OpenShift Container Platform 4.7

Operators

Working with Operators in OpenShift Container Platform
OpenShift Container Platform 4.7 Operators

Working with Operators in OpenShift Container Platform
Abstract

This document provides information for working with Operators in OpenShift Container Platform. This includes instructions for cluster administrators on how to install and manage Operators, as well as information for developers on how to create applications from installed Operators. This also contains guidance on building your own Operator using the Operator SDK.
Table of Contents

CHAPTER 1. OPERATORS OVERVIEW ........................................................................................................... 11
  1.1. FOR DEVELOPERS ................................................................................................................................ 11
  1.2. FOR ADMINISTRATORS ......................................................................................................................... 11
  1.3. NEXT STEPS ........................................................................................................................................ 12

CHAPTER 2. UNDERSTANDING OPERATORS ................................................................................................. 13
  2.1. WHAT ARE OPERATORS? ...................................................................................................................... 13
    2.1.1. Why use Operators? ...................................................................................................................... 13
    2.1.2. Operator Framework ..................................................................................................................... 13
    2.1.3. Operator maturity model .............................................................................................................. 14
  2.2. OPERATOR FRAMEWORK PACKAGING FORMATS ............................................................................ 15
    2.2.1. Bundle Format ............................................................................................................................... 15
      2.2.1.1. Manifests .................................................................................................................................. 15
          Additionally supported objects .......................................................................................................... 15
      2.2.1.2. Annotations ............................................................................................................................. 16
      2.2.1.3. Dependencies file ...................................................................................................................... 17
      2.2.1.4. About opm ............................................................................................................................... 18
    2.2.2. Package Manifest Format ............................................................................................................... 18
  2.3. OPERATOR FRAMEWORK GLOSSARY OF COMMON TERMS .............................................................. 19
    2.3.1. Common Operator Framework terms ............................................................................................. 19
      2.3.1.1. Bundle ...................................................................................................................................... 19
      2.3.1.2. Bundle image ........................................................................................................................... 19
      2.3.1.3. Catalog source .......................................................................................................................... 19
      2.3.1.4. Catalog image ........................................................................................................................... 19
      2.3.1.5. Channel ..................................................................................................................................... 20
      2.3.1.6. Channel head ........................................................................................................................... 20
      2.3.1.7. Cluster service version ............................................................................................................. 20
      2.3.1.8. Dependency ............................................................................................................................... 20
      2.3.1.9. Index image ............................................................................................................................... 20
      2.3.1.10. Install plan ............................................................................................................................... 20
      2.3.1.11. Operator group ......................................................................................................................... 20
      2.3.1.12. Package .................................................................................................................................... 20
      2.3.1.13. Registry ................................................................................................................................... 21
      2.3.1.14. Subscription ............................................................................................................................. 21
      2.3.1.15. Update graph ........................................................................................................................... 21
  2.4. OPERATOR LIFECYCLE MANAGER (OLM) ............................................................................................ 21
    2.4.1. Operator Lifecycle Manager concepts and resources ......................................................................... 21
      2.4.1.1. What is Operator Lifecycle Manager? ...................................................................................... 21
      2.4.1.2. OLM resources ........................................................................................................................ 21
          2.4.1.2.1. Cluster service version ....................................................................................................... 22
          2.4.1.2.2. Catalog source .................................................................................................................... 22
          2.4.1.2.3. Subscription ........................................................................................................................ 25
          2.4.1.2.4. Install plan .......................................................................................................................... 25
          2.4.1.2.5. Operator groups .................................................................................................................. 27
          2.4.1.2.6. Operator conditions .......................................................................................................... 27
    2.4.2. Operator Lifecycle Manager architecture ....................................................................................... 28
      2.4.2.1. Component responsibilities ...................................................................................................... 28
      2.4.2.2. OLM Operator .......................................................................................................................... 29
      2.4.2.3. Catalog Operator ....................................................................................................................... 29
      2.4.2.4. Catalog Registry ....................................................................................................................... 30
    2.4.3. Operator Lifecycle Manager workflow ............................................................................................. 30
2.4.3.1. Operator installation and upgrade workflow in OLM
2.4.3.1.1. Example upgrade path
2.4.3.1.2. Skipping upgrades
2.4.3.1.3. Replacing multiple Operators
2.4.3.1.4. Z-stream support

2.4.4. Operator Lifecycle Manager dependency resolution
2.4.4.1. About dependency resolution
2.4.4.2. Dependencies file
2.4.4.3. Dependency preferences
2.4.4.3.1. Catalog priority
2.4.4.3.2. Channel ordering
2.4.4.3.3. Order within a channel
2.4.4.3.4. Other constraints
2.4.4.3.4.1. Subscription constraint
2.4.4.3.4.2. Package constraint
2.4.4.4. CRD upgrades
2.4.4.5. Dependency best practices
2.4.4.6. Dependency caveats
2.4.4.7. Example dependency resolution scenarios
  Example: Deprecating dependent APIs
  Example: Version deadlock

2.4.5. Operator groups
2.4.5.1. About Operator groups
2.4.5.2. Operator group membership
2.4.5.3. Target namespace selection
2.4.5.4. Operator group CSV annotations
2.4.5.5. Provided APIs annotation
2.4.5.6. Role-based access control
2.4.5.7. Copied CSVs
2.4.5.8. Static Operator groups
2.4.5.9. Operator group intersection
  Rules for intersection
2.4.5.10. Limitations for multi-tenant Operator management
  2.4.5.10.1. Additional resources
2.4.5.11. Troubleshooting Operator groups
  Membership

2.4.6. Operator conditions
2.4.6.1. About Operator conditions
2.4.6.2. Supported conditions
  2.4.6.2.1. Upgradeable condition
2.4.6.3. Additional resources

2.4.7. Operator Lifecycle Manager metrics
  2.4.7.1. Exposed metrics

2.4.8. Webhook management in Operator Lifecycle Manager
  2.4.8.1. Additional resources

2.5. UNDERSTANDING OPERATORHUB
2.5.1. About OperatorHub
2.5.2. OperatorHub architecture
  2.5.2.1. OperatorHub custom resource
2.5.3. Additional resources

2.6. RED HAT-PROVIDED OPERATOR CATALOGS
2.6.1. About Operator catalogs
2.6.2. About Red Hat-provided Operator catalogs
2.7. CRDS

2.7.1. Extending the Kubernetes API with custom resource definitions

2.7.1.1. Custom resource definitions

2.7.1.2. Creating a custom resource definition

2.7.1.3. Creating cluster roles for custom resource definitions

2.7.1.4. Creating custom resources from a file

2.7.1.5. Inspecting custom resources

2.7.2. Managing resources from custom resource definitions

2.7.2.1. Custom resource definitions

2.7.2.2. Creating custom resources from a file

2.7.2.3. Inspecting custom resources

CHAPTER 3. USER TASKS

3.1. CREATING APPLICATIONS FROM INSTALLED OPERATORS

3.1.1. Creating an etcd cluster using an Operator

3.2. INSTALLING OPERATORS IN YOUR NAMESPACE

3.2.1. Prerequisites

3.2.2. Operator installation with OperatorHub

3.2.3. Installing from OperatorHub using the web console

3.2.4. Installing from OperatorHub using the CLI

3.2.5. Installing a specific version of an Operator

CHAPTER 4. ADMINISTRATOR TASKS

4.1. ADDING OPERATORS TO A CLUSTER

4.1.1. Operator installation with OperatorHub

4.1.2. Installing from OperatorHub using the web console

4.1.3. Installing from OperatorHub using the CLI

4.1.4. Installing a specific version of an Operator

4.1.5. Pod placement of Operator workloads

4.2. UPGRADING INSTALLED OPERATORS

4.2.1. Changing the update channel for an Operator

4.2.2. Manually approving a pending Operator upgrade

4.3. DELETING OPERATORS FROM A CLUSTER

4.3.1. Deleting Operators from a cluster using the web console

4.3.2. Deleting Operators from a cluster using the CLI

4.3.3. Refreshing failing subscriptions

4.4. CONFIGURING PROXY SUPPORT IN OPERATOR LIFECYCLE MANAGER

4.4.1. Overriding proxy settings of an Operator

4.4.2. Injecting a custom CA certificate

4.5. VIEWING OPERATOR STATUS

4.5.1. Operator subscription condition types

4.5.2. Viewing Operator subscription status by using the CLI

4.5.3. Viewing Operator catalog source status by using the CLI

4.6. MANAGING OPERATOR CONDITIONS

4.6.1. Overriding Operator conditions

4.6.2. Updating your Operator to use Operator conditions

4.6.2.1. Setting defaults

4.6.3. Additional resources

4.7. ALLOWING NON-CLUSTER ADMINISTRATORS TO INSTALL OPERATORS

4.7.1. Understanding Operator installation policy

4.7.1.1. Installation scenarios

4.7.1.2. Installation workflow

4.7.2. Scoping Operator installations
4.7.2.1. Fine-grained permissions
4.7.3. Troubleshooting permission failures
4.8. MANAGING CUSTOM CATALOGS
4.8.1. Custom catalogs using the Bundle Format
4.8.1.1. Prerequisites
4.8.1.2. Creating an index image
4.8.1.3. Creating a catalog from an index image
4.8.1.4. Updating an index image
4.8.1.5. Pruning an index image
4.8.2. Custom catalogs using the Package Manifest Format
4.8.2.1. Building a Package Manifest Format catalog image
4.8.2.2. Mirroring a Package Manifest Format catalog image
4.8.2.3. Updating a Package Manifest Format catalog image
4.8.2.4. Testing a Package Manifest Format catalog image
4.8.3. Accessing images for Operators from private registries
4.8.4. Disabling the default OperatorHub sources
4.8.5. Removing custom catalogs
4.9. USING OPERATOR LIFECYCLE MANAGER ON RESTRICTED NETWORKS
4.9.1. Prerequisites
4.9.2. Disabling the default OperatorHub sources
4.9.3. Pruning an index image
4.9.4. Mirroring an Operator catalog
4.9.5. Creating a catalog from an index image
4.9.6. Updating an index image

CHAPTER 5. DEVELOPING OPERATORS

5.1. ABOUT THE OPERATOR SDK
5.1.1. What are Operators?
5.1.2. Development workflow
5.1.3. Additional resources
5.2. INSTALLING THE OPERATOR SDK CLI
5.2.1. Installing the Operator SDK CLI
5.3. GO-BASED OPERATORS
5.3.1. Getting started with Operator SDK for Go-based Operators
5.3.1.1. Prerequisites
5.3.1.2. Creating and deploying Go-based Operators
5.3.1.3. Next steps
5.3.2. Operator SDK tutorial for Go-based Operators
5.3.2.1. Prerequisites
5.3.2.2. Creating a project
5.3.2.2.1. PROJECT file
5.3.2.2.2. About the Manager
5.3.2.2.3. About multi-group APIs
5.3.2.3. Creating an API and controller
5.3.2.3.1. Defining the API
5.3.2.3.2. Generating CRD manifests
5.3.2.3.2.1. About OpenAPI validation
5.3.2.4. Implementing the controller
5.3.2.4.1. Resources watched by the controller
5.3.2.4.2. Controller configurations
5.3.2.4.3. Reconcile loop
5.3.2.4.4. Permissions and RBAC manifests
5.3.2.5. Running the Operator
5.5.1.3. Next steps

5.5.2. Operator SDK tutorial for Helm-based Operators
5.5.2.1. Prerequisites
5.5.2.2. Creating a project
  5.5.2.2.1. Existing Helm charts
  5.5.2.2.2. PROJECT file
5.5.2.3. Understanding the Operator logic
  5.5.2.3.1. Sample Helm chart
  5.5.2.3.2. Modifying the custom resource spec
5.5.2.4. Running the Operator
  5.5.2.4.1. Running locally outside the cluster
  5.5.2.4.2. Preparing your Operator to use supported images
  5.5.2.4.3. Running as a deployment on the cluster
  5.5.2.4.4. Bundling an Operator and deploying with Operator Lifecycle Manager
5.5.2.5. Creating a custom resource
5.5.2.6. Additional resources
5.5.3. Project layout for Helm-based Operators
  5.5.3.1. Helm-based project layout
5.5.4. Helm support in Operator SDK
  5.5.4.1. Helm charts

5.6. DEFINING CLUSTER SERVICE VERSIONS (CSVs)
5.6.1. How CSV generation works
  5.6.1.1. Generated files and resources
  5.6.1.2. Version management
5.6.2. Manually-defined CSV fields
  5.6.2.1. Operator metadata annotations
    Example use cases
5.6.3. Enabling your Operator for restricted network environments
5.6.4. Enabling your Operator for multiple architectures and operating systems
  5.6.4.1. Architecture and operating system support for Operators
5.6.5. Setting a suggested namespace
5.6.6. Enabling Operator conditions
5.6.7. Defining webhooks
  5.6.7.1. Webhook considerations for OLM
    Certificate authority constraints
    Admission webhook rules constraints
    Conversion webhook constraints
5.6.8. Understanding your custom resource definitions (CRDs)
  5.6.8.1. Owned CRDs
  5.6.8.2. Required CRDs
  5.6.8.3. CRD upgrades
    5.6.8.3.1. Adding a new CRD version
    5.6.8.3.2. Deprecating or removing a CRD version
  5.6.8.4. CRD templates
  5.6.8.5. Hiding internal objects
  5.6.8.6. Initializing required custom resources
5.6.9. Understanding your API services
  5.6.9.1. Owned API services
    5.6.9.1.1. API service resource creation
    5.6.9.1.2. API service serving certificates
  5.6.9.2. Required API services
5.7. WORKING WITH BUNDLE IMAGES
  5.7.1. Bundling an Operator and deploying with Operator Lifecycle Manager
5.7.2. Testing an Operator upgrade on Operator Lifecycle Manager
5.7.3. Additional resources

5.8. VALIDATING OPERATORS USING THE SCORECARD TOOL
5.8.1. About the scorecard tool
5.8.2. Scorecard configuration
5.8.3. Built-in scorecard tests
5.8.4. Running the scorecard tool
5.8.5. Scorecard output
5.8.6. Selecting tests
5.8.7. Enabling parallel testing
5.8.8. Custom scorecard tests

5.9. CONFIGURING BUILT-IN MONITORING WITH PROMETHEUS
5.9.1. Prometheus Operator support
5.9.2. Metrics helper
  5.9.2.1. Modifying the metrics port
5.9.3. Service monitors
  5.9.3.1. Creating service monitors

5.10. CONFIGURING LEADER ELECTION
5.10.1. Operator leader election examples
  5.10.1.1. Leader-for-life election
  5.10.1.2. Leader-with-lease election

5.11. OPERATOR SDK CLI REFERENCE
5.11.1. bundle
  5.11.1.1. validate
5.11.2. cleanup
5.11.3. completion
5.11.4. create
  5.11.4.1. api
5.11.5. generate
  5.11.5.1. bundle
  5.11.5.2. kustomize
    5.11.5.2.1. manifests
5.11.6. init
5.11.7. run
  5.11.7.1. bundle
  5.11.7.2. bundle-upgrade
5.11.8. scorecard

CHAPTER 6. PLATFORM OPERATORS REFERENCE

6.1. CLOUD CREDENTIAL OPERATOR
  Purpose
  Project
  CRDs
  Configuration objects
  Additional resources

6.2. CLUSTER AUTHENTICATION OPERATOR
  Purpose
  Project

6.3. CLUSTER AUTOSCALER OPERATOR
  Purpose
  Project
  CRDs

6.4. CLUSTER CONFIG OPERATOR
6.5. CLUSTER CSI SNAPSHOT CONTROLLER OPERATOR
Purpose
Project

6.6. CLUSTER IMAGE REGISTRY OPERATOR
Purpose
Project

6.7. CLUSTER MACHINE APPROVER OPERATOR
Purpose
Project

6.8. CLUSTER MONITORING OPERATOR
Purpose
Project
CRDs
Configuration objects

6.9. CLUSTER NETWORK OPERATOR
Purpose

6.10. OPENSHEET CONTROLLER MANAGER OPERATOR
Purpose
Project

6.11. CLUSTER SAMPLES OPERATOR
Purpose
Project

6.12. CLUSTER STORAGE OPERATOR
Purpose
Project
Configuration
Notes

6.13. CLUSTER VERSION OPERATOR
Purpose
Project

6.14. CONSOLE OPERATOR
Purpose
Project

6.15. DNS OPERATOR
Purpose
Project

6.16. ETCD CLUSTER OPERATOR
Purpose
Project
CRDs
Configuration objects

6.17. INGRESS OPERATOR
Purpose
Project
CRDs
Configuration objects
Notes

6.18. INSIGHTS OPERATOR
Purpose
Project
Configuration
6.19. KUBERNETES API SERVER OPERATOR
   Purpose 252
   Project 252
   CRDs 252
   Configuration objects 252

6.20. KUBERNETES CONTROLLER MANAGER OPERATOR
   Purpose 252
   Project 253

6.21. KUBERNETES SCHEDULER OPERATOR
   Purpose 253
   Project 253
   Configuration 253

6.22. KUBERNETES STORAGE VERSION MIGRATOR OPERATOR
   Purpose 253
   Project 253

6.23. MACHINE API OPERATOR
   Purpose 253
   Project 254

6.24. MACHINE CONFIG OPERATOR
   Purpose 254
   Project 254

6.25. MARKETPLACE OPERATOR
   Purpose 254
   Project 254

6.26. NODE TUNING OPERATOR
   Purpose 254
   Project 254

6.27. OPENSHIFT API SERVER OPERATOR
   Purpose 255
   Project 255
   CRDs 255

6.28. OPERATOR LIFECYCLE MANAGER OPERATORS
   Purpose 255
   CRDs 256
   OLM Operator 257
   Catalog Operator 257
   Catalog Registry 258
   Additional resources 258

6.29. OPENSHIFT SERVICE CA OPERATOR
   Purpose 258
   Project 258

6.30. VSPHERE PROBLEM DETECTOR OPERATOR
   Purpose 258
   Configuration 258
   Notes 258
CHAPTER 1. OPERATORS OVERVIEW

Operators are among the most important components of OpenShift Container Platform. Operators are the preferred method of packaging, deploying, and managing services on the control plane. They can also provide advantages to applications that users run.

Operators integrate with Kubernetes APIs and CLI tools such as `kubectl` and `oc` commands. They provide the means of monitoring applications, performing health checks, managing over-the-air (OTA) updates, and ensuring that applications remain in your specified state.

While both follow similar Operator concepts and goals, Operators in OpenShift Container Platform are managed by two different systems, depending on their purpose:

- **Platform Operators**, which are managed by the Cluster Version Operator (CVO), are installed by default to perform cluster functions.
- **Optional add-on Operators**, which are managed by Operator Lifecycle Manager (OLM), can be made accessible for users to run in their applications.

With Operators, you can create applications to monitor the running services in the cluster. Operators are designed specifically for your applications. Operators implement and automate the common Day 1 operations such as installation and configuration as well as Day 2 operations such as autoscaling up and down and creating backups. All these activities are in a piece of software running inside your cluster.

1.1. FOR DEVELOPERS

As a developer, you can perform the following Operator tasks:

- Install Operator SDK CLI.
- Create Go-based Operators, Ansible-based Operators, and Helm-based Operators.
- Use Operator SDK to build, test, and deploy an Operator.
- Install and subscribe an Operator to your namespace.
- Create an application from an installed Operator through the web console.

1.2. FOR ADMINISTRATORS

As a cluster administrator, you can perform the following Operator tasks:

- Manage custom catalogs
- Allow non-cluster administrators to install Operators
- Install an Operator from OperatorHub
- View Operator status.
- Manage Operator conditions
- Upgrade installed Operators
- Delete installed Operators
- Configure proxy support
- Use Operator Lifecycle Manager on restricted networks

To know all about the platform Operators that Red Hat provides, see Platform Operators reference.

1.3. NEXT STEPS

To understand more about Operators, see What are Operators?
2.1. WHAT ARE OPERATORS?

Conceptually, Operators take human operational knowledge and encode it into software that is more easily shared with consumers.

Operators are pieces of software that ease the operational complexity of running another piece of software. They act like an extension of the software vendor’s engineering team, monitoring a Kubernetes environment (such as OpenShift Container Platform) and using its current state to make decisions in real time. Advanced Operators are designed to handle upgrades seamlessly, react to failures automatically, and not take shortcuts, like skipping a software backup process to save time.

More technically, Operators are a method of packaging, deploying, and managing a Kubernetes application.

A Kubernetes application is an app that is both deployed on Kubernetes and managed using the Kubernetes APIs and `kubectl` or `oc` tooling. To be able to make the most of Kubernetes, you require a set of cohesive APIs to extend in order to service and manage your apps that run on Kubernetes. Think of Operators as the runtime that manages this type of app on Kubernetes.

2.1.1. Why use Operators?

Operators provide:

- Repeatability of installation and upgrade.
- Constant health checks of every system component.
- Over-the-air (OTA) updates for OpenShift components and ISV content.
- A place to encapsulate knowledge from field engineers and spread it to all users, not just one or two.

Why deploy on Kubernetes?

Kubernetes (and by extension, OpenShift Container Platform) contains all of the primitives needed to build complex distributed systems – secret handling, load balancing, service discovery, autoscaling – that work across on-premise and cloud providers.

Why manage your app with Kubernetes APIs and `kubectl` tooling?

These APIs are feature rich, have clients for all platforms and plug into the cluster’s access control/auditing. An Operator uses the Kubernetes extension mechanism, custom resource definitions (CRDs), so your custom object, for example MongoDB, looks and acts just like the built-in, native Kubernetes objects.

How do Operators compare with service brokers?

A service broker is a step towards programmatic discovery and deployment of an app. However, because it is not a long running process, it cannot execute Day 2 operations like upgrade, failover, or scaling. Customizations and parameterization of tunables are provided at install time, versus an Operator that is constantly watching the current state of your cluster. Off-cluster services are a good match for a service broker, although Operators exist for these as well.

2.1.2. Operator Framework

The Operator Framework is a family of tools and capabilities to deliver on the customer experience.
described above. It is not just about writing code; testing, delivering, and updating Operators is just as important. The Operator Framework components consist of open source tools to tackle these problems:

**Operator SDK**

The Operator SDK assists Operator authors in bootstrapping, building, testing, and packaging their own Operator based on their expertise without requiring knowledge of Kubernetes API complexities.

**Operator Lifecycle Manager**

Operator Lifecycle Manager (OLM) controls the installation, upgrade, and role-based access control (RBAC) of Operators in a cluster. Deployed by default in OpenShift Container Platform 4.7.

**Operator Registry**

The Operator Registry stores cluster service versions (CSVs) and custom resource definitions (CRDs) for creation in a cluster and stores Operator metadata about packages and channels. It runs in a Kubernetes or OpenShift cluster to provide this Operator catalog data to OLM.

**OperatorHub**

OperatorHub is a web console for cluster administrators to discover and select Operators to install on their cluster. It is deployed by default in OpenShift Container Platform.

These tools are designed to be composable, so you can use any that are useful to you.

### 2.1.3. Operator maturity model

The level of sophistication of the management logic encapsulated within an Operator can vary. This logic is also in general highly dependent on the type of the service represented by the Operator.

One can however generalize the scale of the maturity of the encapsulated operations of an Operator for certain set of capabilities that most Operators can include. To this end, the following Operator maturity model defines five phases of maturity for generic day two operations of an Operator:

**Figure 2.1. Operator maturity model**

The above model also shows how these capabilities can best be developed through the Helm, Go, and Ansible capabilities of the Operator SDK.
2.2. OPERATOR FRAMEWORK PACKAGING FORMATS

This guide outlines the packaging formats for Operators supported by Operator Lifecycle Manager (OLM) in OpenShift Container Platform.

2.2.1. Bundle Format

The Bundle Format for Operators is a new packaging format introduced by the Operator Framework. To improve scalability and to better enable upstream users hosting their own catalogs, the Bundle Format specification simplifies the distribution of Operator metadata.

An Operator bundle represents a single version of an Operator. On-disk bundle manifests are containerized and shipped as a bundle image, which is a non-runnable container image that stores the Kubernetes manifests and Operator metadata. Storage and distribution of the bundle image is then managed using existing container tools like podman and docker and container registries such as Quay.

