO) は、クラスター上での Windows ワークロードのデプロイし、管理するプロセスをオーケストレーションします。WMCO は Windows マシンをコンピュートノードに設定し、Windows コンテナーのワークロードが OpenShift Container Platform クラスターで実行されるようにします。これは、Docker 形式のコンテナーランタイムがインストールされている Windows イメージを使用するマシンセットを作成して行います。WMCO は、クラスターにコンピュートノードとして参加できるように、基礎となる Windows 仮想マシンを設定するために必要なすべての手順を実行します。

プロジェクト

windows-machine-config-operator