Operator metadata can include:

- Information that identifies the Operator, for example its name and version.
- Additional information that drives the UI, for example its icon and some example custom resources (CRs).
- Required and provided APIs.
- Related images.

When loading manifests into the Operator Registry database, the following requirements are validated:

- The bundle must have at least one channel defined in the annotations.
- Every bundle has exactly one cluster service version (CSV).
- If a CSV owns a custom resource definition (CRD), that CRD must exist in the bundle.

2.2.1.1. Manifests

Bundle manifests refer to a set of Kubernetes manifests that define the deployment and RBAC model of the Operator.

A bundle includes one CSV per directory and typically the CRDs that define the owned APIs of the CSV in its /manifests directory.

Example Bundle Format layout

```
estcd
  └── manifests
       └── etcdcluster.crd.yaml
        └── etcdoperator.clusterserviceversion.yaml
       └── secret.yaml
        └── configmap.yaml
        └── metadata
             └── annotations.yaml
             └── dependencies.yaml
```

Additionally supported objects
The following object types can also be optionally included in the `/manifests` directory of a bundle:

Supported optional object types

- ClusterRole
- ClusterRoleBinding
- ConfigMap
- ConsoleYamlSample
- PodDisruptionBudget
- PriorityClass
- PrometheusRule
- Role
- RoleBinding
- Secret
- Service
- ServiceAccount
- ServiceMonitor
- VerticalPodAutoscaler

When these optional objects are included in a bundle, Operator Lifecycle Manager (OLM) can create them from the bundle and manage their lifecycle along with the CSV:

Lifecycle for optional objects

- When the CSV is deleted, OLM deletes the optional object.
- When the CSV is upgraded:
  - If the name of the optional object is the same, OLM updates it in place.
  - If the name of the optional object has changed between versions, OLM deletes and recreates it.

2.2.1.2. Annotations

A bundle also includes an `annotations.yaml` file in its `/metadata` directory. This file defines higher level aggregate data that helps describe the format and package information about how the bundle should be added into an index of bundles:

Example annotations.yaml

```yaml
annotations:
  operators.operatorframework.io.bundle.mediatype.v1: "registry+v1"
```
operators.operatorframework.io.bundle.manifests.v1: "manifests/"  
operators.operatorframework.io.bundle.metadata.v1: "metadata/"  
operators.operatorframework.io.bundle.package.v1: "test-operator"  
operators.operatorframework.io.bundle.channels.v1: "beta,stable"  
operators.operatorframework.io.bundle.channel.default.v1: "stable"

1. The media type or format of the Operator bundle. The registry+v1 format means it contains a CSV and its associated Kubernetes objects.

2. The path in the image to the directory that contains the Operator manifests. This label is reserved for future use and currently defaults to manifests/. The value manifests.v1 implies that the bundle contains Operator manifests.

3. The path in the image to the directory that contains metadata files about the bundle. This label is reserved for future use and currently defaults to metadata/. The value metadata.v1 implies that this bundle has Operator metadata.

4. The package name of the bundle.

5. The list of channels the bundle is subscribing to when added into an Operator Registry.

6. The default channel an Operator should be subscribed to when installed from a registry.

**NOTE**

In case of a mismatch, the annotations.yaml file is authoritative because the on-cluster Operator Registry that relies on these annotations only has access to this file.

### 2.2.1.3. Dependencies file

The dependencies of an Operator are listed in a dependencies.yaml file in the metadata/ folder of a bundle. This file is optional and currently only used to specify explicit Operator-version dependencies.

The dependency list contains a type field for each item to specify what kind of dependency this is. There are two supported types of Operator dependencies:

- **olm.package**: This type indicates a dependency for a specific Operator version. The dependency information must include the package name and the version of the package in semver format. For example, you can specify an exact version such as 0.5.2 or a range of versions such as >0.5.1.

- **olm.gvk**: With a gvk type, the author can specify a dependency with group/version/kind (GVK) information, similar to existing CRD and API-based usage in a CSV. This is a path to enable Operator authors to consolidate all dependencies, API or explicit versions, to be in the same place.

In the following example, dependencies are specified for a Prometheus Operator and etcd CRDs:

**Example dependencies.yaml file**

dependencies:
- type: olm.package
  value:
    packageName: prometheus
2.2.1.4. About opm

The opm CLI tool is provided by the Operator Framework for use with the Operator Bundle Format. This tool allows you to create and maintain catalogs of Operators from a list of bundles, called an index, that are similar to software repositories. The result is a container image, called an index image, which can be stored in a container registry and then installed on a cluster.

An index contains a database of pointers to Operator manifest content that can be queried through an included API that is served when the container image is run. On OpenShift Container Platform, Operator Lifecycle Manager (OLM) can use the index image as a catalog by referencing it in a CatalogSource object, which polls the image at regular intervals to enable frequent updates to installed Operators on the cluster.

- See CLI tools for steps on installing the opm CLI.

2.2.2. Package Manifest Format

The Package Manifest Format for Operators is the legacy packaging format introduced by the Operator Framework. While this format is deprecated in OpenShift Container Platform 4.5, it is still supported and Operators provided by Red Hat are currently shipped using this method.

In this format, a version of an Operator is represented by a single cluster service version (CSV) and typically the custom resource definitions (CRDs) that define the owned APIs of the CSV, though additional objects may be included.

All versions of the Operator are nested in a single directory:

Example Package Manifest Format layout

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

OpenShift Container Platform 4.7 Operators
It also includes a `<name>.package.yaml` file that is the *package manifest* that defines the package name and channels details:

**Example package manifest**

```yaml
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
```

When loading package manifests into the Operator Registry database, the following requirements are validated:

- Every package has at least one channel.
- Every CSV pointed to by a channel in a package exists.
- Every version of an Operator has exactly one CSV.
- If a CSV owns a CRD, that CRD must exist in the directory of the Operator version.
- If a CSV replaces another, both the old and the new must exist in the package.

### 2.3. OPERATOR FRAMEWORK GLOSSARY OF COMMON TERMS

This topic provides a glossary of common terms related to the Operator Framework, including Operator Lifecycle Manager (OLM) and the Operator SDK, for both packaging formats: Package Manifest Format and Bundle Format.

#### 2.3.1. Common Operator Framework terms

**2.3.1.1. Bundle**

In the Bundle Format, a *bundle* is a collection of an Operator CSV, manifests, and metadata. Together, they form a unique version of an Operator that can be installed onto the cluster.

**2.3.1.2. Bundle image**

In the Bundle Format, a *bundle image* is a container image that is built from Operator manifests and that contains one bundle. Bundle images are stored and distributed by Open Container Initiative (OCI) spec container registries, such as Quay.io or DockerHub.

**2.3.1.3. Catalog source**

A *catalog source* is a repository of CSVs, CRDs, and packages that define an application.

**2.3.1.4. Catalog image**
In the Package Manifest Format, a **catalog image** is a containerized datastore that describes a set of Operator metadata and update metadata that can be installed onto a cluster using OLM.

### 2.3.1.5. Channel

A **channel** defines a stream of updates for an Operator and is used to roll out updates for subscribers. The head points to the latest version of that channel. For example, a **stable** channel would have all stable versions of an Operator arranged from the earliest to the latest.

An Operator can have several channels, and a subscription binding to a certain channel would only look for updates in that channel.

### 2.3.1.6. Channel head

A **channel head** refers to the latest known update in a particular channel.

### 2.3.1.7. Cluster service version

A **cluster service version** (CSV) is a YAML manifest created from Operator metadata that assists OLM in running the Operator in a cluster. It is the metadata that accompanies an Operator container image, used to populate user interfaces with information such as its logo, description, and version.

It is also a source of technical information that is required to run the Operator, like the RBAC rules it requires and which custom resources (CRs) it manages or depends on.

### 2.3.1.8. Dependency

An Operator may have a dependency on another Operator being present in the cluster. For example, the Vault Operator has a dependency on the etcd Operator for its data persistence layer.

OLM resolves dependencies by ensuring that all specified versions of Operators and CRDs are installed on the cluster during the installation phase. This dependency is resolved by finding and installing an Operator in a catalog that satisfies the required CRD API, and is not related to packages or bundles.

### 2.3.1.9. Index image

In the Bundle Format, an **index image** refers to an image of a database (a database snapshot) that contains information about Operator bundles including CSVs and CRDs of all versions. This index can host a history of Operators on a cluster and be maintained by adding or removing Operators using the opm CLI tool.

### 2.3.1.10. Install plan

An **install plan** is a calculated list of resources to be created to automatically install or upgrade a CSV.

### 2.3.1.11. Operator group

An **Operator group** configures all Operators deployed in the same namespace as the OperatorGroup object to watch for their CR in a list of namespaces or cluster-wide.

### 2.3.1.12. Package

In the Bundle Format, a **package** is a directory that encloses all released history of an Operator with each version. A released version of an Operator is described in a CSV manifest alongside the CRDs.
2.3.1.13. Registry

A registry is a database that stores bundle images of Operators, each with all of its latest and historical versions in all channels.

2.3.1.14. Subscription

A subscription keeps CSVs up to date by tracking a channel in a package.

2.3.1.15. Update graph

An update graph links versions of CSVs together, similar to the update graph of any other packaged software. Operators can be installed sequentially, or certain versions can be skipped. The update graph is expected to grow only at the head with newer versions being added.

2.4. OPERATOR LIFECYCLE MANAGER (OLM)

2.4.1. Operator Lifecycle Manager concepts and resources

This guide provides an overview of the concepts that drive Operator Lifecycle Manager (OLM) in OpenShift Container Platform.

2.4.1.1. What is Operator Lifecycle Manager?

Operator Lifecycle Manager (OLM) helps users install, update, and manage the lifecycle of Kubernetes native applications (Operators) and their associated services running across their OpenShift Container Platform clusters. It is part of the Operator Framework, an open source toolkit designed to manage Operators in an effective, automated, and scalable way.

Figure 2.2. Operator Lifecycle Manager workflow

OLM runs by default in OpenShift Container Platform 4.7, which aids cluster administrators in installing, upgrading, and granting access to Operators running on their cluster. The OpenShift Container Platform web console provides management screens for cluster administrators to install Operators, as well as grant specific projects access to use the catalog of Operators available on the cluster.

For developers, a self-service experience allows provisioning and configuring instances of databases, monitoring, and big data services without having to be subject matter experts, because the Operator has that knowledge baked into it.

2.4.1.2. OLM resources
The following custom resource definitions (CRDs) are defined and managed by Operator Lifecycle Manager (OLM):

### Table 2.1. CRDs managed by OLM and Catalog Operators

<table>
<thead>
<tr>
<th>Resource</th>
<th>Short name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ClusterServiceVersion</td>
<td>csv</td>
<td>Application metadata. For example: name, version, icon, required resources.</td>
</tr>
<tr>
<td>CatalogSource</td>
<td>catsrc</td>
<td>A repository of CSVs, CRDs, and packages that define an application.</td>
</tr>
<tr>
<td>Subscription</td>
<td>sub</td>
<td>Keeps CSVs up to date by tracking a channel in a package.</td>
</tr>
<tr>
<td>InstallPlan</td>
<td>ip</td>
<td>Calculated list of resources to be created to automatically install or upgrade a CSV.</td>
</tr>
<tr>
<td>OperatorGroup</td>
<td>og</td>
<td>Configures all Operators deployed in the same namespace as the OperatorGroup object to watch for their custom resource (CR) in a list of namespaces or cluster-wide.</td>
</tr>
<tr>
<td>OperatorConditions</td>
<td>-</td>
<td>Creates a communication channel between OLM and an Operator it manages. Operators can write to the Status.Conditions array to communicate complex states to OLM.</td>
</tr>
</tbody>
</table>

#### 2.4.1.2.1. Cluster service version

A *cluster service version* (CSV) represents a specific version of a running Operator on an OpenShift Container Platform cluster. It is a YAML manifest created from Operator metadata that assists Operator Lifecycle Manager (OLM) in running the Operator in the cluster.

OLM requires this metadata about an Operator to ensure that it can be kept running safely on a cluster, and to provide information about how updates should be applied as new versions of the Operator are published. This is similar to packaging software for a traditional operating system; think of the packaging step for OLM as the stage at which you make your *rpm*, *deb*, or *apk* bundle.

A CSV includes the metadata that accompanies an Operator container image, used to populate user interfaces with information such as its name, version, description, labels, repository link, and logo.

A CSV is also a source of technical information required to run the Operator, such as which custom resources (CRs) it manages or depends on, RBAC rules, cluster requirements, and install strategies. This information tells OLM how to create required resources and set up the Operator as a deployment.

#### 2.4.1.2.2. Catalog source

A *catalog source* represents a store of metadata, typically by referencing an *index image* stored in a container registry. Operator Lifecycle Manager (OLM) queries catalog sources to discover and install Operators and their dependencies. The OperatorHub in the OpenShift Container Platform web console also displays the Operators provided by catalog sources.
TIP

Cluster administrators can view the full list of Operators provided by an enabled catalog source on a cluster by using the Administration → Cluster Settings → Global Configuration → OperatorHub page in the web console.

The spec of a CatalogSource object indicates how to construct a pod or how to communicate with a service that serves the Operator Registry gRPC API.

Example 2.1. Example CatalogSource object

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

1 Name for the CatalogSource object. This value is also used as part of the name for the related pod that is created in the requested namespace.

2 Namespace to create the catalog available. To make the catalog available cluster-wide in all namespaces, set this value to openshift-marketplace. The default Red Hat-provided catalog sources also use the openshift-marketplace namespace. Otherwise, set the value to a specific namespace to make the Operator only available in that namespace.

3 Display name for the catalog in the web console and CLI.

4 Index image for the catalog.

5 Weight for the catalog source. OLM uses the weight for prioritization during dependency resolution. A higher weight indicates the catalog is preferred over lower-weighted catalogs.
Source types include the following:

- **grpc** with an **image** reference: OLM pulls the image and runs the pod, which is expected to serve a compliant API.

- **grpc** with an **address** field: OLM attempts to contact the gRPC API at the given address. This should not be used in most cases.

- **configmap**: OLM parses config map data and runs a pod that can serve the gRPC API over it.

Automatically check for new versions at a given interval to stay up-to-date.

Last observed state of the catalog connection. For example:

- **READY**: A connection is successfully established.

- **CONNECTING**: A connection is attempting to establish.

- **TRANSIENT_FAILURE**: A temporary problem has occurred while attempting to establish a connection, such as a timeout. The state will eventually switch back to **CONNECTING** and try again.

See **States of Connectivity** in the gRPC documentation for more details.

Latest time the container registry storing the catalog image was polled to ensure the image is up-to-date.

Status information for the catalog’s Operator Registry service.

Referencing the **name** of a **CatalogSource** object in a subscription instructs OLM where to search to find a requested Operator:

**Example 2.2. Example Subscription object referencing a catalog source**

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

**Additional resources**

- Understanding OperatorHub
- Red Hat-provided Operator catalogs
- Catalog priority
2.4.1.2.3. Subscription

A subscription, defined by a Subscription object, represents an intention to install an Operator. It is the custom resource that relates an Operator to a catalog source.

Subscriptions describe which channel of an Operator package to subscribe to, and whether to perform updates automatically or manually. If set to automatic, the subscription ensures Operator Lifecycle Manager (OLM) manages and upgrades the Operator to ensure that the latest version is always running in the cluster.

Example Subscription object

```yaml
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
```

This Subscription object defines the name and namespace of the Operator, as well as the catalog from which the Operator data can be found. The channel, such as alpha, beta, or stable, helps determine which Operator stream should be installed from the catalog source.

The names of channels in a subscription can differ between Operators, but the naming scheme should follow a common convention within a given Operator. For example, channel names might follow a minor release update stream for the application provided by the Operator (1.2, 1.3) or a release frequency (stable, fast).

In addition to being easily visible from the OpenShift Container Platform web console, it is possible to identify when there is a newer version of an Operator available by inspecting the status of the related subscription. The value associated with the currentCSV field is the newest version that is known to OLM, and installedCSV is the version that is installed on the cluster.

Additional resources

- Viewing Operator subscription status by using the CLI

2.4.1.2.4. Install plan

An install plan, defined by an InstallPlan object, describes a set of resources that Operator Lifecycle Manager (OLM) creates to install or upgrade to a specific version of an Operator. The version is defined by a cluster service version (CSV).

To install an Operator, a cluster administrator, or a user who has been granted Operator installation permissions, must first create a Subscription object. A subscription represents the intent to subscribe to a stream of available versions of an Operator from a catalog source. The subscription then creates an InstallPlan object to facilitate the installation of the resources for the Operator.

The install plan must then be approved according to one of the following approval strategies:
- If the subscription’s `spec.installPlanApproval` field is set to **Automatic**, the install plan is approved automatically.

- If the subscription’s `spec.installPlanApproval` field is set to **Manual**, the install plan must be manually approved by a cluster administrator or user with proper permissions.

After the install plan is approved, OLM creates the specified resources and installs the Operator in the namespace that is specified by the subscription.

**Example 2.3. Example InstallPlan object**

```
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:
  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:
```

OpenShift Container Platform 4.7 Operators
2.4.1.2.5. Operator groups

An Operator group, defined by the OperatorGroup resource, provides multitenant configuration to OLM-installed Operators. An Operator group selects target namespaces in which to generate required RBAC access for its member Operators.

The set of target namespaces is provided by a comma-delimited string stored in the olm.targetNamespaces annotation of a cluster service version (CSV). This annotation is applied to the CSV instances of member Operators and is projected into their deployments.

Additional resources

- Allowing non-cluster administrators to install Operators

2.4.1.2.6. Operator conditions

As part of its role in managing the lifecycle of an Operator, Operator Lifecycle Manager (OLM) infers
the state of an Operator from the state of Kubernetes resources that define the Operator. While this approach provides some level of assurance that an Operator is in a given state, there are many instances where an Operator might need to communicate information to OLM that could not be inferred otherwise. This information can then be used by OLM to better manage the lifecycle of the Operator.

OLM provides a custom resource definition (CRD) called `OperatorCondition` that allows Operators to communicate conditions to OLM. There are a set of supported conditions that influence management of the Operator by OLM when present in the `Status.Conditions` array of an `OperatorCondition` resource.

**Additional resources**

- Operator conditions.

### 2.4.2. Operator Lifecycle Manager architecture

This guide outlines the component architecture of Operator Lifecycle Manager (OLM) in OpenShift Container Platform.

#### 2.4.2.1. Component responsibilities

Operator Lifecycle Manager (OLM) is composed of two Operators: the OLM Operator and the Catalog Operator.

Each of these Operators is responsible for managing the custom resource definitions (CRDs) that are the basis for the OLM framework:

**Table 2.2. CRDs managed by OLM and Catalog Operators**

<table>
<thead>
<tr>
<th>Resource</th>
<th>Short name</th>
<th>Owner</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>ClusterServiceVersion</code></td>
<td>csv</td>
<td>OLM</td>
<td>Application metadata: name, version, icon, required resources, installation, and so on.</td>
</tr>
<tr>
<td><code>InstallPlan</code></td>
<td>ip</td>
<td>Catalog</td>
<td>Calculated list of resources to be created to automatically install or upgrade a CSV.</td>
</tr>
<tr>
<td><code>CatalogSource</code></td>
<td>cats</td>
<td>Catalog</td>
<td>A repository of CSVs, CRDs, and packages that define an application.</td>
</tr>
<tr>
<td><code>Subscription</code></td>
<td>sub</td>
<td>Catalog</td>
<td>Used to keep CSVs up to date by tracking a channel in a package.</td>
</tr>
<tr>
<td><code>OperatorGroup</code></td>
<td>og</td>
<td>OLM</td>
<td>Configures all Operators deployed in the same namespace as the <strong>OperatorGroup</strong> object to watch for their custom resource (CR) in a list of namespaces or cluster-wide.</td>
</tr>
</tbody>
</table>

Each of these Operators is also responsible for creating the following resources:
Table 2.3. Resources created by OLM and Catalog Operators

<table>
<thead>
<tr>
<th>Resource</th>
<th>Owner</th>
</tr>
</thead>
<tbody>
<tr>
<td>Deployments</td>
<td>OLM</td>
</tr>
<tr>
<td>ServiceAccounts</td>
<td></td>
</tr>
<tr>
<td>(Cluster)Roles</td>
<td></td>
</tr>
<tr>
<td>(Cluster)RoleBindings</td>
<td></td>
</tr>
<tr>
<td>CustomResourceDefinitions (CRDs)</td>
<td>Catalog</td>
</tr>
<tr>
<td>ClusterServiceVersions</td>
<td></td>
</tr>
</tbody>
</table>

2.4.2.2. OLM Operator

The OLM Operator is responsible for deploying applications defined by CSV resources after the required resources specified in the CSV are present in the cluster.

The OLM Operator is not concerned with the creation of the required resources; you can choose to manually create these resources using the CLI or using the Catalog Operator. This separation of concern allows users incremental buy-in in terms of how much of the OLM framework they choose to leverage for their application.

The OLM Operator uses the following workflow:

1. Watch for cluster service versions (CSVs) in a namespace and check that requirements are met.
2. If requirements are met, run the install strategy for the CSV.

NOTE
A CSV must be an active member of an Operator group for the install strategy to run.

2.4.2.3. Catalog Operator

The Catalog Operator is responsible for resolving and installing cluster service versions (CSVs) and the required resources they specify. It is also responsible for watching catalog sources for updates to packages in channels and upgrading them, automatically if desired, to the latest available versions.

To track a package in a channel, you can create a Subscription object configuring the desired package, channel, and the CatalogSource object you want to use for pulling updates. When updates are found, an appropriate InstallPlan object is written into the namespace on behalf of the user.

The Catalog Operator uses the following workflow:

1. Connect to each catalog source in the cluster.
2. Watch for unresolved install plans created by a user, and if found:
   a. Find the CSV matching the name requested and add the CSV as a resolved resource.
a. Find the CSV matching the name requested and add the CSV as a resolved resource.

b. For each managed or required CRD, add the CRD as a resolved resource.

c. For each required CRD, find the CSV that manages it.

3. Watch for resolved install plans and create all of the discovered resources for it, if approved by a user or automatically.

4. Watch for catalog sources and subscriptions and create install plans based on them.

2.4.2.4. Catalog Registry

The Catalog Registry stores CSVs and CRDs for creation in a cluster and stores metadata about packages and channels.

A package manifest is an entry in the Catalog Registry that associates a package identity with sets of CSVs. Within a package, channels point to a particular CSV. Because CSVs explicitly reference the CSV that they replace, a package manifest provides the Catalog Operator with all of the information that is required to update a CSV to the latest version in a channel, stepping through each intermediate version.

2.4.3. Operator Lifecycle Manager workflow

This guide outlines the workflow of Operator Lifecycle Manager (OLM) in OpenShift Container Platform.

2.4.3.1. Operator installation and upgrade workflow in OLM

In the Operator Lifecycle Manager (OLM) ecosystem, the following resources are used to resolve Operator installations and upgrades:

- **ClusterServiceVersion (CSV)**
- **CatalogSource**
- **Subscription**

Operator metadata, defined in CSVs, can be stored in a collection called a catalog source. OLM uses catalog sources, which use the Operator Registry API, to query for available Operators as well as upgrades for installed Operators.

**Figure 2.3. Catalog source overview**

- **Community Operators** (CatalogSource)
  - **etcd**
    - alpha
    - beta
  - **couchbase**
    - preview
    - stable
Within a catalog source, Operators are organized into *packages* and streams of updates called *channels*, which should be a familiar update pattern from OpenShift Container Platform or other software on a continuous release cycle like web browsers.

**Figure 2.4. Packages and channels in a Catalog source**

A user indicates a particular package and channel in a particular catalog source in a *subscription*, for example an `etcd` package and its `alpha` channel. If a subscription is made to a package that has not yet been installed in the namespace, the latest Operator for that package is installed.

**NOTE**

OLM deliberately avoids version comparisons, so the "latest" or "newest" Operator available from a given catalog → channel → package path does not necessarily need to be the highest version number. It should be thought of more as the *head* reference of a channel, similar to a Git repository.

Each CSV has a *replaces* parameter that indicates which Operator it replaces. This builds a graph of CSVs that can be queried by OLM, and updates can be shared between channels. Channels can be thought of as entry points into the graph of updates:
Example channels in a package

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

defaultChannel: alpha
```

For OLM to successfully query for updates, given a catalog source, package, channel, and CSV, a catalog must be able to return, unambiguously and deterministically, a single CSV that replaces the input CSV.

2.4.3.1.1. Example upgrade path

For an example upgrade scenario, consider an installed Operator corresponding to CSV version 0.1.1. OLM queries the catalog source and detects an upgrade in the subscribed channel with new CSV version 0.1.3 that replaces an older but not-installed CSV version 0.1.2, which in turn replaces the older and installed CSV version 0.1.1.

OLM walks back from the channel head to previous versions via the replaces field specified in the CSVs to determine the upgrade path 0.1.3 → 0.1.2 → 0.1.1; the direction of the arrow indicates that the former replaces the latter. OLM upgrades the Operator one version at the time until it reaches the channel head.

For this given scenario, OLM installs Operator version 0.1.2 to replace the existing Operator version 0.1.1. Then, it installs Operator version 0.1.3 to replace the previously installed Operator version 0.1.2. At this point, the installed operator version 0.1.3 matches the channel head and the upgrade is completed.

2.4.3.1.2. Skipping upgrades

The basic path for upgrades in OLM is:
- A catalog source is updated with one or more updates to an Operator.

- OLM traverses every version of the Operator until reaching the latest version the catalog source contains.

However, sometimes this is not a safe operation to perform. There will be cases where a published version of an Operator should never be installed on a cluster if it has not already, for example because a version introduces a serious vulnerability.

In those cases, OLM must consider two cluster states and provide an update graph that supports both:

- The "bad" intermediate Operator has been seen by the cluster and installed.
- The "bad" intermediate Operator has not yet been installed onto the cluster.

By shipping a new catalog and adding a *skipped* release, OLM is ensured that it can always get a single unique update regardless of the cluster state and whether it has seen the bad update yet.

**Example CSV with skipped release**

```yaml
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
```

Consider the following example of *Old CatalogSource* and *New CatalogSource*. 
This graph maintains that:

- Any Operator found in **Old CatalogSource** has a single replacement in **New CatalogSource**.
- Any Operator found in **New CatalogSource** has a single replacement in **New CatalogSource**.
- If the bad update has not yet been installed, it will never be.

### 2.4.3.1.3. Replacing multiple Operators

Creating **New CatalogSource** as described requires publishing CSVs that **replace** one Operator, but can **skip** several. This can be accomplished using the **skipRange** annotation:

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

where `<semver_range>` has the version range format supported by the **semver library**.

When searching catalogs for updates, if the head of a channel has a **skipRange** annotation and the currently installed Operator has a version field that falls in the range, OLM updates to the latest entry in the channel.

The order of precedence is:

1. Channel head in the source specified by **sourceName** on the subscription, if the other criteria for skipping are met.
2. The next Operator that replaces the current one, in the source specified by **sourceName**.
3. Channel head in another source that is visible to the subscription, if the other criteria for skipping are met.

4. The next Operator that replaces the current one in any source visible to the subscription.

**Example CSV with `skipRange`**

```yaml
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 support**

A *z-stream*, or patch release, must replace all previous z-stream releases for the same minor version. OLM does not consider major, minor, or patch versions, it just needs to build the correct graph in a catalog.

In other words, OLM must be able to take a graph as in **Old CatalogSource** and, similar to before, generate a graph as in **New CatalogSource**:

**Figure 2.7. Replacing several Operators**

![Diagram](image.png)

This graph maintains that:
- Any Operator found in Old CatalogSource has a single replacement in New CatalogSource.
- Any Operator found in New CatalogSource has a single replacement in New CatalogSource.
- Any z-stream release in Old CatalogSource will update to the latest z-stream release in New CatalogSource.
- Unavailable releases can be considered "virtual" graph nodes; their content does not need to exist, the registry just needs to respond as if the graph looks like this.

2.4.4. Operator Lifecycle Manager dependency resolution

This guide outlines dependency resolution and custom resource definition (CRD) upgrade lifecycles with Operator Lifecycle Manager (OLM) in OpenShift Container Platform.

2.4.4.1. About dependency resolution

OLM manages the dependency resolution and upgrade lifecycle of running Operators. In many ways, the problems OLM faces are similar to other operating system package managers like yum and rpm.

However, there is one constraint that similar systems do not generally have that OLM does: because Operators are always running, OLM attempts to ensure that you are never left with a set of Operators that do not work with each other.

This means that OLM must never do the following:

- Install a set of Operators that require APIs that cannot be provided.
- Update an Operator in a way that breaks another that depends upon it.

2.4.4.2. Dependencies file

The dependencies of an Operator are listed in a dependencies.yaml file in the metadata/ folder of a bundle. This file is optional and currently only used to specify explicit Operator-version dependencies.

The dependency list contains a type field for each item to specify what kind of dependency this is. There are two supported types of Operator dependencies:

- **olm.package**: This type indicates a dependency for a specific Operator version. The dependency information must include the package name and the version of the package in semver format. For example, you can specify an exact version such as 0.5.2 or a range of versions such as >0.5.1.

- **olm.gvk**: With a gvk type, the author can specify a dependency with group/version/kind (GVK) information, similar to existing CRD and API-based usage in a CSV. This is a path to enable Operator authors to consolidate all dependencies, API or explicit versions, to be in the same place.

In the following example, dependencies are specified for a Prometheus Operator and etcd CRDs:

**Example dependencies.yaml file**

```yaml
dependencies:
  - type: olm.package
    value:
      packageName: prometheus
```

OpenShift Container Platform 4.7 Operators
2.4.4.3. Dependency preferences

There can be many options that equally satisfy a dependency of an Operator. The dependency resolver in Operator Lifecycle Manager (OLM) determines which option best fits the requirements of the requested Operator. As an Operator author or user, it can be important to understand how these choices are made so that dependency resolution is clear.

2.4.4.3.1. Catalog priority

On OpenShift Container Platform cluster, OLM reads catalog sources to know which Operators are available for installation.

Example CatalogSource object

```yaml
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
```

A CatalogSource object has a **priority** field, which is used by the resolver to know how to prefer options for a dependency.

There are two rules that govern catalog preference:

- Options in higher-priority catalogs are preferred to options in lower-priority catalogs.
- Options in the same catalog as the dependent are preferred to any other catalogs.

2.4.4.3.2. Channel ordering

An Operator package in a catalog is a collection of update channels that a user can subscribe to in a OpenShift Container Platform cluster. Channels can be used to provide a particular stream of updates for a minor release (1.2, 1.3) or a release frequency (stable, fast).

It is likely that a dependency might be satisfied by Operators in the same package, but different channels. For example, version 1.2 of an Operator might exist in both the stable and fast channels.

Each package has a default channel, which is always preferred to non-default channels. If no option in the default channel can satisfy a dependency, options are considered from the remaining channels in lexicographic order of the channel name.
2.4.4.3. Order within a channel

There are almost always multiple options to satisfy a dependency within a single channel. For example, Operators in one package and channel provide the same set of APIs.

When a user creates a subscription, they indicate which channel to receive updates from. This immediately reduces the search to just that one channel. But within the channel, it is likely that many Operators satisfy a dependency.

Within a channel, newer Operators that are higher up in the update graph are preferred. If the head of a channel satisfies a dependency, it will be tried first.

2.4.4.3.4. Other constraints

In addition to the constraints supplied by package dependencies, OLM includes additional constraints to represent the desired user state and enforce resolution invariants.

2.4.4.3.4.1. Subscription constraint

A subscription constraint filters the set of Operators that can satisfy a subscription. Subscriptions are user-supplied constraints for the dependency resolver. They declare the intent to either install a new Operator if it is not already on the cluster, or to keep an existing Operator updated.

2.4.4.3.4.2. Package constraint

Within a namespace, no two Operators may come from the same package.

2.4.4.4. CRD upgrades

OLM upgrades a custom resource definition (CRD) immediately if it is owned by a singular cluster service version (CSV). If a CRD is owned by multiple CSVs, then the CRD is upgraded when it has satisfied all of the following backward compatible conditions:

- All existing serving versions in the current CRD are present in the new CRD.
- All existing instances, or custom resources, that are associated with the serving versions of the CRD are valid when validated against the validation schema of the new CRD.

Additional resources

- Adding a new CRD version
- Deprecating or removing a CRD version

2.4.4.5. Dependency best practices

When specifying dependencies, there are best practices you should consider.

Depend on APIs or a specific version range of Operators

Operators can add or remove APIs at any time; always specify an `olm.gvk` dependency on any APIs your Operators requires. The exception to this is if you are specifying `olm.package` constraints instead.

Set a minimum version
The Kubernetes documentation on API changes describes what changes are allowed for Kubernetes-style Operators. These versioning conventions allow an Operator to update an API without bumping the API version, as long as the API is backwards-compatible. For Operator dependencies, this means that knowing the API version of a dependency might not be enough to ensure the dependent Operator works as intended.

For example:

- TestOperator v1.0.0 provides v1alpha1 API version of the **MyObject** resource.
- TestOperator v1.0.1 adds a new field `spec.newfield` to **MyObject**, but still at v1alpha1.

Your Operator might require the ability to write `spec.newfield` into the **MyObject** resource. An `olm.gvk` constraint alone is not enough for OLM to determine that you need TestOperator v1.0.1 and not TestOperator v1.0.0.

Whenever possible, if a specific Operator that provides an API is known ahead of time, specify an additional `olm.package` constraint to set a minimum.

### Omit a maximum version or allow a very wide range

Because Operators provide cluster-scoped resources such as API services and CRDs, an Operator that specifies a small window for a dependency might unnecessarily constrain updates for other consumers of that dependency. Whenever possible, do not set a maximum version. Alternatively, set a very wide semantic range to prevent conflicts with other Operators. For example, `>1.0.0 <2.0.0`.

Unlike with conventional package managers, Operator authors explicitly encode that updates are safe through channels in OLM. If an update is available for an existing subscription, it is assumed that the Operator author is indicating that it can update from the previous version. Setting a maximum version for a dependency overrides the update stream of the author by unnecessarily truncating it at a particular upper bound.

**NOTE**

Cluster administrators cannot override dependencies set by an Operator author.

However, maximum versions can and should be set if there are known incompatibilities that must be avoided. Specific versions can be omitted with the version range syntax, for example `> 1.0.0 !1.2.1`.

### Additional resources

- Kubernetes documentation: [Changing the API](#)

### 2.4.4.6. Dependency caveats

When specifying dependencies, there are caveats you should consider.

**No compound constraints (AND)**

There is currently no method for specifying an AND relationship between constraints. In other words, there is no way to specify that one Operator depends on another Operator that both provides a given API and has version `>1.1.0`. This means that when specifying a dependency such as:
It would be possible for OLM to satisfy this with two Operators: one that provides EtcdCluster and one that has version \( >3.1.0 \). Whether that happens, or whether an Operator is selected that satisfies both constraints, depends on the ordering that potential options are visited. Dependency preferences and ordering options are well-defined and can be reasoned about, but to exercise caution, Operators should stick to one mechanism or the other.

### Cross-namespace compatibility

OLM performs dependency resolution at the namespace scope. It is possible to get into an update deadlock if updating an Operator in one namespace would be an issue for an Operator in another namespace, and vice-versa.

### 2.4.4.7. Example dependency resolution scenarios

In the following examples, a provider is an Operator which "owns" a CRD or API service.

#### Example: Deprecating dependent APIs

A and B are APIs (CRDs):

- The provider of A depends on B.
- The provider of B has a subscription.
- The provider of B updates to provide C but deprecates B.

This results in:

- B no longer has a provider.
- A no longer works.

This is a case OLM prevents with its upgrade strategy.

#### Example: Version deadlock

A and B are APIs:

- The provider of A requires B.
- The provider of B requires A.
- The provider of A updates to (provide A2, require B2) and deprecate A.
- The provider of B updates to (provide B2, require A2) and deprecate B.

If OLM attempts to update A without simultaneously updating B, or vice-versa, it is unable to progress to new versions of the Operators, even though a new compatible set can be found.
This is another case OLM prevents with its upgrade strategy.

2.4.5. Operator groups

This guide outlines the use of Operator groups with Operator Lifecycle Manager (OLM) in OpenShift Container Platform.

2.4.5.1. About Operator groups

An Operator group, defined by the OperatorGroup resource, provides multitenant configuration to OLM-installed Operators. An Operator group selects target namespaces in which to generate required RBAC access for its member Operators.

The set of target namespaces is provided by a comma-delimited string stored in the olm.targetNamespaces annotation of a cluster service version (CSV). This annotation is applied to the CSV instances of member Operators and is projected into their deployments.

2.4.5.2. Operator group membership

An Operator is considered a member of an Operator group if the following conditions are true:

- The CSV of the Operator exists in the same namespace as the Operator group.
- The install modes in the CSV of the Operator support the set of namespaces targeted by the Operator group.

An install mode in a CSV consists of an InstallModeType field and a boolean Supported field. The spec of a CSV can contain a set of install modes of four distinct InstallModeTypes:

<table>
<thead>
<tr>
<th>InstallModeType</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>OwnNamespace</td>
<td>The Operator can be a member of an Operator group that selects its own namespace.</td>
</tr>
<tr>
<td>SingleNamespace</td>
<td>The Operator can be a member of an Operator group that selects one namespace.</td>
</tr>
<tr>
<td>MultiNamespace</td>
<td>The Operator can be a member of an Operator group that selects more than one namespace.</td>
</tr>
<tr>
<td>AllNamespaces</td>
<td>The Operator can be a member of an Operator group that selects all namespaces (target namespace set is the empty string &quot;&quot;).</td>
</tr>
</tbody>
</table>

NOTE

If the spec of a CSV omits an entry of InstallModeType, then that type is considered unsupported unless support can be inferred by an existing entry that implicitly supports it.

2.4.5.3. Target namespace selection
You can explicitly name the target namespace for an Operator group using the `spec.targetNamespaces` parameter:

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

You can alternatively specify a namespace using a label selector with the `spec.selector` parameter:

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

**IMPORTANT**

Listing multiple namespaces via `spec.targetNamespaces` or use of a label selector via `spec.selector` is not recommended, as the support for more than one target namespace in an Operator group will likely be removed in a future release.

If both `spec.targetNamespaces` and `spec.selector` are defined, `spec.selector` is ignored. Alternatively, you can omit both `spec.selector` and `spec.targetNamespaces` to specify a global Operator group, which selects all namespaces:

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

The resolved set of selected namespaces is shown in the `status.namespaces` parameter of an Operator group. The `status.namespace` of a global Operator group contains the empty string (""), which signals to a consuming Operator that it should watch all namespaces.

### 2.4.5.4. Operator group CSV annotations

Member CSVs of an Operator group have the following annotations:

<table>
<thead>
<tr>
<th>Annotation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>oim.operatorGroup=&lt;group_name&gt;</td>
<td>Contains the name of the Operator group.</td>
</tr>
<tr>
<td>Annotation</td>
<td>Description</td>
</tr>
<tr>
<td>----------------------------------</td>
<td>---------------------------------------------------------------</td>
</tr>
<tr>
<td>olm.operatorNamespace=</td>
<td>Contains the namespace of the Operator group.</td>
</tr>
<tr>
<td>&lt;group_namespace&gt;</td>
<td></td>
</tr>
<tr>
<td>olm.targetNamespaces=</td>
<td>Contains a comma-delimited string that lists the</td>
</tr>
<tr>
<td>&lt;target_namespaces&gt;</td>
<td>target namespace selection of the Operator group.</td>
</tr>
</tbody>
</table>

**NOTE**

All annotations except `olm.targetNamespaces` are included with copied CSVs. Omitting the `olm.targetNamespaces` annotation on copied CSVs prevents the duplication of target namespaces between tenants.

### 2.4.5.5. Provided APIs annotation

A group/version/kind (GVK) is a unique identifier for a Kubernetes API. Information about what GVKs are provided by an Operator group are shown in an `olm.providedAPIs` annotation. The value of the annotation is a string consisting of `<kind>.<version>.<group>` delimited with commas. The GVKs of CRDs and API services provided by all active member CSVs of an Operator group are included.

Review the following example of an `OperatorGroup` object with a single active member CSV that provides the `PackageManifest` resource:

```yaml
apiVersion: operators.coreos.com/v1
description: 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. Role-based access control

When an Operator group is created, three cluster roles are generated. Each contains a single aggregation rule with a cluster role selector set to match a label, as shown below:
The following RBAC resources are generated when a CSV becomes an active member of an Operator group, as long as the CSV is watching all namespaces with the `AllNamespaces` install mode and is not in a failed state with reason `InterOperatorGroupOwnerConflict`:

- Cluster roles for each API resource from a CRD
- Cluster roles for each API resource from an API service
- Additional roles and role bindings

**Table 2.5. Cluster roles generated for each API resource from a CRD**

<table>
<thead>
<tr>
<th>Cluster role</th>
<th>Settings</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>&lt;operatorgroup_name&gt;-admin</code></td>
<td>olm.opgroup.permissions/aggregate-to-admin: <code>&lt;operatorgroup_name&gt;</code></td>
</tr>
<tr>
<td><code>&lt;operatorgroup_name&gt;-edit</code></td>
<td>olm.opgroup.permissions/aggregate-to-edit: <code>&lt;operatorgroup_name&gt;</code></td>
</tr>
<tr>
<td><code>&lt;operatorgroup_name&gt;-view</code></td>
<td>olm.opgroup.permissions/aggregate-to-view: <code>&lt;operatorgroup_name&gt;</code></td>
</tr>
<tr>
<td>Cluster role</td>
<td>Settings</td>
</tr>
<tr>
<td>--------------------------------------------------</td>
<td>----------</td>
</tr>
</tbody>
</table>
| `<kind>.<group>-<version>-edit`                  | Verbs on `<kind>`:  
  - create  
  - update  
  - patch  
  - delete  
Aggregation labels:  
  - `rbac.authorization.k8s.io/aggregate-to-edit: true`  
  - `olm.opgroup.permissions/aggregate-to-edit: <operatorgroup_name>` |
| `<kind>.<group>-<version>-view`                   | Verbs on `<kind>`:  
  - get  
  - list  
  - watch  
Aggregation labels:  
  - `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  
Aggregation labels:  
  - `rbac.authorization.k8s.io/aggregate-to-view: true`  
  - `olm.opgroup.permissions/aggregate-to-view: <operatorgroup_name>` |

Table 2.6. Cluster roles generated for each API resource from an API service
<table>
<thead>
<tr>
<th>Cluster role</th>
<th>Settings</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;kind&gt;.&lt;group&gt;-&lt;version&gt;-admin</td>
<td><img src="#" alt="Verbs on &lt;kind&gt;:" /></td>
</tr>
<tr>
<td></td>
<td><img src="#" alt="Aggregation labels:" /></td>
</tr>
<tr>
<td></td>
<td><img src="#" alt="rbac.authorization.k8s.io/aggregate-to-admin: true" /></td>
</tr>
<tr>
<td></td>
<td><img src="#" alt="olm.opgroup.permissions/aggregate-to-admin: &lt;operatorgroup_name&gt;" /></td>
</tr>
<tr>
<td>&lt;kind&gt;.&lt;group&gt;-&lt;version&gt;-edit</td>
<td><img src="#" alt="Verbs on &lt;kind&gt;:" /></td>
</tr>
<tr>
<td></td>
<td><img src="#" alt="Aggregation labels:" /></td>
</tr>
<tr>
<td></td>
<td><img src="#" alt="rbac.authorization.k8s.io/aggregate-to-edit: true" /></td>
</tr>
<tr>
<td></td>
<td><img src="#" alt="olm.opgroup.permissions/aggregate-to-edit: &lt;operatorgroup_name&gt;" /></td>
</tr>
<tr>
<td>&lt;kind&gt;.&lt;group&gt;-&lt;version&gt;-view</td>
<td><img src="#" alt="Verbs on &lt;kind&gt;:" /></td>
</tr>
<tr>
<td></td>
<td><img src="#" alt="Aggregation labels:" /></td>
</tr>
<tr>
<td></td>
<td><img src="#" alt="rbac.authorization.k8s.io/aggregate-to-view: true" /></td>
</tr>
<tr>
<td></td>
<td><img src="#" alt="olm.opgroup.permissions/aggregate-to-view: &lt;operatorgroup_name&gt;" /></td>
</tr>
</tbody>
</table>

Additional roles and role bindings

- If the CSV defines exactly one target namespace that contains `*`, then a cluster role and corresponding cluster role binding are generated for each permission defined in the `permissions` field of the CSV. All resources generated are given the `olm.owner: <csv_name>` and `olm.owner.namespace: <csv_namespace>` labels.
If the CSV does not define exactly one target namespace that contains `*`, then all roles and role bindings in the Operator namespace with the `olm.owner: <csv_name>` and `olm.owner.namespace: <csv_namespace>` labels are copied into the target namespace.

### 2.4.5.7. Copied CSVs

OLM creates copies of all active member CSVs of an Operator group in each of the target namespaces of that Operator group. The purpose of a copied CSV is to tell users of a target namespace that a specific Operator is configured to watch resources created there.

Copied CSVs have a status reason **Copied** and are updated to match the status of their source CSV. The `olm.targetNamespaces` annotation is stripped from copied CSVs before they are created on the cluster. Omitting the target namespace selection avoids the duplication of target namespaces between tenants.

Copied CSVs are deleted when their source CSV no longer exists or the Operator group that their source CSV belongs to no longer targets the namespace of the copied CSV.

### 2.4.5.8. Static Operator groups

An Operator group is **static** if its `spec.staticProvidedAPIs` field is set to **true**. As a result, OLM does not modify the `olm.providedAPIs` annotation of an Operator group, which means that it can be set in advance. This is useful when a user wants to use an Operator group to prevent resource contention in a set of namespaces but does not have active member CSVs that provide the APIs for those resources.

Below is an example of an Operator group that protects Prometheus resources in all namespaces with the `something.cool.io/cluster-monitoring: "true"` annotation:

```yaml
apiVersion: operators.coreos.com/v1
kind: OperatorGroup
metadata:
  name: cluster-monitoring
  namespace: cluster-monitoring
  annotations:
    olm.providedAPIs:
spec:
  staticProvidedAPIs: true
  selector:
    matchLabels:
      something.cool.io/cluster-monitoring: "true"
```

### 2.4.5.9. Operator group intersection

Two Operator groups are said to have **intersecting provided APIs** if the intersection of their target namespace sets is not an empty set and the intersection of their provided API sets, defined by `olm.providedAPIs` annotations, is not an empty set.

A potential issue is that Operator groups with intersecting provided APIs can compete for the same resources in the set of intersecting namespaces.
NOTE

When checking intersection rules, an Operator group namespace is always included as part of its selected target namespaces.

Rules for intersection
Each time an active member CSV synchronizes, OLM queries the cluster for the set of intersecting provided APIs between the Operator group of the CSV and all others. OLM then checks if that set is an empty set:

- If true and the CSV’s provided APIs are a subset of the Operator group’s:
  - Continue transitioning.

- If true and the CSV’s provided APIs are not a subset of the Operator group’s:
  - If the Operator group is static:
    - Clean up any deployments that belong to the CSV.
    - Transition the CSV to a failed state with status reason CannotModifyStaticOperatorGroupProvidedAPIs.
  - If the Operator group is not static:
    - Replace the Operator group’s olm.providedAPIs annotation with the union of itself and the CSV’s provided APIs.

- If false and the CSV’s provided APIs are not a subset of the Operator group’s:
  - Clean up any deployments that belong to the CSV.
  - Transition the CSV to a failed state with status reason InterOperatorGroupOwnerConflict.

- If false and the CSV’s provided APIs are a subset of the Operator group’s:
  - If the Operator group is static:
    - Clean up any deployments that belong to the CSV.
    - Transition the CSV to a failed state with status reason CannotModifyStaticOperatorGroupProvidedAPIs.
  - If the Operator group is not static:
    - Replace the Operator group’s olm.providedAPIs annotation with the difference between itself and the CSV’s provided APIs.

NOTE

Failure states caused by Operator groups are non-terminal.

The following actions are performed each time an Operator group synchronizes:

- The set of provided APIs from active member CSVs is calculated from the cluster. Note that copied CSVs are ignored.
• The cluster set is compared to `olm.providedAPIs`, and if `olm.providedAPIs` contains any extra APIs, then those APIs are pruned.

• All CSVs that provide the same APIs across all namespaces are requeued. This notifies conflicting CSVs in intersecting groups that their conflict has possibly been resolved, either through resizing or through deletion of the conflicting CSV.

2.4.5.10. Limitations for multi-tenant Operator management

OpenShift Container Platform provides limited support for simultaneously installing different variations of an Operator on a cluster. Operators are control plane extensions. All tenants, or namespaces, share the same control plane of a cluster. Therefore, tenants in a multi-tenant environment also have to share Operators.

The Operator Lifecycle Manager (OLM) installs Operators multiple times in different namespaces. One constraint of this is that the Operator’s API versions must be the same.

Different major versions of an Operator often have incompatible custom resource definitions (CRDs). This makes it difficult to quickly verify OLMs.

2.4.5.10.1. Additional resources

• Allowing non-cluster administrators to install Operators

2.4.5.11. Troubleshooting Operator groups

Membership

• If more than one Operator group exists in a single namespace, any CSV created in that namespace transitions to a failure state with the reason `TooManyOperatorGroups`. CSVs in a failed state for this reason transition to pending after the number of Operator groups in their namespaces reaches one.

• If the install modes of a CSV do not support the target namespace selection of the Operator group in its namespace, the CSV transitions to a failure state with the reason `UnsupportedOperatorGroup`. CSVs in a failed state for this reason transition to pending after either the target namespace selection of the Operator group changes to a supported configuration, or the install modes of the CSV are modified to support the target namespace selection.

2.4.6. Operator conditions

This guide outlines how Operator Lifecycle Manager (OLM) uses Operator conditions.

2.4.6.1. About Operator conditions

As part of its role in managing the lifecycle of an Operator, Operator Lifecycle Manager (OLM) infers the state of an Operator from the state of Kubernetes resources that define the Operator. While this approach provides some level of assurance that an Operator is in a given state, there are many instances where an Operator might need to communicate information to OLM that could not be inferred otherwise. This information can then be used by OLM to better manage the lifecycle of the Operator.

OLM provides a custom resource definition (CRD) called `OperatorCondition` that allows Operators to communicate conditions to OLM. There are a set of supported conditions that influence management of the Operator by OLM when present in the `Status.Conditions` array of an `OperatorCondition`
2.4.6.2. Supported conditions

Operator Lifecycle Manager (OLM) supports the following Operator conditions.

2.4.6.2.1. Upgradeable condition

The **Upgradeable** Operator condition prevents an existing cluster service version (CSV) from being replaced by a newer version of the CSV. This condition is useful when:

- An Operator is about to start a critical process and should not be upgraded until the process is completed.
- An Operator is performing a migration of custom resources (CRs) that must be completed before the Operator is ready to be upgraded.

**Example Upgradeable Operator condition**

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

1. Name of the condition.
2. A *False* value indicates the Operator is not ready to be upgraded. OLM prevents a CSV that replaces the existing CSV of the Operator from leaving the **Pending** phase.

2.4.6.3. Additional resources

- Managing Operator conditions
- Enabling Operator conditions

2.4.7. Operator Lifecycle Manager metrics

2.4.7.1. Exposed metrics

Operator Lifecycle Manager (OLM) exposes certain OLM-specific resources for use by the Prometheus-based OpenShift Container Platform cluster monitoring stack.

**Table 2.7. Metrics exposed by OLM**
## Name Description

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>catalog_source_count</strong></td>
<td>Number of catalog sources.</td>
</tr>
<tr>
<td><strong>csv_abnormal</strong></td>
<td>When reconciling a cluster service version (CSV), present whenever a CSV version is in any state other than <em>Succeeded</em>, for example when it is not installed. Includes the <em>name</em>, <em>namespace</em>, <em>phase</em>, <em>reason</em>, and <em>version</em> labels. A Prometheus alert is created when this metric is present.</td>
</tr>
<tr>
<td><strong>csv_count</strong></td>
<td>Number of CSVs successfully registered.</td>
</tr>
<tr>
<td><strong>csv_succeeded</strong></td>
<td>When reconciling a CSV, represents whether a CSV version is in a <em>Succeeded</em> state (value 1) or not (value 0). Includes the <em>name</em>, <em>namespace</em>, and <em>version</em> labels.</td>
</tr>
<tr>
<td><strong>csv_upgrade_count</strong></td>
<td>Monotonic count of CSV upgrades.</td>
</tr>
<tr>
<td><strong>install_plan_count</strong></td>
<td>Number of install plans.</td>
</tr>
<tr>
<td><strong>subscription_count</strong></td>
<td>Number of subscriptions.</td>
</tr>
<tr>
<td><strong>subscription_sync_total</strong></td>
<td>Monotonic count of subscription syncs. Includes the <em>channel</em>, <em>installed</em> CSV, and subscription <em>name</em> labels.</td>
</tr>
</tbody>
</table>

### 2.4.8. Webhook management in Operator Lifecycle Manager

Webhooks allow Operator authors to intercept, modify, and accept or reject resources before they are saved to the object store and handled by the Operator controller. Operator Lifecycle Manager (OLM) can manage the lifecycle of these webhooks when they are shipped alongside your Operator.

See [Defining cluster service versions (CSVs)](chapter2) for details on how an Operator developer can define webhooks for their Operator, as well as considerations when running on OLM.

#### 2.4.8.1. Additional resources

- Types of webhook admission plug-ins
- Kubernetes documentation:
  - Validating admission webhooks
  - Mutating admission webhooks
  - Conversion webhooks

### 2.5. UNDERSTANDING OPERATORHUB
2.5.1. About OperatorHub

OperatorHub is the web console interface in OpenShift Container Platform that cluster administrators use to discover and install Operators. With one click, an Operator can be pulled from its off-cluster source, installed and subscribed on the cluster, and made ready for engineering teams to self-service manage the product across deployment environments using Operator Lifecycle Manager (OLM).

Cluster administrators can choose from catalogs grouped into the following categories:

<table>
<thead>
<tr>
<th>Category</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Red Hat Operators</td>
<td>Red Hat products packaged and shipped by Red Hat. Supported by Red Hat.</td>
</tr>
<tr>
<td>Certified Operators</td>
<td>Products from leading independent software vendors (ISVs). Red Hat partners with ISVs to package and ship. Supported by the ISV.</td>
</tr>
<tr>
<td>Red Hat Marketplace</td>
<td>Certified software that can be purchased from Red Hat Marketplace.</td>
</tr>
<tr>
<td>Community Operators</td>
<td>Optionally-visible software maintained by relevant representatives in the <code>operator-framework/community-operators</code> GitHub repository. No official support.</td>
</tr>
<tr>
<td>Custom Operators</td>
<td>Operators you add to the cluster yourself. If you have not added any custom Operators, the Custom category does not appear in the web console on your OperatorHub.</td>
</tr>
</tbody>
</table>

Operators on OperatorHub are packaged to run on OLM. This includes a YAML file called a cluster service version (CSV) containing all of the CRDs, RBAC rules, deployments, and container images required to install and securely run the Operator. It also contains user-visible information like a description of its features and supported Kubernetes versions.

The Operator SDK can be used to assist developers packaging their Operators for use on OLM and OperatorHub. If you have a commercial application that you want to make accessible to your customers, get it included using the certification workflow provided on the Red Hat Partner Connect portal at connect.redhat.com.

2.5.2. OperatorHub architecture

The OperatorHub UI component is driven by the Marketplace Operator by default on OpenShift Container Platform in the `openshift-marketplace` namespace.

2.5.2.1. OperatorHub custom resource

The Marketplace Operator manages an OperatorHub custom resource (CR) named `cluster` that manages the default `CatalogSource` objects provided with OperatorHub. You can modify this resource to enable or disable the default catalogs, which is useful when configuring OpenShift Container Platform in restricted network environments.

Example OperatorHub custom resource

```yaml
apiVersion: config.openshift.io/v1
kind: OperatorHub
metadata:
```
disableAllDefaultSources is an override that controls availability of all default catalogs that are configured by default during an OpenShift Container Platform installation.

Disable default catalogs individually by changing the disabled parameter value per source.

2.5.3. Additional resources

- Catalog source
- About the Operator SDK
- Defining cluster service versions (CSVs)
- Operator installation and upgrade workflow in OLM
- Red Hat Partner Connect
- Red Hat Marketplace

2.6. RED HAT-PROVIDED OPERATOR CATALOGS

2.6.1. About Operator catalogs

An Operator catalog is a repository of metadata that Operator Lifecycle Manager (OLM) can query to discover and install Operators and their dependencies on a cluster. OLM always installs Operators from the latest version of a catalog. As of OpenShift Container Platform 4.6, Red Hat-provided catalogs are distributed using index images.

An index image, based on the Operator Bundle Format, is a containerized snapshot of a catalog. It is an immutable artifact that contains the database of pointers to a set of Operator manifest content. A catalog can reference an index image to source its content for OLM on the cluster.

NOTE

Starting in OpenShift Container Platform 4.6, index images provided by Red Hat replace the App Registry catalog images, based on the deprecated Package Manifest Format, that are distributed for previous versions of OpenShift Container Platform 4. While App Registry catalog images are not distributed by Red Hat for OpenShift Container Platform 4.6 and later, custom catalog images based on the Package Manifest Format are still supported.
As catalogs are updated, the latest versions of Operators change, and older versions may be removed or altered. In addition, when OLM runs on an OpenShift Container Platform cluster in a restricted network environment, it is unable to access the catalogs directly from the Internet to pull the latest content.

As a cluster administrator, you can create your own custom index image, either based on a Red Hat-provided catalog or from scratch, which can be used to source the catalog content on the cluster. Creating and updating your own index image provides a method for customizing the set of Operators available on the cluster, while also avoiding the aforementioned restricted network environment issues.

**IMPORTANT**

When creating custom catalog images, previous versions of OpenShift Container Platform 4 required using the `oc adm catalog build` command, which has been deprecated for several releases. With the availability of Red Hat-provided index images starting in OpenShift Container Platform 4.6, catalog builders should start switching to using the `opm index` command to manage index images before the `oc adm catalog build` command is removed in a future release.

Additional resources

- Managing custom catalogs
- Using Operator Lifecycle Manager on restricted networks

### 2.6.2. About Red Hat-provided Operator catalogs

The following Operator catalogs are distributed by Red Hat:

<table>
<thead>
<tr>
<th>Catalog</th>
<th>Index image</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>certified-operators</td>
<td>registry.redhat.io/redhat/certified-operator-index:v4.7</td>
<td>Products from leading independent software vendors (ISVs). Red Hat partners with ISVs to package and ship. Supported by the ISV.</td>
</tr>
<tr>
<td>redhat-marketplace</td>
<td>registry.redhat.io/redhat/redhat-marketplace-index:v4.7</td>
<td>Certified software that can be purchased from Red Hat Marketplace.</td>
</tr>
<tr>
<td>community-operators</td>
<td>registry.redhat.io/redhat/community-operator-index:v4.7</td>
<td>Software maintained by relevant representatives in the operator-framework/community-operators GitHub repository. No official support.</td>
</tr>
</tbody>
</table>
2.7. CRDS

2.7.1. Extending the Kubernetes API with custom resource definitions

Operators use the Kubernetes extension mechanism, custom resource definitions (CRDs), so that custom objects managed by the Operator look and act just like the built-in, native Kubernetes objects. This guide describes how cluster administrators can extend their OpenShift Container Platform cluster by creating and managing CRDs.

2.7.1.1. Custom resource definitions

In the Kubernetes API, a resource is an endpoint that stores a collection of API objects of a certain kind. For example, the built-in Pods resource contains a collection of Pod objects.

A custom resource definition (CRD) object defines a new, unique object type, called a kind, in the cluster and lets the Kubernetes API server handle its entire lifecycle.

Custom resource (CR) objects are created from CRDs that have been added to the cluster by a cluster administrator, allowing all cluster users to add the new resource type into projects.

When a cluster administrator adds a new CRD to the cluster, the Kubernetes API server reacts by creating a new RESTful resource path that can be accessed by the entire cluster or a single project (namespace) and begins serving the specified CR.

Cluster administrators that want to grant access to the CR to other users can use cluster role aggregation to grant access to users with the admin, edit, or view default cluster roles. Cluster role aggregation allows the insertion of custom policy rules into these cluster roles. This behavior integrates the new resource into the RBAC policy of the cluster as if it was a built-in resource.

Operators in particular make use of CRDs by packaging them with any required RBAC policy and other software-specific logic. Cluster administrators can also add CRDs manually to the cluster outside of the lifecycle of an Operator, making them available to all users.

NOTE

While only cluster administrators can create CRDs, developers can create the CR from an existing CRD if they have read and write permission to it.

2.7.1.2. Creating a custom resource definition

To create custom resource (CR) objects, cluster administrators must first create a custom resource definition (CRD).

Prerequisites

- Access to an OpenShift Container Platform cluster with cluster-admin user privileges.

Procedure

To create a CRD:

1. Create a YAML file that contains the following field types:

   Example YAML file for a CRD
Use the **apiextensions.k8s.io/v1** API.

Specify a name for the definition. This must be in the `<plural-name>.<group>` format using the values from the `group` and `plural` fields.

Specify a group name for the API. An API group is a collection of objects that are logically related. For example, all batch objects like `Job` or `ScheduledJob` could be in the batch API group (such as `batch.api.example.com`). A good practice is to use a fully-qualified-domain name (FQDN) of your organization.

Specify a version name to be used in the URL. Each API group can exist in multiple versions, for example `v1alpha`, `v1beta`, `v1`.

Specify whether the custom objects are available to a project (**Namespaced**) or all projects in the cluster (**Cluster**).

Specify the plural name to use in the URL. The `plural` field is the same as a resource in an API URL.

Specify a singular name to use as an alias on the CLI and for display.

Specify the kind of objects that can be created. The type can be in CamelCase.

Specify a shorter string to match your resource on the CLI.

**NOTE**

By default, a CRD is cluster-scoped and available to all projects.

2. Create the CRD object:

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

A new RESTful API endpoint is created at:

```
/apis/<spec:group>/<spec:version>/<scope>/*/<names-plural>/...
```
For example, using the example file, the following endpoint is created:

/api/stable.example.com/v1/namespaces/*/crontabs/...

You can now use this endpoint URL to create and manage CRs. The object kind is based on the `spec.kind` field of the CRD object you created.

### 2.7.1.3. Creating cluster roles for custom resource definitions

Cluster administrators can grant permissions to existing cluster-scoped custom resource definitions (CRDs). If you use the `admin`, `edit`, and `view` default cluster roles, you can take advantage of cluster role aggregation for their rules.

**IMPORTANT**

You must explicitly assign permissions to each of these roles. The roles with more permissions do not inherit rules from roles with fewer permissions. If you assign a rule to a role, you must also assign that verb to roles that have more permissions. For example, if you grant the `get crontabs` permission to the `view` role, you must also grant it to the `edit` and `admin` roles. The `admin` or `edit` role is usually assigned to the user that created a project through the project template.

**Prerequisites**

- Create a CRD.

**Procedure**

1. Create a cluster role definition file for the CRD. The cluster role definition is a YAML file that contains the rules that apply to each cluster role. A OpenShift Container Platform controller adds the rules that you specify to the default cluster roles.

**Example YAML file for a cluster role definition**

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

Specify a name for the definition.

Specify this label to grant permissions to the admin default role.

Specify this label to grant permissions to the edit default role.

Specify the group name of the CRD.

Specify the plural name of the CRD that these rules apply to.

Specify the verbs that represent the permissions that are granted to the role. For example, apply read and write permissions to the `admin` and `edit` roles and only read permission to the `view` role.

Specify this label to grant permissions to the `view` default role.

Specify this label to grant permissions to the `cluster-reader` default role.

2. Create the cluster role:

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

2.7.1.4. Creating custom resources from a file

After a custom resource definitions (CRD) has been added to the cluster, custom resources (CRs) can be created with the CLI from a file using the CR specification.

**Prerequisites**

- CRD added to the cluster by a cluster administrator.

**Procedure**

1. Create a YAML file for the CR. In the following example definition, the `cronSpec` and `image` custom fields are set in a CR of `Kind: CronTab`. The `Kind` comes from the `spec.kind` field of the CRD object:

   **Example YAML file for a CR**

   ```yaml
   apiVersion: "stable.example.com/v1"
   kind: CronTab
   metadata:
     name: my-new-cron-object
   finalizers:
     - finalizer.stable.example.com
   ```
Specify the group name and API version (name/version) from the CRD.

Specify the type in the CRD.

Specify a name for the object.

Specify the **finalizers** for the object, if any. Finalizers allow controllers to implement conditions that must be completed before the object can be deleted.

Specify conditions specific to the type of object.

2. After you create the file, create the object:

   $ oc create -f <file_name>.yaml

### 2.7.1.5. Inspecting custom resources

You can inspect custom resource (CR) objects that exist in your cluster using the CLI.

**Prerequisites**

- A CR object exists in a namespace to which you have access.

**Procedure**

1. To get information on a specific kind of a CR, run:

   $ oc get <kind>

   For example:

   $ oc get crontab

**Example output**

<table>
<thead>
<tr>
<th>NAME</th>
<th>KIND</th>
</tr>
</thead>
<tbody>
<tr>
<td>my-new-cron-object</td>
<td>CronTab.v1.stable.example.com</td>
</tr>
</tbody>
</table>

Resource names are not case-sensitive, and you can use either the singular or plural forms defined in the CRD, as well as any short name. For example:

$ oc get crontabs

$ oc get crontab

$ oc get ct
2. You can also view the raw YAML data for a CR:

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

For example:

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

**Example output**

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

Custom data from the YAML that you used to create the object displays.

### 2.7.2. Managing resources from custom resource definitions

This guide describes how developers can manage custom resources (CRs) that come from custom resource definitions (CRDs).

#### 2.7.2.1. Custom resource definitions

In the Kubernetes API, a resource is an endpoint that stores a collection of API objects of a certain kind. For example, the built-in **Pods** resource contains a collection of **Pod** objects.

A **custom resource definition** (CRD) object defines a new, unique object type, called a **kind**, in the cluster and lets the Kubernetes API server handle its entire lifecycle.

Custom resource (CR) objects are created from CRDs that have been added to the cluster by a cluster administrator, allowing all cluster users to add the new resource type into projects.

Operators in particular make use of CRDs by packaging them with any required RBAC policy and other software-specific logic. Cluster administrators can also add CRDs manually to the cluster outside of the lifecycle of an Operator, making them available to all users.
NOTE

While only cluster administrators can create CRDs, developers can create the CR from an existing CRD if they have read and write permission to it.

2.7.2.2. Creating custom resources from a file

After a custom resource definitions (CRD) has been added to the cluster, custom resources (CRs) can be created with the CLI from a file using the CR specification.

Prerequisites

- CRD added to the cluster by a cluster administrator.

Procedure

1. Create a YAML file for the CR. In the following example definition, the `cronSpec` and `image` custom fields are set in a CR of `Kind: CronTab`. The `Kind` comes from the `spec.kind` field of the CRD object:

   **Example YAML file for a CR**

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

   - **1** Specify the group name and API version (name/version) from the CRD.
   - **2** Specify the type in the CRD.
   - **3** Specify a name for the object.
   - **4** Specify the `finalizers` for the object, if any. Finalizers allow controllers to implement conditions that must be completed before the object can be deleted.
   - **5** Specify conditions specific to the type of object.

2. After you create the file, create the object:

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

2.7.2.3. Inspecting custom resources

You can inspect custom resource (CR) objects that exist in your cluster using the CLI.

Prerequisites
- A CR object exists in a namespace to which you have access.

**Procedure**

1. To get information on a specific kind of a CR, run:

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

   For example:

   ```
   $ oc get crontab
   ```

**Example output**

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

Resource names are not case-sensitive, and you can use either the singular or plural forms defined in the CRD, as well as any short name. For example:

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

2. You can also view the raw YAML data for a CR:

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

   For example:

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

**Example output**

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

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

1 2
CHAPTER 3. USER TASKS

3.1. CREATING APPLICATIONS FROM INSTALLED OPERATORS

This guide walks developers through an example of creating applications from an installed Operator using the OpenShift Container Platform web console.

3.1.1. Creating an etcd cluster using an Operator

This procedure walks through creating a new etcd cluster using the etcd Operator, managed by Operator Lifecycle Manager (OLM).

Prerequisites

- Access to an OpenShift Container Platform 4.7 cluster.
- The etcd Operator already installed cluster-wide by an administrator.

Procedure

1. Create a new project in the OpenShift Container Platform web console for this procedure. This example uses a project called `my-etcd`.

2. Navigate to the Operators → Installed Operators page. The Operators that have been installed to the cluster by the cluster administrator and are available for use are shown here as a list of cluster service versions (CSVs). CSVs are used to launch and manage the software provided by the Operator.

   TIP

   You can get this list from the CLI using:

   ```
   $ oc get csv
   ```

3. On the Installed Operators page, click the etcd Operator to view more details and available actions.
   
   As shown under Provided APIs, this Operator makes available three new resource types, including one for an etcd Cluster (the `EtcdCluster` resource). These objects work similar to the built-in native Kubernetes ones, such as `Deployment` or `ReplicaSet`, but contain logic specific to managing etcd.

4. Create a new etcd cluster:

   a. In the etcd Cluster API box, click Create instance.

   b. The next screen allows you to make any modifications to the minimal starting template of an `EtcdCluster` object, such as the size of the cluster. For now, click Create to finalize. This triggers the Operator to start up the pods, services, and other components of the new etcd cluster.

5. Click on the example etcd cluster, then click the Resources tab to see that your project now contains a number of resources created and configured automatically by the Operator.
Verify that a Kubernetes service has been created that allows you to access the database from other pods in your project.

6. All users with the edit role in a given project can create, manage, and delete application instances (an etcd cluster, in this example) managed by Operators that have already been created in the project, in a self-service manner, just like a cloud service. If you want to enable additional users with this ability, project administrators can add the role using the following command:

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

You now have an etcd cluster that will react to failures and rebalance data as pods become unhealthy or are migrated between nodes in the cluster. Most importantly, cluster administrators or developers with proper access can now easily use the database with their applications.

### 3.2. INSTALLING OPERATORS IN YOUR NAMESPACE

If a cluster administrator has delegated Operator installation permissions to your account, you can install and subscribe an Operator to your namespace in a self-service manner.

#### 3.2.1. Prerequisites

- A cluster administrator must add certain permissions to your OpenShift Container Platform user account to allow self-service Operator installation to a namespace. See [Allowing non-cluster administrators to install Operators](#) for details.

#### 3.2.2. Operator installation with OperatorHub

OperatorHub is a user interface for discovering Operators; it works in conjunction with Operator Lifecycle Manager (OLM), which installs and manages Operators on a cluster.

As a user with the proper permissions, you can install an Operator from OperatorHub using the OpenShift Container Platform web console or CLI.

During installation, you must determine the following initial settings for the Operator:

**Installation Mode**

Choose a specific namespace in which to install the Operator.

**Update Channel**

If an Operator is available through multiple channels, you can choose which channel you want to subscribe to. For example, to deploy from the `stable` channel, if available, select it from the list.

**Approval Strategy**

You can choose automatic or manual updates.

If you choose automatic updates for an installed Operator, when a new version of that Operator is available in the selected channel, Operator Lifecycle Manager (OLM) automatically upgrades the running instance of your Operator without human intervention.

If you select manual updates, when a newer version of an Operator is available, OLM creates an update request. As a cluster administrator, you must then manually approve that update request to have the Operator updated to the new version.

- Understanding OperatorHub
3.2.3. Installing from OperatorHub using the web console

You can install and subscribe to an Operator from OperatorHub using the OpenShift Container Platform web console.

Prerequisites

- Access to an OpenShift Container Platform cluster using an account with Operator installation permissions.

Procedure

1. Navigate in the web console to the Operators → OperatorHub page.

2. Scroll or type a keyword into the Filter by keyword box to find the Operator you want. For example, type advanced to find the Advanced Cluster Management for Kubernetes Operator. You can also filter options by Infrastructure Features. For example, select Disconnected if you want to see Operators that work in disconnected environments, also known as restricted network environments.

3. Select the Operator to display additional information.

   **NOTE**
   
   Choosing a Community Operator warns that Red Hat does not certify Community Operators; you must acknowledge the warning before continuing.

4. Read the information about the Operator and click Install.

5. On the Install Operator page:

   a. Choose a specific, single namespace in which to install the Operator. The Operator will only watch and be made available for use in this single namespace.

   b. Select an Update Channel (if more than one is available).

   c. Select Automatic or Manual approval strategy, as described earlier.

6. Click Install to make the Operator available to the selected namespaces on this OpenShift Container Platform cluster.

   a. If you selected a Manual approval strategy, the upgrade status of the subscription remains Upgrading until you review and approve the install plan. After approving on the Install Plan page, the subscription upgrade status moves to Up to date.

   b. If you selected an Automatic approval strategy, the upgrade status should resolve to Up to date without intervention.

7. After the upgrade status of the subscription is Up to date, select Operators → Installed Operators to verify that the cluster service version (CSV) of the installed Operator eventually shows up. The Status should ultimately resolve to InstallSucceeded in the relevant namespace.
NOTE

For the All namespaces... installation mode, the status resolves to InstallSucceeded in the openshift-operators namespace, but the status is Copied if you check in other namespaces.

If it does not:

a. Check the logs in any pods in the openshift-operators project (or other relevant namespace if A specific namespace... installation mode was selected) on the Workloads → Pods page that are reporting issues to troubleshoot further.

3.2.4. Installing from OperatorHub using the CLI

Instead of using the OpenShift Container Platform web console, you can install an Operator from OperatorHub using the CLI. Use the oc command to create or update a Subscription object.

Prerequisites

- Access to an OpenShift Container Platform cluster using an account with Operator installation permissions.
- Install the oc command to your local system.

Procedure

1. View the list of Operators available to the cluster from OperatorHub:

   $ oc get packagemanifests -n openshift-marketplace

   **Example output**

<table>
<thead>
<tr>
<th>NAME</th>
<th>CATALOG</th>
<th>AGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>3scale-operator</td>
<td>Red Hat Operators</td>
<td>91m</td>
</tr>
<tr>
<td>advanced-cluster-management</td>
<td>Red Hat Operators</td>
<td>91m</td>
</tr>
<tr>
<td>amq7-cert-manager</td>
<td>Red Hat Operators</td>
<td>91m</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
</tr>
<tr>
<td>couchbase-enterprise-certified</td>
<td>Certified Operators</td>
<td>91m</td>
</tr>
<tr>
<td>crunchy-postgres-operator</td>
<td>Certified Operators</td>
<td>91m</td>
</tr>
<tr>
<td>crunchycertooperator</td>
<td>Certified Operators</td>
<td>91m</td>
</tr>
<tr>
<td>etcd</td>
<td>Community Operators</td>
<td>91m</td>
</tr>
<tr>
<td>jaeger</td>
<td>Community Operators</td>
<td>91m</td>
</tr>
<tr>
<td>kubefed</td>
<td>Community Operators</td>
<td>91m</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

   Note the catalog for your desired Operator.

2. Inspect your desired Operator to verify its supported install modes and available channels:

   $ oc describe packagemanifests <operator_name> -n openshift-marketplace
3. An Operator group, defined by an OperatorGroup object, selects target namespaces in which to generate required RBAC access for all Operators in the same namespace as the Operator group.

The namespace to which you subscribe the Operator must have an Operator group that matches the install mode of the Operator, either the AllNamespaces or SingleNamespace mode. If the Operator you intend to install uses the AllNamespaces, then the openshift-operators namespace already has an appropriate Operator group in place.

However, if the Operator uses the SingleNamespace mode and you do not already have an appropriate Operator group in place, you must create one.

NOTE

The web console version of this procedure handles the creation of the OperatorGroup and Subscription objects automatically behind the scenes for you when choosing SingleNamespace mode.

a. Create an OperatorGroup object YAML file, for example operatorgroup.yaml:

Example OperatorGroup object

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

b. Create the OperatorGroup object:

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

4. Create a Subscription object YAML file to subscribe a namespace to an Operator, for example sub.yaml:

Example Subscription object

```yaml
apiVersion: operators.coreos.com/v1alpha1
kind: Subscription
metadata:
  name: <subscription_name>
  namespace: openshift-operators
spec:
  channel: <channel_name>
  name: <operator_name>
  source: redhat-operators
  sourceNamespace: openshift-marketplace
  config:
    env:
      - 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

For AllNamespaces install mode usage, specify the openshift-operators namespace. Otherwise, specify the relevant single namespace for SingleNamespace install mode usage.

2 Name of the channel to subscribe to.

3 Name of the Operator to subscribe to.

4 Name of the catalog source that provides the Operator.

5 Namespace of the catalog source. Use openshift-marketplace for the default OperatorHub catalog sources.

6 The env parameter defines a list of Environment Variables that must exist in all containers in the pod created by OLM.

7 The envFrom parameter defines a list of sources to populate Environment Variables in the container.

8 The volumes parameter defines a list of Volumes that must exist on the pod created by OLM.

9 The volumeMounts parameter defines a list of VolumeMounts that must exist in all containers in the pod created by OLM. If a volumeMount references a volume that does not exist, OLM fails to deploy the Operator.

10 The tolerations parameter defines a list of Tolerations for the pod created by OLM.

11 The resources parameter defines resource constraints for all the containers in the pod created by OLM.

12 The nodeSelector parameter defines a NodeSelector for the pod created by OLM.
5. Create the **Subscription** object:

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

At this point, OLM is now aware of the selected Operator. A cluster service version (CSV) for the Operator should appear in the target namespace, and APIs provided by the Operator should be available for creation.

**Additional resources**

- Operator groups
- Channel names

### 3.2.5. Installing a specific version of an Operator

You can install a specific version of an Operator by setting the cluster service version (CSV) in a **Subscription** object.

**Prerequisites**

- Access to an OpenShift Container Platform cluster using an account with Operator installation permissions
- OpenShift CLI (**oc**) installed

**Procedure**

1. Create a **Subscription** object YAML file that subscribes a namespace to an Operator with a specific version by setting the `startingCSV` field. Set the `installPlanApproval` field to **Manual** to prevent the Operator from automatically upgrading if a later version exists in the catalog. For example, the following `sub.yaml` file can be used to install the Red Hat Quay Operator specifically to version 3.4.0:

   **Subscription with a specific starting Operator version**

   ```yaml
   apiVersion: operators.coreos.com/v1alpha1
   kind: Subscription
   metadata:
     name: quay-operator
     namespace: quay
   spec:
     channel: quay-v3.4
     installPlanApproval: Manual
     name: quay-operator
     source: redhat-operators
     sourceNamespace: openshift-marketplace
     startingCSV: quay-operator.v3.4.0
   ```

   1. Set the approval strategy to **Manual** in case your specified version is superseded by a later version in the catalog. This plan prevents an automatic upgrade to a later version and requires manual approval before the starting CSV can complete the installation.

   2. Set a specific version of an Operator CSV.
2. Create the **Subscription** object:

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

3. Manually approve the pending install plan to complete the Operator installation.

**Additional resources**

- Manually approving a pending Operator upgrade
4. ADMINISTRATOR TASKS

4.1. ADDING OPERATORS TO A CLUSTER

Cluster administrators can install Operators to an OpenShift Container Platform cluster by subscribing Operators to namespaces with OperatorHub.

4.1.1. Operator installation with OperatorHub

OperatorHub is a user interface for discovering Operators; it works in conjunction with Operator Lifecycle Manager (OLM), which installs and manages Operators on a cluster.

As a user with the proper permissions, you can install an Operator from OperatorHub using the OpenShift Container Platform web console or CLI.

During installation, you must determine the following initial settings for the Operator:

**Installation Mode**

Choose a specific namespace in which to install the Operator.

**Update Channel**

If an Operator is available through multiple channels, you can choose which channel you want to subscribe to. For example, to deploy from the **stable** channel, if available, select it from the list.

**Approval Strategy**

You can choose automatic or manual updates.

If you choose automatic updates for an installed Operator, when a new version of that Operator is available in the selected channel, Operator Lifecycle Manager (OLM) automatically upgrades the running instance of your Operator without human intervention.

If you select manual updates, when a newer version of an Operator is available, OLM creates an update request. As a cluster administrator, you must then manually approve that update request to have the Operator updated to the new version.

- Understanding OperatorHub

4.1.2. Installing from OperatorHub using the web console

You can install and subscribe to an Operator from OperatorHub using the OpenShift Container Platform web console.

**Prerequisites**

- Access to an OpenShift Container Platform cluster using an account with **cluster-admin** permissions.
- Access to an OpenShift Container Platform cluster using an account with Operator installation permissions.

**Procedure**

1. Navigate in the web console to the **Operators → OperatorHub** page.
2. Scroll or type a keyword into the Filter by keyword box to find the Operator you want. For example, type advanced to find the Advanced Cluster Management for Kubernetes Operator. You can also filter options by Infrastructure Features. For example, select Disconnected if you want to see Operators that work in disconnected environments, also known as restricted network environments.

3. Select the Operator to display additional information.

NOTE
Choosing a Community Operator warns that Red Hat does not certify Community Operators; you must acknowledge the warning before continuing.

4. Read the information about the Operator and click Install.

5. On the Install Operator page:
   a. Select one of the following:
      - All namespaces on the cluster (default) installs the Operator in the default openshift-operators namespace to watch and be made available to all namespaces in the cluster. This option is not always available.
      - A specific namespace on the cluster allows you to choose a specific, single namespace in which to install the Operator. The Operator will only watch and be made available for use in this single namespace.
   b. Choose a specific, single namespace in which to install the Operator. The Operator will only watch and be made available for use in this single namespace.
   c. Select an Update Channel (if more than one is available).
   d. Select Automatic or Manual approval strategy, as described earlier.

6. Click Install to make the Operator available to the selected namespaces on this OpenShift Container Platform cluster.
   a. If you selected a Manual approval strategy, the upgrade status of the subscription remains Upgrading until you review and approve the install plan. After approving on the Install Plan page, the subscription upgrade status moves to Up to date.
   b. If you selected an Automatic approval strategy, the upgrade status should resolve to Up to date without intervention.

7. After the upgrade status of the subscription is Up to date, select Operators → Installed Operators to verify that the cluster service version (CSV) of the installed Operator eventually shows up. The Status should ultimately resolve to InstallSucceeded in the relevant namespace.

NOTE
For the All namespaces... installation mode, the status resolves to InstallSucceeded in the openshift-operators namespace, but the status is Copied if you check in other namespaces.

If it does not:
a. Check the logs in any pods in the **openshift-operators** project (or other relevant namespace if a specific namespace... installation mode was selected) on the **Workloads → Pods** page that are reporting issues to troubleshoot further.

### 4.1.3. Installing from OperatorHub using the CLI

Instead of using the OpenShift Container Platform web console, you can install an Operator from OperatorHub using the CLI. Use the `oc` command to create or update a **Subscription** object.

**Prerequisites**

- Access to an OpenShift Container Platform cluster using an account with Operator installation permissions.
- Install the `oc` command to your local system.

**Procedure**

1. View the list of Operators available to the cluster from OperatorHub:

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

   **Example output**

<table>
<thead>
<tr>
<th>NAME</th>
<th>CATALOG</th>
<th>AGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>3scale-operator</td>
<td>Red Hat Operators</td>
<td>91m</td>
</tr>
<tr>
<td>advanced-cluster-management</td>
<td>Red Hat Operators</td>
<td>91m</td>
</tr>
<tr>
<td>amq7-cert-manager</td>
<td>Red Hat Operators</td>
<td>91m</td>
</tr>
<tr>
<td>couchbase-enterprise-certified</td>
<td>Certified Operators</td>
<td>91m</td>
</tr>
<tr>
<td>crunchy-postgres-operator</td>
<td>Certified Operators</td>
<td>91m</td>
</tr>
<tr>
<td>crunchy-postgres-operator</td>
<td>Certified Operators</td>
<td>91m</td>
</tr>
<tr>
<td>etcd</td>
<td>Community Operators</td>
<td>91m</td>
</tr>
<tr>
<td>jaeger</td>
<td>Community Operators</td>
<td>91m</td>
</tr>
<tr>
<td>kubefed</td>
<td>Community Operators</td>
<td>91m</td>
</tr>
</tbody>
</table>

   Note the catalog for your desired Operator.

2. Inspect your desired Operator to verify its supported install modes and available channels:

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

3. An Operator group, defined by an **OperatorGroup** object, selects target namespaces in which to generate required RBAC access for all Operators in the same namespace as the Operator group.

   The namespace to which you subscribe the Operator must have an Operator group that matches the install mode of the Operator, either the **AllNamespaces** or **SingleNamespace** mode. If the Operator you intend to install uses the **AllNamespaces** mode, then the **openshift-operators** namespace already has an appropriate Operator group in place.

   However, if the Operator uses the **SingleNamespace** mode and you do not already have an appropriate Operator group in place, you must create one.
NOTE

The web console version of this procedure handles the creation of the OperatorGroup and Subscription objects automatically behind the scenes for you when choosing SingleNamespace mode.

a. Create an OperatorGroup object YAML file, for example operatorgroup.yaml:

Example OperatorGroup object

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

b. Create the OperatorGroup object:

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

4. Create a Subscription object YAML file to subscribe a namespace to an Operator, for example sub.yaml:

Example Subscription object

```yaml
apiVersion: operators.coreos.com/v1alpha1
kind: Subscription
metadata:
  name: <subscription_name>
  namespace: openshift-operators
spec:
  channel: <channel_name>
  name: <operator_name>
  source: redhat-operators
  sourceNamespace: openshift-marketplace
config:
  env:
  - name: ARGSS
    value: "-v=10"
  envFrom:
  - secretRef:
    name: license-secret
volumes:
  - name: <volume_name>
    configMap:
      name: <configmap_name>
volumeMounts:
  - mountPath: <directory_name>
```
- operator: "Exists"
  resources:  
    requests:  
      memory: "64Mi"
      cpu: "250m"
    limits:  
      memory: "128Mi"
      cpu: "500m"
  nodeSelector:  
    foo: bar

For **AllNamespaces** install mode usage, specify the `openshift-operators` namespace. Otherwise, specify the relevant single namespace for **SingleNamespace** install mode usage.

2. Name of the channel to subscribe to.

3. Name of the Operator to subscribe to.

4. Name of the catalog source that provides the Operator.

5. Namespace of the catalog source. Use `openshift-marketplace` for the default OperatorHub catalog sources.

6. The `env` parameter defines a list of Environment Variables that must exist in all containers in the pod created by OLM.

7. The `envFrom` parameter defines a list of sources to populate Environment Variables in the container.

8. The `volumes` parameter defines a list of Volumes that must exist on the pod created by OLM.

9. The `volumeMounts` parameter defines a list of VolumeMounts that must exist in all containers in the pod created by OLM. If a `volumeMount` references a `volume` that does not exist, OLM fails to deploy the Operator.

10. The `tolerations` parameter defines a list of Toleraions for the pod created by OLM.

11. The `resources` parameter defines resource constraints for all the containers in the pod created by OLM.

12. The `nodeSelector` parameter defines a `NodeSelector` for the pod created by OLM.

5. Create the **Subscription** object:

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

   At this point, OLM is now aware of the selected Operator. A cluster service version (CSV) for the Operator should appear in the target namespace, and APIs provided by the Operator should be available for creation.

**Additional resources**

- [About Operator groups](#)
4.1.4. Installing a specific version of an Operator

You can install a specific version of an Operator by setting the cluster service version (CSV) in a Subscription object.

Prerequisites

- Access to an OpenShift Container Platform cluster using an account with Operator installation permissions
- OpenShift CLI (oc) installed

Procedure

1. Create a Subscription object YAML file that subscribes a namespace to an Operator with a specific version by setting the startingCSV field. Set the installPlanApproval field to Manual to prevent the Operator from automatically upgrading if a later version exists in the catalog. For example, the following sub.yaml file can be used to install the Red Hat Quay Operator specifically to version 3.4.0:

   **Subscription with a specific starting Operator version**

   ```yaml
   apiVersion: operators.coreos.com/v1alpha1
   kind: Subscription
   metadata:
     name: quay-operator
     namespace: quay
   spec:
     channel: quay-v3.4
     installPlanApproval: Manual
     name: quay-operator
     source: redhat-operators
     sourceNamespace: openshift-marketplace
     startingCSV: quay-operator.v3.4.0
   ```

   1. Set the approval strategy to Manual in case your specified version is superseded by a later version in the catalog. This plan prevents an automatic upgrade to a later version and requires manual approval before the starting CSV can complete the installation.
   2. Set a specific version of an Operator CSV.

2. Create the Subscription object:

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

3. Manually approve the pending install plan to complete the Operator installation.

Additional resources

- Manually approving a pending Operator upgrade

4.1.5. Pod placement of Operator workloads
By default, Operator Lifecycle Manager (OLM) places pods on arbitrary worker nodes when installing an Operator or deploying Operand workloads. As an administrator, you can use projects with a combination of node selectors, taints, and tolerations to control the placement of Operators and Operands to specific nodes.

Controlling pod placement of Operator and Operand workloads has the following prerequisites:

1. Determine a node or set of nodes to target for the pods per your requirements. If available, note an existing label, such as `node-role.kubernetes.io/app`, that identifies the node or nodes. Otherwise, add a label, such as `myoperator`, by using a machine set or editing the node directly. You will use this label in a later step as the node selector on your project.

2. If you want to ensure that only pods with a certain label are allowed to run on the nodes, while steering unrelated workloads to other nodes, add a taint to the node or nodes by using a machine set or editing the node directly. Use an effect that ensures that new pods that do not match the taint cannot be scheduled on the nodes. For example, a `myoperator:NoSchedule` taint ensures that new pods that do not match the taint are not scheduled onto that node, but existing pods on the node are allowed to remain.

3. Create a project that is configured with a default node selector and, if you added a taint, a matching toleration.

At this point, the project you created can be used to steer pods towards the specified nodes in the following scenarios:

**For Operator pods**

Administrators can create a **Subscription** object in the project. As a result, the Operator pods are placed on the specified nodes.

**For Operand pods**

Using an installed Operator, users can create an application in the project, which places the custom resource (CR) owned by the Operator in the project. As a result, the Operand pods are placed on the specified nodes, unless the Operator is deploying cluster-wide objects or resources in other namespaces, in which case this customized pod placement does not apply.

Additional resources

- Adding taints and tolerations manually to nodes or with machine sets
- Creating project-wide node selectors
- Creating a project with a node selector and toleration

**4.2. UPGRADING INSTALLED OPERATORS**

As a cluster administrator, you can upgrade Operators that have been previously installed using Operator Lifecycle Manager (OLM) on your OpenShift Container Platform cluster.

**4.2.1. Changing the update channel for an Operator**

The subscription of an installed Operator specifies an update channel, which is used to track and receive updates for the Operator. To upgrade the Operator to start tracking and receiving updates from a newer channel, you can change the update channel in the subscription.

The names of update channels in a subscription can differ between Operators, but the naming scheme should follow a common convention within a given Operator. For example, channel names might follow a
minor release update stream for the application provided by the Operator (1.2, 1.3) or a release frequency (stable, fast).

NOTE

Installed Operators cannot change to a channel that is older than the current channel.

If the approval strategy in the subscription is set to Automatic, the upgrade process initiates as soon as a new Operator version is available in the selected channel. If the approval strategy is set to Manual, you must manually approve pending upgrades.

Prerequisites

- An Operator previously installed using Operator Lifecycle Manager (OLM).

Procedure

1. In the Administrator perspective of the OpenShift Container Platform web console, navigate to Operators → Installed Operators.
2. Click the name of the Operator you want to change the update channel for.
3. Click the Subscription tab.
4. Click the name of the update channel under Channel.
5. Click the newer update channel that you want to change to, then click Save.
6. For subscriptions with an Automatic approval strategy, the upgrade begins automatically. Navigate back to the Operators → Installed Operators page to monitor the progress of the upgrade. When complete, the status changes to Succeeded and Up to date.
   For subscriptions with a Manual approval strategy, you can manually approve the upgrade from the Subscription tab.

4.2.2. Manually approving a pending Operator upgrade

If an installed Operator has the approval strategy in its subscription set to Manual, when new updates are released in its current update channel, the update must be manually approved before installation can begin.

Prerequisites

- An Operator previously installed using Operator Lifecycle Manager (OLM).

Procedure

1. In the Administrator perspective of the OpenShift Container Platform web console, navigate to Operators → Installed Operators.
2. Operators that have a pending upgrade display a status with Upgrade available. Click the name of the Operator you want to upgrade.
3. Click the Subscription tab. Any upgrades requiring approval are displayed next to Upgrade Status. For example, it might display 1 requires approval.
4. Click 1 requires approval then click Preview Install Plan.

5. Review the resources that are listed as available for upgrade. When satisfied, click Approve.

6. Navigate back to the Operators → Installed Operators page to monitor the progress of the upgrade. When complete, the status changes to Succeeded and Up to date.

4.3. DELETING OPERATORS FROM A CLUSTER

The following describes how to delete Operators that were previously installed using Operator Lifecycle Manager (OLM) on your OpenShift Container Platform cluster.

4.3.1. Deleting Operators from a cluster using the web console

Cluster administrators can delete installed Operators from a selected namespace by using the web console.

Prerequisites

- Access to an OpenShift Container Platform cluster web console using an account with cluster-admin permissions.

Procedure

1. From the Operators → Installed Operators page, scroll or type a keyword into the Filter by name to find the Operator you want. Then, click on it.

2. On the right-hand side of the Operator Details page, select Uninstall Operator from the Actions drop-down menu.
   An Uninstall Operator? dialog box is displayed, reminding you that:
   
   Removing the Operator will not remove any of its custom resource definitions or managed resources. If your Operator has deployed applications on the cluster or configured off-cluster resources, these will continue to run and need to be cleaned up manually.

   This action removes the Operator as well as the Operator deployments and pods, if any. Any Operands, and resources managed by the Operator, including CRDs and CRs, are not removed. The web console enables dashboards and navigation items for some Operators. To remove these after uninstalling the Operator, you might need to manually delete the Operator CRDs.

3. Select Uninstall. This Operator stops running and no longer receives updates.

4.3.2. Deleting Operators from a cluster using the CLI

Cluster administrators can delete installed Operators from a selected namespace by using the CLI.

Prerequisites

- Access to an OpenShift Container Platform cluster using an account with cluster-admin permissions.

- oc command installed on workstation.

Procedure
1. Check the current version of the subscribed Operator (for example, jaeger) in the `currentCSV` field:

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

**Example output**

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

2. Delete the subscription (for example, jaeger):

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

**Example output**

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

3. Delete the CSV for the Operator in the target namespace using the `currentCSV` value from the previous step:

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

**Example output**

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

### 4.3.3. Refreshing failing subscriptions

In Operator Lifecycle Manager (OLM), if you subscribe to an Operator that references images that are not accessible on your network, you can find jobs in the `openshift-marketplace` namespace that are failing with the following errors:

**Example output**

```
[...]
```

You can refresh a failing subscription by deleting the subscription, cluster service version (CSV), and other related objects. After recreating the subscription, OLM then reinstalls the correct version of the Operator.

**Prerequisites**

```
- You have a failing subscription that is unable to pull an inaccessible bundle image.
- You have confirmed that the correct bundle image is accessible.

**Procedure**

1. Get the names of the **Subscription** and **ClusterServiceVersion** objects from the namespace where the Operator is installed:

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

   **Example output**

<table>
<thead>
<tr>
<th>NAME</th>
<th>PACKAGE</th>
<th>SOURCE</th>
<th>CHANNEL</th>
</tr>
</thead>
<tbody>
<tr>
<td>subscription.operators.coreos.com/elasticsearch-operator</td>
<td>elasticsearch-operator</td>
<td>redhat-operators</td>
<td>5.0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>NAME</th>
<th>DISPLAY</th>
<th>VERSION</th>
</tr>
</thead>
<tbody>
<tr>
<td>replaces</td>
<td>REPLACES phase</td>
<td></td>
</tr>
<tr>
<td>clusterserviceversion.operators.coreos.com/elasticsearch-operator.5.0.0-65</td>
<td>OpenShift</td>
<td></td>
</tr>
<tr>
<td>Elasticsearch Operator 5.0.0-65</td>
<td>Succeeded</td>
<td></td>
</tr>
</tbody>
</table>

2. Delete the subscription:

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

3. Delete the cluster service version:

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

4. Get the names of any failing jobs and related config maps in the **openshift-marketplace** namespace:

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

   **Example output**

<table>
<thead>
<tr>
<th>NAME</th>
<th>COMPLETIONS</th>
<th>DURATION</th>
<th>AGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>job.batch/1de9443b6324e629ddf31fed0a853a121275806170e34c926d69e53a7fcbccb</td>
<td>1/1</td>
<td>26s</td>
<td>9m30s</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>NAME</th>
<th>DATA</th>
<th>AGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>configmap/1de9443b6324e629ddf31fed0a853a121275806170e34c926d69e53a7fcbccb</td>
<td>3</td>
<td>9m30s</td>
</tr>
</tbody>
</table>

5. Delete the job:

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

   This ensures pods that try to pull the inaccessible image are not recreated.

6. Delete the config map:
7. Reinstall the Operator using OperatorHub in the web console.

**Verification**

- Check that the Operator has been reinstalled successfully:

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

### 4.4. CONFIGURING PROXY SUPPORT IN OPERATOR LIFECYCLE MANAGER

If a global proxy is configured on the OpenShift Container Platform cluster, Operator Lifecycle Manager (OLM) automatically configures Operators that it manages with the cluster-wide proxy. However, you can also configure installed Operators to override the global proxy or inject a custom CA certificate.

**Additional resources**

- Configuring the cluster-wide proxy
- Configuring a custom PKI (custom CA certificate)

#### 4.4.1. Overriding proxy settings of an Operator

If a cluster-wide egress proxy is configured, Operators running with Operator Lifecycle Manager (OLM) inherit the cluster-wide proxy settings on their deployments. Cluster administrators can also override these proxy settings by configuring the subscription of an Operator.

**IMPORTANT**

Operators must handle setting environment variables for proxy settings in the pods for any managed Operands.

**Prerequisites**

- Access to an OpenShift Container Platform cluster using an account with **cluster-admin** permissions.

**Procedure**

1. Navigate in the web console to the **Operators → OperatorHub** page.

2. Select the Operator and click **Install**.

3. On the **Install Operator** page, modify the **Subscription** object to include one or more of the following environment variables in the **spec** section:

   - **HTTP_PROXY**
   - **HTTPS_PROXY**
   - **NO_PROXY**

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

**Subscription object with proxy setting overrides**

```yaml
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
```

**NOTE**

These environment variables can also be unset using an empty value to remove any previously set cluster-wide or custom proxy settings.

OLM handles these environment variables as a unit; if at least one of them is set, all three are considered overridden and the cluster-wide defaults are not used for the deployments of the subscribed Operator.

4. Click **Install** to make the Operator available to the selected namespaces.

5. After the CSV for the Operator appears in the relevant namespace, you can verify that custom proxy environment variables are set in the deployment. For example, using the CLI:

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

**Example output**

- name: HTTP_PROXY
  value: test_http
- name: HTTPS_PROXY
  value: test_https
- name: NO_PROXY
  value: test
image: quay.io/coreos/etcd-
4.4.2. Injecting a custom CA certificate

When a cluster administrator adds a custom CA certificate to a cluster using a config map, the Cluster Network Operator merges the user-provided certificates and system CA certificates into a single bundle. You can inject this merged bundle into your Operator running on Operator Lifecycle Manager (OLM), which is useful if you have a man-in-the-middle HTTPS proxy.

Prerequisites

- Access to an OpenShift Container Platform cluster using an account with **cluster-admin** permissions.
- Custom CA certificate added to the cluster using a config map.
- Desired Operator installed and running on OLM.

Procedure

1. Create an empty config map in the namespace where the subscription for your Operator exists and include the following label:

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

   1 Name of the config map.
   2 Requests the Cluster Network Operator to inject the merged bundle.

   After creating this config map, it is immediately populated with the certificate contents of the merged bundle.

2. Update your **Subscription** object to include a **spec.config** section that mounts the **trusted-ca** config map as a volume to each container within a pod that requires a custom CA:

   ```yaml
   apiVersion: operators.coreos.com/v1alpha1
   kind: Subscription
   metadata:
     name: my-operator
   spec:
     package: etcd
     channel: alpha
     config:
       selector:
         matchLabels:
           <labels_for_pods>
       volumes:
   ```

   1
   2
   3
- name: trusted-ca
  configMap:
    name: trusted-ca
  items:
    - key: ca-bundle.crt
      path: tls-ca-bundle.pem
  volumeMounts:
    - name: trusted-ca
      mountPath: /etc/pki/ca-trust/extracted/pem
      readOnly: true

1. Add a `config` section if it does not exist.
2. Specify labels to match pods that are owned by the Operator.
3. Create a `trusted-ca` volume.
4. `ca-bundle.crt` is required as the config map key.
5. `tls-ca-bundle.pem` is required as the config map path.
6. Create a `trusted-ca` volume mount.

4.5. VIEWING OPERATOR STATUS

Understanding the state of the system in Operator Lifecycle Manager (OLM) is important for making decisions about and debugging problems with installed Operators. OLM provides insight into subscriptions and related catalog sources regarding their state and actions performed. This helps users better understand the healthiness of their Operators.

4.5.1. Operator subscription condition types

Subscriptions can report the following condition types:

Table 4.1. Subscription condition types

<table>
<thead>
<tr>
<th>Condition</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CatalogSourcesUnhealthy</td>
<td>Some or all of the catalog sources to be used in resolution are unhealthy.</td>
</tr>
<tr>
<td>InstallPlanMissing</td>
<td>An install plan for a subscription is missing.</td>
</tr>
<tr>
<td>InstallPlanPending</td>
<td>An install plan for a subscription is pending installation.</td>
</tr>
<tr>
<td>InstallPlanFailed</td>
<td>An install plan for a subscription has failed.</td>
</tr>
</tbody>
</table>
NOTE
Default OpenShift Container Platform cluster Operators are managed by the Cluster Version Operator (CVO) and they do not have a Subscription object. Application Operators are managed by Operator Lifecycle Manager (OLM) and they have a Subscription object.

Additional resources
- Refreshing failing subscriptions

4.5.2. Viewing Operator subscription status by using the CLI
You can view Operator subscription status by using the CLI.

Prerequisites
- You have access to the cluster as a user with the cluster-admin role.
- You have installed the OpenShift CLI (oc).

Procedure
1. List Operator subscriptions:
   
   ```
   $ oc get subs -n <operator_namespace>
   ```

2. Use the oc describe command to inspect a Subscription resource:
   
   ```
   $ oc describe sub <subscription_name> -n <operator_namespace>
   ```

3. In the command output, find the Conditions section for the status of Operator subscription condition types. In the following example, the CatalogSourcesUnhealthy condition type has a status of false because all available catalog sources are healthy:

   Example output

   ```
   Conditions:
   Message: all available catalogsources are healthy
   Reason: AllCatalogSourcesHealthy
   Status: False
   Type: CatalogSourcesUnhealthy
   ```

4.5.3. Viewing Operator catalog source status by using the CLI

NOTE
Default OpenShift Container Platform cluster Operators are managed by the Cluster Version Operator (CVO) and they do not have a Subscription object. Application Operators are managed by Operator Lifecycle Manager (OLM) and they have a Subscription object.
You can view the status of an Operator catalog source by using the CLI.

Prerequisites

- You have access to the cluster as a user with the `cluster-admin` role.
- You have installed the OpenShift CLI (`oc`).

Procedure

1. List the catalog sources in a namespace. For example, you can check the `openshift-marketplace` namespace, which is used for cluster-wide catalog sources:

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

   **Example output**

<table>
<thead>
<tr>
<th>NAME</th>
<th>DISPLAY</th>
<th>TYPE</th>
<th>PUBLISHER</th>
<th>AGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>certified-operators</td>
<td>Certified Operators</td>
<td>grpc</td>
<td>Red Hat</td>
<td>55m</td>
</tr>
<tr>
<td>community-operators</td>
<td>Community Operators</td>
<td>grpc</td>
<td>Red Hat</td>
<td>55m</td>
</tr>
<tr>
<td>example-catalog</td>
<td>Example Catalog</td>
<td>grpc</td>
<td>Example Org</td>
<td>2m25s</td>
</tr>
<tr>
<td>redhat-marketplace</td>
<td>Red Hat Marketplace</td>
<td>grpc</td>
<td>Red Hat</td>
<td>55m</td>
</tr>
<tr>
<td>redhat-operators</td>
<td>Red Hat Operators</td>
<td>grpc</td>
<td>Red Hat</td>
<td>55m</td>
</tr>
</tbody>
</table>

2. Use the `oc describe` command to get more details and status about a catalog source:

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

   **Example output**

   ```
   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
   ```

   In the preceding example output, the last observed state is `TRANSIENT_FAILURE`. This state indicates that there is a problem establishing a connection for the catalog source.

3. List the pods in the namespace where your catalog source was created:

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

   **Example output**
When a catalog source is created in a namespace, a pod for the catalog source is created in that namespace. In the preceding example output, the status for the example-catalog-bwt8z pod is `ImagePullBackOff`. This status indicates that there is an issue pulling the catalog source’s index image.

4. Use the `oc describe` command to inspect a pod for more detailed information:

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

**Example output**

```
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-jyryyg2-f76d1-ggdbq-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
```

In the preceding example output, the error messages indicate that the catalog source’s index image is failing to pull successfully because of an authorization issue. For example, the index image might be stored in a registry that requires login credentials.

**Additional resources**

- Operator Lifecycle Manager concepts and resources → Catalog source
- gRPC documentation: States of Connectivity
- Accessing images for Operators from private registries
4.6. MANAGING OPERATOR CONDITIONS

As a cluster administrator, you can manage Operator conditions by using Operator Lifecycle Manager (OLM).

4.6.1. Overriding Operator conditions

As a cluster administrator, you might want to ignore a supported Operator condition reported by an Operator. When present, Operator conditions in the `Spec.Overrides` array override the conditions in the `Status.Conditions` array, allowing cluster administrators to deal with situations where an Operator is incorrectly reporting a state to Operator Lifecycle Manager (OLM).

For example, consider a known version of an Operator that always communicates that it is not upgradeable. In this instance, you might want to upgrade the Operator despite the Operator communicating that it is not upgradeable. This could be accomplished by overriding the Operator condition by adding the condition `type` and `status` to the `Spec.Overrides` array in the `OperatorCondition` resource.

Prerequisites

- An Operator with an `OperatorCondition` resource, installed using OLM.

Procedure

1. Edit the `OperatorCondition` resource for the Operator:

   $ oc edit operatorcondition <name>

2. Add a `Spec.Overrides` array to the object:

   Example Operator condition override

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

   1 Allows the cluster administrator to change the upgrade readiness to `True`. 
4.6.2. Updating your Operator to use Operator conditions

Operator Lifecycle Manager (OLM) automatically creates an OperatorCondition resource for each ClusterServiceVersion resource that it reconciles. All service accounts in the CSV are granted the RBAC to interact with the OperatorCondition owned by the Operator.

An Operator author can develop their Operator to use the operator-lib library such that, after the Operator has been deployed by OLM, it can set its own conditions. For more on writing logic to set Operator conditions as an Operator author, see the Operator SDK documentation.

4.6.2.1. Setting defaults

In an effort to remain backwards compatible, OLM treats the absence of an OperatorCondition resource as opting out of the condition. Therefore, an Operator that opts in to using Operator conditions should set default conditions before the ready probe for the pod is set to true. This provides the Operator with a grace period to update the condition to the correct state.

4.6.3. Additional resources

- Operator conditions

4.7. ALLOWING NON-CLUSTER ADMINISTRATORS TO INSTALL OPERATORS

Operators can require wide privileges to run, and the required privileges can change between versions. Operator Lifecycle Manager (OLM) runs with cluster-admin privileges. By default, Operator authors can specify any set of permissions in the cluster service version (CSV) and OLM will consequently grant it to the Operator.

Cluster administrators should take measures to ensure that an Operator cannot achieve cluster-scoped privileges and that users cannot escalate privileges using OLM. One method for locking this down requires cluster administrators auditing Operators before they are added to the cluster. Cluster administrators are also provided tools for determining and constraining which actions are allowed during an Operator installation or upgrade using service accounts.

By associating an Operator group with a service account that has a set of privileges granted to it, cluster administrators can set policy on Operators to ensure they operate only within predetermined boundaries using RBAC rules. The Operator is unable to do anything that is not explicitly permitted by those rules.

This self-sufficient, limited scope installation of Operators by non-cluster administrators means that more of the Operator Framework tools can safely be made available to more users, providing a richer experience for building applications with Operators.

4.7.1. Understanding Operator installation policy

Using Operator Lifecycle Manager (OLM), cluster administrators can choose to specify a service account for an Operator group so that all Operators associated with the group are deployed and run against the privileges granted to the service account.

The APIService and CustomResourceDefinition resources are always created by OLM using the cluster-admin role. A service account associated with an Operator group should never be granted privileges to write these resources.
If the specified service account does not have adequate permissions for an Operator that is being installed or upgraded, useful and contextual information is added to the status of the respective resource(s) so that it is easy for the cluster administrator to troubleshoot and resolve the issue.

Any Operator tied to this Operator group is now confined to the permissions granted to the specified service account. If the Operator asks for permissions that are outside the scope of the service account, the install fails with appropriate errors.

### 4.7.1.1. Installation scenarios

When determining whether an Operator can be installed or upgraded on a cluster, Operator Lifecycle Manager (OLM) considers the following scenarios:

- A cluster administrator creates a new Operator group and specifies a service account. All Operator(s) associated with this Operator group are installed and run against the privileges granted to the service account.

- A cluster administrator creates a new Operator group and does not specify any service account. OpenShift Container Platform maintains backward compatibility, so the default behavior remains and Operator installs and upgrades are permitted.

- For existing Operator groups that do not specify a service account, the default behavior remains and Operator installs and upgrades are permitted.

- A cluster administrator updates an existing Operator group and specifies a service account. OLM allows the existing Operator to continue to run with their current privileges. When such an existing Operator is going through an upgrade, it is reinstalled and run against the privileges granted to the service account like any new Operator.

- A service account specified by an Operator group changes by adding or removing permissions, or the existing service account is swapped with a new one. When existing Operators go through an upgrade, it is reinstalled and run against the privileges granted to the updated service account like any new Operator.

- A cluster administrator removes the service account from an Operator group. The default behavior remains and Operator installs and upgrades are permitted.

### 4.7.1.2. Installation workflow

When an Operator group is tied to a service account and an Operator is installed or upgraded, Operator Lifecycle Manager (OLM) uses the following workflow:

1. The given Subscription object is picked up by OLM.

2. OLM fetches the Operator group tied to this subscription.

3. OLM determines that the Operator group has a service account specified.

4. OLM creates a client scoped to the service account and uses the scoped client to install the Operator. This ensures that any permission requested by the Operator is always confined to that of the service account in the Operator group.

5. OLM creates a new service account with the set of permissions specified in the CSV and assigns it to the Operator. The Operator runs as the assigned service account.

### 4.7.2. Scoping Operator installations
To provide scoping rules to Operator installations and upgrades on Operator Lifecycle Manager (OLM), associate a service account with an Operator group.

Using this example, a cluster administrator can confine a set of Operators to a designated namespace.

**Procedure**

1. Create a new namespace:

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

2. Allocate permissions that you want the Operator(s) to be confined to. This involves creating a new service account, relevant role(s), and role binding(s).

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

   The following example grants the service account permissions to do anything in the designated namespace for simplicity. In a production environment, you should create a more fine-grained set of permissions:

   ```
   $ 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
   ```
3. Create an **OperatorGroup** object in the designated namespace. This Operator group targets the designated namespace to ensure that its tenancy is confined to it. In addition, Operator groups allow a user to specify a service account. Specify the service account created in the previous step:

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

Any Operator installed in the designated namespace is tied to this Operator group and therefore to the service account specified.

4. Create a **Subscription** object in the designated namespace to install an Operator:

```bash
$ 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>  
  sourceNamespace: <catalog_source_namespace>  
EOF
```

1. Specify a catalog source that already exists in the designated namespace or one that is in the global catalog namespace.

2. Specify a namespace where the catalog source was created.

Any Operator tied to this Operator group is confined to the permissions granted to the specified service account. If the Operator requests permissions that are outside the scope of the service account, the installation fails with relevant errors.

**4.7.2.1. Fine-grained permissions**

Operator Lifecycle Manager (OLM) uses the service account specified in an Operator group to create or update the following resources related to the Operator being installed:

- **ClusterServiceVersion**
To confine Operators to a designated namespace, cluster administrators can start by granting the following permissions to the service account:

**NOTE**

The following role is a generic example and additional rules might be required based on the specific Operator.

```yaml
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"]
  resources: ["deployments"]
  verbs: ["list", "watch", "get", "create", "update", "patch", "delete"]
- apiGroups: [""]
  resources: ["pods"]
  verbs: ["list", "watch", "get", "create", "update", "patch", "delete"]
```

1 Add permissions to create other resources, such as deployments and pods shown here.

In addition, if any Operator specifies a pull secret, the following permissions must also be added:

```yaml
kind: ClusterRole
rules:
- apiGroups: [""]
  resources: ["secrets"]
  verbs: ["get"]
---
kind: Role
rules:
- apiGroups: [""]
  resources: ["secrets"]
  verbs: ["create", "update", "patch"]
```
Required to get the secret from the OLM namespace.

4.7.3. Troubleshooting permission failures

If an Operator installation fails due to lack of permissions, identify the errors using the following procedure.

Procedure

1. Review the Subscription object. Its status has an object reference installPlanRef that points to the InstallPlan object that attempted to create the necessary [Cluster]Role[Binding] object(s) for the Operator:

   ```json
   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. Check the status of the InstallPlan object for any errors:

   ```json
   apiVersion: operators.coreos.com/v1
   kind: InstallPlan
   status:
     conditions:
       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
   ```

The error message tells you:

- The type of resource it failed to create, including the API group of the resource. In this case, it was clusterroles in the rbac.authorization.k8s.io group.

- The name of the resource.

- The type of error: is forbidden tells you that the user does not have enough permission to do the operation.
The name of the user who attempted to create or update the resource. In this case, it refers to the service account specified in the Operator group.

- The scope of the operation: **cluster scope** or not. The user can add the missing permission to the service account and then iterate.

**NOTE**

Operator Lifecycle Manager (OLM) does not currently provide the complete list of errors on the first try.

### 4.8. MANAGING CUSTOM CATALOGS

This guide describes how to work with custom catalogs for Operators packaged using either the **Bundle Format** or the legacy **Package Manifest Format** on Operator Lifecycle Manager (OLM) in OpenShift Container Platform.

**Additional resources**

- Red Hat-provided Operator catalogs

### 4.8.1. Custom catalogs using the Bundle Format

#### 4.8.1.1. Prerequisites

- Install the **opm** CLI.

#### 4.8.1.2. Creating an index image

You can create an index image using the **opm** CLI.

**Prerequisites**

- **opm** version 1.12.3+
- **podman** version 1.9.3+
- A bundle image built and pushed to a registry that supports Docker v2-2

**IMPORTANT**

The internal registry of the OpenShift Container Platform cluster cannot be used as the target registry because it does not support pushing without a tag, which is required during the mirroring process.

**Procedure**

1. Start a new index:

```bash
$ 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. Comma-separated list of bundle images to add to the index.
2. The image tag that you want the index image to have.
3. Optional: An alternative registry base image to use for serving the catalog.

2. Push the index image to a registry.
   a. If required, authenticate with your target registry:
      
      ```
      $ podman login <registry>
      ```
   b. Push the index image:
      
      ```
      $ podman push <registry>/<namespace>/test-catalog:latest
      ```

### 4.8.1.3. Creating a catalog from an index image

You can create an Operator catalog from an index image and apply it to an OpenShift Container Platform cluster for use with Operator Lifecycle Manager (OLM).

**Prerequisites**

- An index image built and pushed to a registry.

**Procedure**

1. Create a **CatalogSource** object that references your index image.
   a. Modify the following to your specifications and save it as a **catalogSource.yaml** file:

   ```yaml
   apiVersion: operators.coreos.com/v1alpha1
   kind: CatalogSource
   metadata:
     name: my-operator-catalog
     namespace: openshift-marketplace
   spec:
     sourceType: grpc
     displayName: My Operator Catalog
     publisher: <publisher_name>
     updateStrategy:
       registryPoll:
         interval: 30m
   ```

   - If you want the catalog source to be available globally to users in all namespaces, specify the **openshift-marketplace** namespace. Otherwise, you can specify a different namespace for the catalog to be scoped and available only for that namespace.
   - Specify your index image.
   - Specify your name or an organization name publishing the catalog.
Catalog sources can automatically check for new versions to keep up to date.

b. Use the file to create the `CatalogSource` object:

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

2. Verify the following resources are created successfully.

a. Check the pods:

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

Example output

<table>
<thead>
<tr>
<th>NAME</th>
<th>READY</th>
<th>STATUS</th>
<th>RESTARTS</th>
<th>AGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>my-operator-catalog-6njx6</td>
<td>1/1</td>
<td>Running</td>
<td>0</td>
<td>28s</td>
</tr>
<tr>
<td>marketplace-operator-d9f549946-96sgr</td>
<td>1/1</td>
<td>Running</td>
<td>0</td>
<td>26h</td>
</tr>
</tbody>
</table>

b. Check the catalog source:

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

Example output

<table>
<thead>
<tr>
<th>NAME</th>
<th>DISPLAY</th>
<th>TYPE</th>
<th>PUBLISHER</th>
<th>AGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>my-operator-catalog</td>
<td>My Operator Catalog</td>
<td>grpc</td>
<td></td>
<td>5s</td>
</tr>
</tbody>
</table>

c. Check the package manifest:

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

Example output

<table>
<thead>
<tr>
<th>NAME</th>
<th>CATALOG</th>
<th>AGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>jaeger-product</td>
<td>My Operator Catalog</td>
<td>93s</td>
</tr>
</tbody>
</table>

You can now install the Operators from the OperatorHub page on your OpenShift Container Platform web console.

Additional resources

- If your index image is hosted on a private registry and requires authentication, see Accessing images for Operators from private registries.

### 4.8.1.4. Updating an index image

After configuring OperatorHub to use a catalog source that references a custom index image, cluster administrators can keep the available Operators on their cluster up to date by adding bundle images to the index image.

You can update an existing index image using the `opm index add` command.
Prerequisites

- **opm** version 1.12.3+
- **podman** version 1.9.3+
- An index image built and pushed to a registry.
- An existing catalog source referencing the index image.

Procedure

1. Update the existing index by adding bundle images:

   ```bash
   $ 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. The **--bundles** flag specifies a comma-separated list of additional bundle images to add to the index.
   2. The **--from-index** flag specifies the previously pushed index.
   3. The **--tag** flag specifies the image tag to apply to the updated index image.
   4. The **--pull-tool** flag specifies the tool used to pull container images.

   where:

   **<registry>**
   Specifies the hostname of the registry, such as **quay.io** or **mirror.example.com**.

   **<namespace>**
   Specifies the namespace of the registry, such as **ocs-dev** or **abc**.

   **<new_bundle_image>**
   Specifies the new bundle image to add to the registry, such as **ocs-operator**.

   **<digest>**
   Specifies the SHA image ID, or digest, of the bundle image, such as **c7f11097a628f092d8bad148406aa0e0951094a03445f4d4bc0775431ef683a41**.

   **<existing_index_image>**
   Specifies the previously pushed image, such as **abc-redhat-operator-index**.

   **<existing_tag>**
   Specifies a previously pushed image tag, such as **4.7**.

   **<updated_tag>**
   Specifies the image tag to apply to the updated index image, such as **4.7.1**.

Example command

```bash
$ opm index add \n```
After Operator Lifecycle Manager (OLM) automatically polls the index image referenced in the catalog source at its regular interval, verify that the new packages are successfully added:

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

### 4.8.1.5. Pruning an index image

An index image, based on the Operator Bundle Format, is a containerized snapshot of an Operator catalog. You can prune an index of all but a specified list of packages, which creates a copy of the source index containing only the Operators that you want.

**Prerequisites**

- **podman** version 1.9.3+
- **grpcurl** (third-party command-line tool)
- **opm** version 1.18.0+
- Access to a registry that supports **Docker v2-2**

**IMPORTANT**

The internal registry of the OpenShift Container Platform cluster cannot be used as the target registry because it does not support pushing without a tag, which is required during the mirroring process.

**Procedure**

1. Authenticate with your target registry:

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

2. Determine the list of packages you want to include in your pruned index.
   a. Run the source index image that you want to prune in a container. For example:

   ```bash
   ```

   **Example output**
b. In a separate terminal session, use the `grpcurl` command to get a list of the packages provided by the index:

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

c. Inspect the `packages.out` file and identify which package names from this list you want to keep in your pruned index. For example:

**Example snippets of packages list**

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

d. In the terminal session where you executed the `podman run` command, press Ctrl and C to stop the container process.

3. Run the following command to prune the source index of all but the specified packages:

```
$ opm index prune
  -f registry.redhat.io/redhat/redhat-operator-index:v4.7
  -p advanced-cluster-management,jaeger-product,quay-operator
  [-i registry.redhat.io/openshift4/ose-operator-registry:v4.7]
  -t <target_registry>:<port>/<namespace>/redhat-operator-index:v4.7
```

**1.** Index to prune.

**2.** Comma-separated list of packages to keep.

**3.** Required only for IBM Power Systems and IBM Z images: Operator Registry base image with the tag that matches the target OpenShift Container Platform cluster major and minor version.

**4.** Custom tag for new index image being built.

4. Run the following command to push the new index image to your target registry:
4.8.2. Custom catalogs using the Package Manifest Format

4.8.2.1. Building a Package Manifest Format catalog image

Cluster administrators can build a custom Operator catalog image based on the Package Manifest Format to be used by Operator Lifecycle Manager (OLM). The catalog image can be pushed to a container image registry that supports Docker v2-2. For a cluster on a restricted network, this registry can be a registry that the cluster has network access to, such as a mirror registry created during a restricted network cluster installation.

IMPORTANT

The internal registry of the OpenShift Container Platform cluster cannot be used as the target registry because it does not support pushing without a tag, which is required during the mirroring process.

For this example, the procedure assumes use of a mirror registry that has access to both your network and the Internet.

NOTE

Only the Linux version of the `oc` client can be used for this procedure, because the Windows and macOS versions do not provide the `oc adm catalog build` command.

Prerequisites

- Workstation with unrestricted network access
- `oc` version 4.3.5+ Linux client
- `podman` version 1.9.3+
- Access to mirror registry that supports Docker v2-2
- If you are working with private registries, set the `REG_CREDS` environment variable to the file path of your registry credentials for use in later steps. For example, for the `podman` CLI:

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

- If you are working with private namespaces that your `quay.io` account has access to, you must set a Quay authentication token. Set the `AUTH_TOKEN` environment variable for use with the `-auth-token` flag by making a request against the login API using your `quay.io` credentials:

```bash
$ AUTH_TOKEN=$(curl -sH "Content-Type: application/json" -XPOST https://quay.io/cnr/api/v1/users/login -d '{
   "user": {
      "username": """"<quay_username>"""
   }
}')
```

```bash
$ podman push <target_registry>:<port>/<namespace>/redhat-operator-index:v4.7
```

where `<namespace>` is any existing namespace on the registry.
Procedure

1. On the workstation with unrestricted network access, authenticate with the target mirror registry:

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

2. Authenticate with `registry.redhat.io` so that the base image can be pulled during the build:

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

3. Build a catalog image based on the `redhat-operators` catalog from Quay.io, tagging and pushing it to your mirror registry:

   ```
   $ oc adm catalog build \
   --appregistry-org redhat-operators \1
   --from=registry.redhat.io/openshift4/ose-operator-registry:v4.7 \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
   ```

   **Organization (namespace) to pull from an App Registry instance.**

   **Set `--from` to the Operator Registry base image using the tag that matches the target OpenShift Container Platform cluster major and minor version.**

   **Set `--filter-by-os` to the operating system and architecture to use for the base image, which must match the target OpenShift Container Platform cluster. Valid values are `linux/amd64`, `linux/ppc64le`, and `linux/s390x`.**

   **Name your catalog image and include a tag, for example, `v1`.**

   **Optional: If required, specify the location of your registry credentials file.**

   **Optional: If you do not want to configure trust for the target registry, add the `--insecure` flag.**

   **Optional: If other application registry catalogs are used that are not public, specify a Quay authentication token.**

Example output

```
INFO[0013] loading Bundles
dir=/var/folders/st/9csxqs53l3wdn434vw4cd80000gn/T/300666084/manifests-829192605 ...
Pushed sha256:f73d42950021f9240389f99ddc5b0c7f1b533c054ba344654ff1edaf6bf827e3
to example_registry:5000/olm/redhat-operators:v1
```
Sometimes invalid manifests are accidentally introduced catalogs provided by Red Hat; when this happens, you might see some errors:

Example output with errors

```
INFO[0014] directory
dir=/var/folders/st/9cskxqs53ll3wdn4v4cd80000gn/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
```

These errors are usually non-fatal, and if the Operator package mentioned does not contain an Operator you plan to install or a dependency of one, then they can be ignored.

Additional resources

- Mirroring images for a disconnected installation

### 4.8.2.2. Mirroring a Package Manifest Format catalog image

Cluster administrators can mirror a custom Operator catalog image based on the Package Manifest Format into a registry and use a catalog source to load the content onto their cluster. For this example, the procedure uses a custom `redhat-operators` catalog image previously built and pushed to a supported registry.

**Prerequisites**

- Workstation with unrestricted network access
- A custom Operator catalog image based on the Package Manifest Format pushed to a supported registry
- `oc` version 4.3.5+
- `podman` version 1.9.3+
- Access to mirror registry that supports Docker v2-2

**IMPORTANT**

The internal registry of the OpenShift Container Platform cluster cannot be used as the target registry because it does not support pushing without a tag, which is required during the mirroring process.

- If you are working with private registries, set the `REG_CREDS` environment variable to the file path of your registry credentials for use in later steps. For example, for the `podman` CLI:

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

**Procedure**
1. The `oc adm catalog mirror` command extracts the contents of your custom Operator catalog image to generate the manifests required for mirroring. You can choose to either:

- Allow the default behavior of the command to automatically mirror all of the image content to your mirror registry after generating manifests, or

- Add the `--manifests-only` flag to only generate the manifests required for mirroring, but do not actually mirror the image content to a registry yet. This can be useful for reviewing what will be mirrored, and it allows you to make any changes to the mapping list if you only require a subset of the content. You can then use that file with the `oc image mirror` command to mirror the modified list of images in a later step.

On your workstation with unrestricted network access, run the following command:

```
$ oc adm catalog mirror
  <registry_host_name>:<port>/olm/redhat-operators:v1 \  
  <registry_host_name>:<port> \  
  [-a ${REG_CREDS}] \  
  [--insecure] \  
  [--index-filter-by-os='<platform>/<arch>'] \  
  [--manifests-only]
```

1. Specify your Operator catalog image.
2. Specify the fully qualified domain name (FQDN) for the target registry.
3. Optional: If required, specify the location of your registry credentials file.
4. Optional: If you do not want to configure trust for the target registry, add the `--insecure` flag.
5. Optional: Specify which platform and architecture of the catalog image to select when multiple variants are available. Images are passed as `<platform>/<arch>[/<variant>]`. This does not apply to images referenced by the catalog image. Valid values are `linux/amd64`, `linux/ppc64le`, and `linux/s390x`.
6. Optional: Only generate the manifests required for mirroring and do not actually mirror the image content to a registry.

**Example output**

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

1. Temporary database generated by the command.

After running the command, a `manifests-<index_image_name>-<random_number>/` directory is created in the current directory and generates the following files:
- The `catalogSource.yaml` file is a basic definition for a CatalogSource object that is pre-populated with your catalog image tag and other relevant metadata. This file can be used as is or modified to add the catalog source to your cluster.

- The `imageContentSourcePolicy.yaml` file defines an ImageContentSourcePolicy object that can configure nodes to translate between the image references stored in Operator manifests and the mirrored registry.

**NOTE**

If your cluster uses an ImageContentSourcePolicy object to configure repository mirroring, you can use only global pull secrets for mirrored registries. You cannot add a pull secret to a project.

- The `mapping.txt` file contains all of the source images and where to map them in the target registry. This file is compatible with the `oc image mirror` command and can be used to further customize the mirroring configuration.

2. If you used the `--manifests-only` flag in the previous step and want to mirror only a subset of the content:

   a. Modify the list of images in your `mapping.txt` file to your specifications. If you are unsure of the exact names and versions of the subset of images you want to mirror, use the following steps to find them:

      i. Run the `sqlite3` tool against the temporary database that was generated by the `oc adm catalog mirror` command to retrieve a list of images matching a general search query. The output helps inform how you will later edit your `mapping.txt` file.

         For example, to retrieve a list of images that are similar to the string `clusterlogging.4.3`:

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

         Refer to the previous output of the `oc adm catalog mirror` command to find the path of the database file.

         **Example output**

         ```
         image = registry.redhat.io/openshift-logging/kibana6-rhel8@sha256:aa4a8b2a00836d0e28aa6497ad90a3c116f135f382d8211e3c55f34fb36dfe61
         operatorbundle_name = clusterlogging.4.3.33-202008111029.p0
         
         image = registry.redhat.io/openshift4/ose-oauth-proxy@sha256:6b4db07f6e6c962fc96473d86c44532c93b146bbe3e311d0c348117bf759c506
         operatorbundle_name = clusterlogging.4.3.33-202008111029.p0
         ```

   ii. Use the results from the previous step to edit the `mapping.txt` file to only include the subset of images you want to mirror.

      For example, you can use the `image` values from the previous example output to find that the following matching lines exist in your `mapping.txt` file:
Matching image mappings in `mapping.txt`

```bash
registry.redhat.io/openshift-logging/kibana6-rhel8@sha256:aa4a8b2a00836d0e28aa6497ad90a3c116f135f382d8211e3c55f34fb36
dfe61=<registry_host_name>:<port>/kibana6-rhel8:a767c8f0
registry.redhat.io/openshift4/ose-oauth-proxy@sha256:6b4db07f6e6c962fc96473d86c44532c93b146bbee6e311d0c348117bf75
9c506=<registry_host_name>:<port>/openshift4-ose-oauth-proxy:3754ea2b
```

In this example, if you only want to mirror these images, you would then remove all other entries in the `mapping.txt` file and leave only the above two lines.

b. Still on your workstation with unrestricted network access, use your modified `mapping.txt` file to mirror the images to your registry using the `oc image mirror` command:

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

**WARNING**

If the `--filter-by-os` flag remains unset or set to any value other than `.*`, the command filters out different architectures, which changes the digest of the manifest list, also known as a *multi-arch image*. The incorrect digest causes deployments of those images and Operators on disconnected clusters to fail.

3. Create the `ImageContentSourcePolicy` object:

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

You can now create a `CatalogSource` object to reference your mirrored content.

Additional resources

- **Architecture and operating system support for Operators**
- If your catalog image is hosted on a private registry and requires authentication, see [Accessing images for Operators from private registries](#).

**4.8.2.3. Updating a Package Manifest Format catalog image**

After a cluster administrator has configured OperatorHub to use custom Operator catalog images, administrators can keep their OpenShift Container Platform cluster up to date with the latest Operators by capturing updates made to App Registry catalogs provided by Red Hat. This is done by building and pushing a new Operator catalog image, then replacing the existing `spec.image` parameter in the `CatalogSource` object with the new image digest.
For this example, the procedure assumes a custom redhat-operators catalog image is already configured for use with OperatorHub.

NOTE

Only the Linux version of the oc client can be used for this procedure, because the Windows and macOS versions do not provide the oc adm catalog build command.

Prerequisites

- Workstation with unrestricted network access
- oc version 4.3.5+ Linux client
- podman version 1.9.3+
- Access to mirror registry that supports Docker v2-2

IMPORTANT

The internal registry of the OpenShift Container Platform cluster cannot be used as the target registry because it does not support pushing without a tag, which is required during the mirroring process.

- OperatorHub configured to use custom catalog images
- If you are working with private registries, set the REG_CREDS environment variable to the file path of your registry credentials for use in later steps. For example, for the podman CLI:

  $ REG_CREDS=${XDG_RUNTIME_DIR}/containers/auth.json
  $ 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')

  $ podman login <registry_host_name>

- If you are working with private namespaces that your quay.io account has access to, you must set a Quay authentication token. Set the AUTH_TOKEN environment variable for use with the -auth-token flag by making a request against the login API using your quay.io credentials:

  $ 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')

Procedure

1. On the workstation with unrestricted network access, authenticate with the target mirror registry:

   $ podman login <registry_host_name>

2. Authenticate with registry.redhat.io so that the base image can be pulled during the build:
$ podman login registry.redhat.io

3. Build a new catalog image based on the redhat-operators catalog from Quay.io, tagging and pushing it to your mirror registry:

```
$ oc adm catalog build \
   --appregistry-org redhat-operators \1
   --from=registry.redhat.io/openshift4/ose-operator-registry:v4.7 \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 Organization (namespace) to pull from an App Registry instance.
2 Set --from to the Operator Registry base image using the tag that matches the target OpenShift Container Platform cluster major and minor version.
3 Set --filter-by-os to the operating system and architecture to use for the base image, which must match the target OpenShift Container Platform cluster. Valid values are linux/amd64, linux/ppc64le, and linux/s390x.
4 Name your catalog image and include a tag, for example, v2 because it is the updated catalog.
5 Optional: If required, specify the location of your registry credentials file.
6 Optional: If you do not want to configure trust for the target registry, add the --insecure flag.
7 Optional: If other application registry catalogs are used that are not public, specify a Quay authentication token.

Example output

```
INFO[0013] loading Bundles
dir=/var/folders/st/9cskxqs53ll3wdn434vw4cd80000gn/T/300666084/manifests-829192605 ...
Pushed sha256:f73d42950021f924038f999dd5b0c7f1b533c054ba344654ff1edaf6bf827e3
to example_registry:5000/olm/redhat-operators:v2
```

4. Mirror the contents of your catalog to your target registry. The following oc adm catalog mirror command extracts the contents of your custom Operator catalog image to generate the manifests required for mirroring and mirrors the images to your registry:

```
$ 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 <registry_host_name>:<port> of your target registry.
2 <registry_host_name>:<port> of your target registry.
3 Optional: If required, specify the location of your registry credentials file.
4 Optional: If you do not want to configure trust for the target registry, add the --insecure flag.
5 Optional: If other application registry catalogs are used that are not public, specify a Quay authentication token.
Specify your new Operator catalog image.

Specify the fully qualified domain name (FQDN) for the target registry.

Optional: If required, specify the location of your registry credentials file.

Optional: If you do not want to configure trust for the target registry, add the `--insecure` flag.

Optional: Specify which platform and architecture of the catalog image to select when multiple variants are available. Images are passed as `<platform>/<arch>[/<variant>]`. This does not apply to images referenced by the catalog image. Valid values are `linux/amd64`, `linux/ppc64le`, and `linux/s390x`.

5. Apply the newly generated manifests:

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

**IMPORTANT**

It is possible that you do not need to apply the `imageContentSourcePolicy.yaml` manifest. Complete a `diff` of the files to determine if changes are necessary.

6. Update your `CatalogSource` object that references your catalog image.

   a. If you have your original `catalogsource.yaml` file for this `CatalogSource` object:

      i. Edit your `catalogsource.yaml` file to reference your new catalog image in the `spec.image` field:

```
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
  displayName: My Operator Catalog
  publisher: grpc
```

   ii. Use the updated file to replace the `CatalogSource` object:

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

   b. Alternatively, edit the catalog source using the following command and reference your new catalog image in the `spec.image` parameter:

```
$ oc edit catalogsource <catalog_source_name> -n openshift-marketplace
```
Updated Operators should now be available from the OperatorHub page on your OpenShift Container Platform cluster.

Additional resources

- Architecture and operating system support for Operators

4.8.2.4. Testing a Package Manifest Format catalog image

You can validate Operator catalog image content by running it as a container and querying its gRPC API. To further test the image, you can then resolve a subscription in Operator Lifecycle Manager (OLM) by referencing the image in a catalog source. For this example, the procedure uses a custom `redhat-operators` catalog image previously built and pushed to a supported registry.

Prerequisites

- A custom Package Manifest Format catalog image pushed to a supported registry
- `podman` version 1.9.3+
- `oc` version 4.3.5+
- Access to mirror registry that supports Docker v2-2

**IMPORTANT**

The internal registry of the OpenShift Container Platform cluster cannot be used as the target registry because it does not support pushing without a tag, which is required during the mirroring process.

- `grpcurl`

Procedure

1. Pull the Operator catalog image:

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

2. Run the image:

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

3. Query the running image for available packages using `grpcurl`:

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

   **Example output**

   ```json
   {
     "name": "3scale-operator"
   }
   ```
4. Get the latest Operator bundle in a channel:

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

**Example output**

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

5. Get the digest of the image:

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

**Example output**

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

6. Assuming an Operator group exists in namespace `my-ns` that supports your Operator and its dependencies, create a **CatalogSource** object using the image digest. For example:

```yaml
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. Create a subscription that resolves the latest available **servicemeshoperator** and its dependencies from your catalog image:

```yaml
apiVersion: operators.coreos.com/v1alpha1
kind: Subscription
metadata:
  name: servicemeshoperator
  namespace: my-ns
```
4.8.3. Accessing images for Operators from private registries

If certain images relevant to Operators managed by Operator Lifecycle Manager (OLM) are hosted in an authenticated container image registry, also known as a private registry, OLM and OperatorHub are unable to pull the images by default. To enable access, you can create a pull secret that contains the authentication credentials for the registry. By referencing one or more pull secrets in a catalog source, OLM can handle placing the secrets in the Operator and catalog namespace to allow installation.

Other images required by an Operator or its Operands might require access to private registries as well. OLM does not handle placing the secrets in target tenant namespaces for this scenario, but authentication credentials can be added to the global cluster pull secret or individual namespace service accounts to enable the required access.

The following types of images should be considered when determining whether Operators managed by OLM have appropriate pull access:

**Index or catalog images**

A CatalogSource object can reference an index image or a catalog image, which are catalog sources packaged as container images hosted in images registries. Index images use the Bundle Format and reference bundle images, while catalog images use the Package Manifest Format. If an index or catalog image is hosted in a private registry, a secret can be used to enable pull access.

**Bundle images**

Operator bundle images are metadata and manifests packaged as container images that represent a unique version of an Operator. If any bundle images referenced in a catalog source are hosted in one or more private registries, a secret can be used to enable pull access.

**Operator and Operand images**

If an Operator installed from a catalog source uses a private image, either for the Operator image itself or one of the Operand images it watches, the Operator will fail to install because the deployment will not have access to the required registry authentication. Referencing secrets in a catalog source does not enable OLM to place the secrets in target tenant namespaces in which Operands are installed.

Instead, the authentication details can be added to the global cluster pull secret in the openshift-config namespace, which provides access to all namespaces on the cluster. Alternatively, if providing access to the entire cluster is not permissible, the pull secret can be added to the default service accounts of the target tenant namespaces.

**Prerequisites**

- At least one of the following hosted in a private registry:
  - An index image or catalog image.
  - An Operator bundle image.
  - An Operator or Operand image.

**Procedure**
1. Create a secret for each required private registry.

   a. Log in to the private registry to create or update your registry credentials file:

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

   **NOTE**

   The file path of your registry credentials can be different depending on the container tool used to log in to the registry. For the `podman` CLI, the default location is `${XDG_RUNTIME_DIR}/containers/auth.json`. For the `docker` CLI, the default location is `/root/.docker/config.json`.

   b. It is recommended to include credentials for only one registry per secret, and manage credentials for multiple registries in separate secrets. Multiple secrets can be included in a CatalogSource object in later steps, and OpenShift Container Platform will merge the secrets into a single virtual credentials file for use during an image pull. A registry credentials file can, by default, store details for more than one registry. Verify the current contents of your file. For example:

   **File storing credentials for two registries**

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

   Because this file is used to create secrets in later steps, ensure that you are storing details for only one registry per file. This can be accomplished by using either of the following methods:

   - Use the `podman logout <registry>` command to remove credentials for additional registries until only the one registry you want remains.

   - Edit your registry credentials file and separate the registry details to be stored in multiple files. For example:

     **File storing credentials for one registry**

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

     **File storing credentials for another registry**

     ```json
     {
       "auths": {
         "quay.io": {
           "auth": "Xd2lhdsbnRib21iMQ=="
         }
       }
     }
     ```
c. Create a secret in the **openshift-marketplace** namespace that contains the authentication credentials for a private registry:

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

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

Repeat this step to create additional secrets for any other required private registries, updating the --from-file flag to specify another registry credentials file path.

2. Create or update an existing **CatalogSource** object to reference one or more secrets:

```yaml
apiVersion: operators.coreos.com/v1alpha1
kind: CatalogSource
metadata:
  name: my-operator-catalog
  namespace: openshift-marketplace
spec:
  sourceType: grpc
  secrets: 1
  - "<secret_name_1>"
  - "<secret_name_2>"
  image: <registry>:<port>/<namespace>/<image>:<tag>
  displayName: My Operator Catalog
  publisher: <publisher_name>
  updateStrategy:
    registryPoll:
      interval: 30m
```

1 Add a **spec.secrets** section and specify any required secrets.

3. If any Operator or Operand images that are referenced by a subscribed Operator require access to a private registry, you can either provide access to all namespaces in the cluster, or individual target tenant namespaces.

- To provide access to all namespaces in the cluster, add authentication details to the global cluster pull secret in the **openshift-config** namespace.
Cluster resources must adjust to the new global pull secret, which can temporarily limit the usability of the cluster.

a. Extract the `.dockerconfigjson` file from the global pull secret:

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

b. Update the `.dockerconfigjson` file with your authentication credentials for the required private registry or registries and save it as a new file:

   ```bash
   $ cat .dockerconfigjson | 
   jq --compact-output .auths["<registry>:<port>/<namespace>"] |= . + {"auth": "<token>", "<token>"} | > new_dockerconfigjson
   
   Replace `<registry>:<port>/<namespace>` with the private registry details and `<token>` with your authentication credentials.
   ```

   c. Update the global pull secret with the new file:

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

   To update an individual namespace, add a pull secret to the service account for the Operator that requires access in the target tenant namespace.

   a. Recreate the secret that you created for the `openshift-marketplace` in the tenant namespace:

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

   b. Verify the name of the service account for the Operator by searching the tenant namespace:

   ```bash
   $ oc get sa -n <tenant_namespace> 
   
   1 If the Operator was installed in an individual namespace, search that namespace. If the Operator was installed for all namespaces, search the `openshift-operators` namespace.

   Example output
1 Service account for an installed etcd Operator.

c. Link the secret to the service account for the Operator:

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

Additional resources

- See [What is a secret?](#) for more information on the types of secrets, including those used for registry credentials.
- See [Updating the global cluster pull secret](#) for more details on the impact of changing this secret.
- See [Allowing pods to reference images from other secured registries](#) for more details on linking pull secrets to service accounts per namespace.

### 4.8.4. Disabling the default OperatorHub sources

Operator catalogs that source content provided by Red Hat and community projects are configured for OperatorHub by default during an OpenShift Container Platform installation. As a cluster administrator, you can disable the set of default catalogs.

**Procedure**

- Disable the sources for the default catalogs by adding `disableAllDefaultSources: true` to the `OperatorHub` object:

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

**TIP**

Alternatively, you can use the web console to manage catalog sources. From the Administration → Cluster Settings → Global Configuration → OperatorHub page, click the Sources tab, where you can create, delete, disable, and enable individual sources.

### 4.8.5. Removing custom catalogs

As a cluster administrator, you can remove custom Operator catalogs that have been previously added to your cluster by deleting the related catalog source.

**Procedure**

<table>
<thead>
<tr>
<th>NAME</th>
<th>SECRETS</th>
<th>AGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>builder</td>
<td>2</td>
<td>6m1s</td>
</tr>
<tr>
<td>default</td>
<td>2</td>
<td>6m1s</td>
</tr>
<tr>
<td>deployer</td>
<td>2</td>
<td>6m1s</td>
</tr>
<tr>
<td>etcd-operator</td>
<td>2</td>
<td>5m18s</td>
</tr>
</tbody>
</table>
1. In the Administrator perspective of the web console, navigate to Administration → Cluster Settings.

2. Click the Global Configuration tab, and then click OperatorHub.

3. Click the Sources tab.

4. Select the Options menu for the catalog that you want to remove, and then click Delete CatalogSource.

4.9. USING OPERATOR LIFECYCLE MANAGER ON RESTRICTED NETWORKS

For OpenShift Container Platform clusters that are installed on restricted networks, also known as disconnected clusters, Operator Lifecycle Manager (OLM) by default cannot access the Red Hat-provided OperatorHub sources hosted on remote registries because those remote sources require full Internet connectivity.

However, as a cluster administrator you can still enable your cluster to use OLM in a restricted network if you have a workstation that has full Internet access. The workstation, which requires full Internet access to pull the remote OperatorHub content, is used to prepare local mirrors of the remote sources, and push the content to a mirror registry.

The mirror registry can be located on a bastion host, which requires connectivity to both your workstation and the disconnected cluster, or a completely disconnected, or airgapped, host, which requires removable media to physically move the mirrored content to the disconnected environment.

This guide describes the following process that is required to enable OLM in restricted networks:

- Disable the default remote OperatorHub sources for OLM.
- Use a workstation with full Internet access to create and push local mirrors of the OperatorHub content to a mirror registry.
- Configure OLM to install and manage Operators from local sources on the mirror registry instead of the default remote sources.

After enabling OLM in a restricted network, you can continue to use your unrestricted workstation to keep your local OperatorHub sources updated as newer versions of Operators are released.
IMPORTANT

While OLM can manage Operators from local sources, the ability for a given Operator to run successfully in a restricted network still depends on the Operator itself. The Operator must:

- List any related images, or other container images that the Operator might require to perform their functions, in the `relatedImages` parameter of its `ClusterServiceVersion` (CSV) object.
- Reference all specified images by a digest (SHA) and not by a tag.

See the following Red Hat Knowledgebase Article for a list of Red Hat Operators that support running in disconnected mode:

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

Additional resources

- Red Hat-provided Operator catalogs
- Enabling your Operator for restricted network environments

4.9.1. Prerequisites

- Log in to your OpenShift Container Platform cluster as a user with `cluster-admin` privileges.
- If you want to prune the default catalog and selectively mirror only a subset of Operators, install the `opm` CLI.

NOTE

If you are using OLM in a restricted network on IBM Z, you must have at least 12 GB allocated to the directory where you place your registry.

4.9.2. Disabling the default OperatorHub sources

Operator catalogs that source content provided by Red Hat and community projects are configured for OperatorHub by default during an OpenShift Container Platform installation. In a restricted network environment, you must disable the default catalogs as a cluster administrator. You can then configure OperatorHub to use local catalog sources.

Procedure

- Disable the sources for the default catalogs by adding `disableAllDefaultSources: true` to the `OperatorHub` object:

```bash
$ oc patch OperatorHub cluster --type json -p '[["op": "add", "path": "/spec/disableAllDefaultSources", "value": true]]'
```
TIP
Alternatively, you can use the web console to manage catalog sources. From the Administration → Cluster Settings → Global Configuration → OperatorHub page, click the Sources tab, where you can create, delete, disable, and enable individual sources.

4.9.3. Pruning an index image

An index image, based on the Operator Bundle Format, is a containerized snapshot of an Operator catalog. You can prune an index of all but a specified list of packages, which creates a copy of the source index containing only the Operators that you want.

When configuring Operator Lifecycle Manager (OLM) to use mirrored content on restricted network OpenShift Container Platform clusters, use this pruning method if you want to only mirror a subset of Operators from the default catalogs.

For the steps in this procedure, the target registry is an existing mirror registry that is accessible by your workstation with unrestricted network access. This example also shows pruning the index image for the default redhat-operators catalog, but the process is the same for any index image.

Prerequisites
- Workstation with unrestricted network access
- podman version 1.9.3+
- grpcurl (third-party command-line tool)
- opm version 1.18.0+
- Access to a registry that supports Docker v2-2

IMPORTANT
The internal registry of the OpenShift Container Platform cluster cannot be used as the target registry because it does not support pushing without a tag, which is required during the mirroring process.

Procedure
1. Authenticate with registry.redhat.io:
   ```sh
   $ podman login registry.redhat.io
   ```
2. Authenticate with your target registry:
   ```sh
   $ podman login <target_registry>
   ```
3. Determine the list of packages you want to include in your pruned index.
   a. Run the source index image that you want to prune in a container. For example:
      ```sh
      ```
Example output

```
Trying to pull registry.redhat.io/redhat/redhat-operator-index:v4.7...
Getting image source signatures
Copying blob ae8a0c23f5b1 done
...
INFO[0000] serving registry                               database=/database/index.db port=50051
```

b. In a separate terminal session, use the `grpcurl` command to get a list of the packages provided by the index:

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

c. Inspect the `packages.out` file and identify which package names from this list you want to keep in your pruned index. For example:

**Example snippets of packages list**

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

d. In the terminal session where you executed the `podman run` command, press `Ctrl` and `C` to stop the container process.

4. Run the following command to prune the source index of all but the specified packages:

```
$ opm index prune \
  -f registry.redhat.io/redhat/redhat-operator-index:v4.7 \ 1
  -p advanced-cluster-management,jaeger-product,quay-operator \ 2
  [-i registry.redhat.io/openshift4/ose-operator-registry:v4.7] \ 3
  -t <target_registry>:<port>/<namespace>/redhat-operator-index:v4.7 4
```

1. Index to prune.
2. Comma-separated list of packages to keep.
3. Required only for IBM Power Systems and IBM Z images: Operator Registry base image with the tag that matches the target OpenShift Container Platform cluster major and minor version.
4. Custom tag for new index image being built.
5. Run the following command to push the new index image to your target registry:

```
$ podman push <target_registry>:<port>/<namespace>/redhat-operator-index:v4.7
```

where `<namespace>` is any existing namespace on the registry. For example, you might create an `olm-mirror` namespace to push all mirrored content to.

### 4.9.4. Mirroring an Operator catalog

You can mirror the Operator content of a Red Hat-provided catalog, or a custom catalog, into a container image registry using the `oc adm catalog mirror` command. The target registry must support Docker v2-2. For a cluster on a restricted network, this registry can be one that the cluster has network access to, such as a mirror registry created during a restricted network cluster installation.

**IMPORTANT**

The internal registry of the OpenShift Container Platform cluster cannot be used as the target registry because it does not support pushing without a tag, which is required during the mirroring process.

The `oc adm catalog mirror` command also automatically mirrors the index image that is specified during the mirroring process, whether it be a Red Hat-provided index image or your own custom-built index image, to the target registry. You can then use the mirrored index image to create a catalog source that allows Operator Lifecycle Manager (OLM) to load the mirrored catalog onto your OpenShift Container Platform cluster.

**Prerequisites**

- Workstation with unrestricted network access.
- `podman` version 1.9.3 or later.
- Access to mirror registry that supports Docker v2-2.
- Decide which namespace on your mirror registry you will use to store the mirrored Operator content. For example, you might create an `olm-mirror` namespace.
- If your mirror registry does not have Internet access, connect removable media to your workstation with unrestricted network access.
- If you are working with private registries, including `registry.redhat.io`, set the `REG_CREDS` environment variable to the file path of your registry credentials for use in later steps. For example, for the `podman` CLI:

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

**Procedure**

1. If you want to mirror a Red Hat-provided catalog, run the following command on your workstation with unrestricted network access to authenticate with `registry.redhat.io`:

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

2. The `oc adm catalog mirror` command extracts the contents of an index image to generate the...
manifests required for mirroring. The default behavior of the command generates manifests, then automatically mirrors all of the image content from the index image, as well as the index image itself, to your mirror registry. Alternatively, if your mirror registry is on a completely disconnected, or airgapped, host, you can first mirror the content to removable media, move the media to the disconnected environment, then mirror the content from the media to the registry.

- **Option A:** If your mirror registry is on the same network as your workstation with unrestricted network access, take the following actions on your workstation:

  a. If your mirror registry requires authentication, run the following command to log in to the registry:

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

  b. Run the following command to mirror the content:

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

  1. Specify the index image for the catalog you want to mirror. For example, this might be a pruned index image that you created previously, or one of the source index images for the default catalogs, such as `registry.redhat.io/redhat/redhat-operator-index:v4.7`.

  2. Specify the fully qualified domain name (FQDN) for the target registry and namespace to mirror the Operator content to, where `<namespace>` is any existing namespace on the registry. For example, you might create an `olm-mirror` namespace to push all mirrored content to.

  3. Optional: If required, specify the location of your registry credentials file. `REG_CREDS` is required for `registry.redhat.io`.

  4. Optional: If you do not want to configure trust for the target registry, add the `--insecure` flag.

  5. Optional: Specify which platform and architecture of the index image to select when multiple variants are available. Images are passed as `'<platform>/<arch> [/<variant>]'`. This does not apply to images referenced by the index. Valid values are `linux/amd64`, `linux/ppc64le`, and `linux/s390x`.

  6. Optional: Generate only the manifests required for mirroring, and do not actually mirror the image content to a registry. This option can be useful for reviewing what will be mirrored, and it allows you to make any changes to the mapping list if you require only a subset of packages. You can then use the `mapping.txt` file with the `oc image mirror` command to mirror the modified list of images in a later step. This flag is intended for only advanced selective mirroring of content from the catalog; the `opm index prune` command, if you used it previously to prune the index image, is suitable for most catalog management use cases.
Example output

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

...  
wrote mirroring manifests to manifests-redhat-operator-index-1614211642

1 Directory for the temporary index.db database generated by the command.

2 Be sure to record the manifests directory name that is generated. This directory name is used in a later step.

NOTE

Red Hat Quay does not support nested repositories. As a result, running the `oc adm catalog mirror` command will fail with a 401 unauthorized error. As a workaround, you can use the `--max-components=2` option when running the `oc adm catalog mirror` command to disable the creation of nested repositories. For more information on this workaround, see the Unauthorized error thrown while using catalog mirror command with Quay registry Knowledgebase Solution article.

• Option B: If your mirror registry is on a disconnected host, take the following actions.
  
a. Run the following command on your workstation with unrestricted network access to mirror the content to local files:

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

1 Specify the index image for the catalog you want to mirror. For example, this might be a pruned index image that you created previously, or one of the source index images for the default catalogs, such as `registry.redhat.io/redhat/redhat-operator-index:v4.7`.

2 Mirrors content to local files in your current directory.

Example output

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

To upload local images to a registry, run:

```
oc adm catalog mirror file:///local/index/myrepo/my-index:v1
REGISTRY/REPOSITORY
```
Be sure to record the manifests directory name that is generated. This directory name is used in a later step.

Record the expanded file:// path that based on your provided index image. This path is used in a later step.

b. Copy the v2/ directory that is generated in your current directory to removable media.

c. Physically remove the media and attach it to a host in the disconnected environment that has access to the mirror registry.

d. If your mirror registry requires authentication, run the following command on your host in the disconnected environment to log in to the registry:

   $ podman login <mirror_registry>

e. Run the following command from the parent directory containing the v2/ directory to upload the images from local files to the mirror registry:

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

   1 Specify the file:// path from the previous command output.
   2 Specify the fully qualified domain name (FQDN) for the target registry and namespace to mirror the Operator content to, where <namespace> is any existing namespace on the registry. For example, you might create an olm-mirror namespace to push all mirrored content to.

   NOTE

   Red Hat Quay does not support nested repositories. As a result, running the oc adm catalog mirror command will fail with a 401 unauthorized error. As a workaround, you can use the --max-components=2 option when running the oc adm catalog mirror command to disable the creation of nested repositories. For more information on this workaround, see the Unauthorized error thrown while using catalog mirror command with Quay registry Knowledgebase Solution article.

f. Run the oc adm catalog mirror command again. Use the newly mirrored index image as the source and the same mirror registry namespace used in the previous step as the target:

   $ oc adm catalog mirror \
   <mirror_registry>:<port>/<index_image> \
   <mirror_registry>:<port>/<namespace> \
   --manifests-only \1
   [-a ${REG_CREDS}] \
   [-insecure]
The `--manifests-only` flag is required for this step so that the command does not copy all of the mirrored content again.

**IMPORTANT**

This step is required because the image mappings in the `imageContentSourcePolicy.yaml` file generated during the previous step must be updated from local paths to valid mirror locations. Failure to do so will cause errors when you create the `ImageContentSourcePolicy` object in a later step.

3. After mirroring the content to your registry, inspect the manifests directory that is generated in your current directory.

**NOTE**

The manifests directory name is used in a later step.

If you mirrored content to a registry on the same network in the previous step, the directory name takes the following form:

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

If you mirrored content to a registry on a disconnected host in the previous step, the directory name takes the following form:

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

The manifests directory contains the following files, some of which might require further modification:

- **The `catalogSource.yaml` file** is a basic definition for a `CatalogSource` object that is pre-populated with your index image tag and other relevant metadata. This file can be used as is or modified to add the catalog source to your cluster.

  **IMPORTANT**

  If you mirrored the content to local files, you must modify your `catalogSource.yaml` file to remove any backslash (`/`) characters from the `metadata.name` field. Otherwise, when you attempt to create the object, it fails with an "invalid resource name" error.

- **The `imageContentSourcePolicy.yaml` file** defines an `ImageContentSourcePolicy` object that can configure nodes to translate between the image references stored in Operator manifests and the mirrored registry.

  **NOTE**

  If your cluster uses an `ImageContentSourcePolicy` object to configure repository mirroring, you can use only global pull secrets for mirrored registries. You cannot add a pull secret to a project.
• The *mapping.txt* file contains all of the source images and where to map them in the target registry. This file is compatible with the *oc image mirror* command and can be used to further customize the mirroring configuration.

**IMPORTANT**

If you used the *--manifests-only* flag during the mirroring process and want to further trim the subset of packages to be mirrored, see the steps in the “Mirroring a Package Manifest Format catalog image” procedure about modifying your *mapping.txt* file and using the file with the *oc image mirror* command. After following those further actions, you can continue this procedure.

4. On a host with access to the disconnected cluster, create the *ImageContentSourcePolicy* (ICSP) object by running the following command to specify the *imageContentSourcePolicy.yaml* file in your manifests directory:

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

where `<path/to/manifests/dir>` is the path to the manifests directory for your mirrored content.

**NOTE**

Applying the ICSP causes all worker nodes in the cluster to restart. You must wait for this reboot process to finish cycling through each of your worker nodes before proceeding.

You can now create a *CatalogSource* object to reference your mirrored index image and Operator content.

**Additional resources**

- Mirroring images for a disconnected installation
- Architecture and operating system support for Operators
- Mirroring a Package Manifest Format catalog image

**4.9.5. Creating a catalog from an index image**

You can create an Operator catalog from an index image and apply it to an OpenShift Container Platform cluster for use with Operator Lifecycle Manager (OLM).

**Prerequisites**

- An index image built and pushed to a registry.

**Procedure**

1. Create a *CatalogSource* object that references your index image. If you used the *oc adm catalog mirror* command to mirror your catalog to a target registry, you can use the generated *catalogSource.yaml* file as a starting point.

   a. Modify the following to your specifications and save it as a *catalogSource.yaml* file:
If you mirrored content to local files before uploading to a registry, remove any backslash (\) characters from the `metadata.name` field to avoid an "invalid resource name" error when you create the object.

If you want the catalog source to be available globally to users in all namespaces, specify the `openshift-marketplace` namespace. Otherwise, you can specify a different namespace for the catalog to be scoped and available only for that namespace.

Specify your index image.

Specify your name or an organization name publishing the catalog.

Catalog sources can automatically check for new versions to keep up to date.

b. Use the file to create the `CatalogSource` object:

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

2. Verify the following resources are created successfully.

a. Check the pods:

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

Example output

<table>
<thead>
<tr>
<th>NAME</th>
<th>READY</th>
<th>STATUS</th>
<th>RESTARTS</th>
<th>AGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>my-operator-catalog-6njx6</td>
<td>1/1</td>
<td>Running</td>
<td>0</td>
<td>28s</td>
</tr>
<tr>
<td>marketplace-operator-d9f549946-96sgr</td>
<td>1/1</td>
<td>Running</td>
<td>0</td>
<td>26h</td>
</tr>
</tbody>
</table>

b. Check the catalog source:

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

Example output

<table>
<thead>
<tr>
<th>NAME</th>
<th>DISPLAY</th>
<th>TYPE</th>
<th>PUBLISHER</th>
<th>AGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>my-operator-catalog</td>
<td>My Operator Catalog</td>
<td>grpc</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
c. Check the package manifest:

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

**Example output**

<table>
<thead>
<tr>
<th>NAME</th>
<th>CATALOG</th>
<th>AGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>jaeger-product</td>
<td>My Operator Catalog</td>
<td>93s</td>
</tr>
</tbody>
</table>

You can now install the Operators from the OperatorHub page on your OpenShift Container Platform web console.

**Additional resources**

- If your index image is hosted on a private registry and requires authentication, see [Accessing images for Operators from private registries](#).

### 4.9.6. Updating an index image

After configuring OperatorHub to use a catalog source that references a custom index image, cluster administrators can keep the available Operators on their cluster up to date by adding bundle images to the index image.

You can update an existing index image using the `opm index add` command. For restricted networks, the updated content must also be mirrored again to the cluster.

**Prerequisites**

- `opm` version 1.12.3+
- `podman` version 1.9.3+
- An index image built and pushed to a registry.
- An existing catalog source referencing the index image.

**Procedure**

1. Update the existing index by adding bundle images:

```bash
$ 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. The `--bundles` flag specifies a comma-separated list of additional bundle images to add to the index.
2. The `--from-index` flag specifies the previously pushed index.
3. The `--tag` flag specifies the image tag to apply to the updated index image.
The `--pull-tool` flag specifies the tool used to pull container images.

where:

`<registry>`
- Specifies the hostname of the registry, such as `quay.io` or `mirror.example.com`.

`<namespace>`
- Specifies the namespace of the registry, such as `ocs-dev` or `abc`.

`<new_bundle_image>`
- Specifies the new bundle image to add to the registry, such as `ocs-operator`.

`<digest>`
- Specifies the SHA image ID, or digest, of the bundle image, such as `c7f11097a628f092d8bad148406aa0e0951094a03445fd4bc0775431ef683a41`.

`<existing_index_image>`
- Specifies the previously pushed image, such as `abc-redhat-operator-index`.

`<existing_tag>`
- Specifies a previously pushed image tag, such as `4.7`.

`<updated_tag>`
- Specifies the image tag to apply to the updated index image, such as `4.7.1`.

**Example command**

```bash
$ opm index add
   --bundles quay.io/ocs-dev/ocs-operator@sha256:c7f11097a628f092d8bad148406aa0e0951094a03445fd4bc0775431ef683a41
   --from-index mirror.example.com/abc/abc-redhat-operator-index:4.7
   --tag mirror.example.com/abc/abc-redhat-operator-index:4.7.1
   --pull-tool podman
```

2. Push the updated index image:

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

3. Follow the steps in the *Mirroring an Operator catalog* procedure again to mirror the updated content. However, when you get to the step about creating the `ImageContentSourcePolicy` (ICSP) object, use the `oc replace` command instead of the `oc create` command. For example:

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

This change is required because the object already exists and must be updated.

**NOTE**

Normally, the `oc apply` command can be used to update existing objects that were previously created using `oc apply`. However, due to a known issue regarding the size of the `metadata.annotations` field in ICSP objects, the `oc replace` command must be used for this step currently.
4. After Operator Lifecycle Manager (OLM) automatically polls the index image referenced in the catalog source at its regular interval, verify that the new packages are successfully added:

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

Additional resources

- Mirroring an Operator catalog
5.1. ABOUT THE OPERATOR SDK

The Operator Framework is an open source toolkit to manage Kubernetes native applications, called Operators, in an effective, automated, and scalable way. Operators take advantage of Kubernetes extensibility to deliver the automation advantages of cloud services, like provisioning, scaling, and backup and restore, while being able to run anywhere that Kubernetes can run.

Operators make it easy to manage complex, stateful applications on top of Kubernetes. However, writing an Operator today can be difficult because of challenges such as using low-level APIs, writing boilerplate, and a lack of modularity, which leads to duplication.

The Operator SDK, a component of the Operator Framework, provides a command-line interface (CLI) tool that Operator developers can use to build, test, and deploy an Operator.

Why use the Operator SDK?

The Operator SDK simplifies this process of building Kubernetes-native applications, which can require deep, application-specific operational knowledge. The Operator SDK not only lowers that barrier, but it also helps reduce the amount of boilerplate code required for many common management capabilities, such as metering or monitoring.

The Operator SDK is a framework that uses the controller-runtime library to make writing Operators easier by providing the following features:

- High-level APIs and abstractions to write the operational logic more intuitively
- Tools for scaffolding and code generation to quickly bootstrap a new project
- Integration with Operator Lifecycle Manager (OLM) to streamline packaging, installing, and running Operators on a cluster
- Extensions to cover common Operator use cases
- Metrics set up automatically in any generated Go-based Operator for use on clusters where the Prometheus Operator is deployed

Operator authors with cluster administrator access to a Kubernetes-based cluster (such as OpenShift Container Platform) can use the Operator SDK CLI to develop their own Operators based on Go, Ansible, or Helm. Kubebuilder is embedded into the Operator SDK as the scaffolding solution for Go-based Operators, which means existing Kubebuilder projects can be used as is with the Operator SDK and continue to work.

NOTE

OpenShift Container Platform 4.7 supports Operator SDK v1.3.0 or later.

5.1.1. What are Operators?

For an overview about basic Operator concepts and terminology, see Understanding Operators.

5.1.2. Development workflow

The Operator SDK provides the following workflow to develop a new Operator:
1. Create an Operator project by using the Operator SDK command-line interface (CLI).

2. Define new resource APIs by adding custom resource definitions (CRDs).

3. Specify resources to watch by using the Operator SDK API.

4. Define the Operator reconciling logic in a designated handler and use the Operator SDK API to interact with resources.

5. Use the Operator SDK CLI to build and generate the Operator deployment manifests.

**Figure 5.1. Operator SDK workflow**

**Operator SDK  Build, test, iterate**

At a high level, an Operator that uses the Operator SDK processes events for watched resources in an Operator author-defined handler and takes actions to reconcile the state of the application.

5.1.3. Additional resources

- Certified Operator Build Guide

5.2. INSTALLING THE OPERATOR SDK CLI

The Operator SDK provides a command-line interface (CLI) tool that Operator developers can use to build, test, and deploy an Operator. You can install the Operator SDK CLI on your workstation so that you are prepared to start authoring your own Operators.

**NOTE**

OpenShift Container Platform 4.7 supports Operator SDK v1.3.0.

5.2.1. Installing the Operator SDK CLI

You can install the OpenShift SDK CLI tool on Linux.

**Prerequisites**

- **Go** v1.13+

- **docker** v1.703+, **podman** v1.9.3+, or **buildah** v1.7+
Procedure

1. Navigate to the OpenShift mirror site.

2. From the 4.7.23 directory, download the latest version of the tarball for Linux.

3. Unpack the archive:

   $ tar xvf operator-sdk-v1.3.0-ocp-linux-x86_64.tar.gz

4. Make the file executable:

   $ chmod +x operator-sdk

5. Move the extracted operator-sdk binary to a directory that is on your PATH.

   TIP

   To check your PATH:

   $ echo $PATH

   $ sudo mv ./operator-sdk /usr/local/bin/operator-sdk

Verification

- After you install the Operator SDK CLI, verify that it is available:

  $ operator-sdk version

Example output

operator-sdk version: "v1.3.0-ocp", ...

5.3. GO-BASED OPERATORS

5.3.1. Getting started with Operator SDK for Go-based Operators

To demonstrate the basics of setting up and running a Go-based Operator using tools and libraries provided by the Operator SDK, Operator developers can build an example Go-based Operator for Memcached, a distributed key-value store, and deploy it to a cluster.

5.3.1.1. Prerequisites

- Operator SDK CLI installed

- OpenShift CLI (oc) v4.7+ installed

- Logged into an OpenShift Container Platform 4.7 cluster with oc with an account that has cluster-admin permissions
To allow the cluster pull the image, the repository where you push your image must be set as public, or you must configure an image pull secret.

### 5.3.1.2. Creating and deploying Go-based Operators

You can build and deploy a simple Go-based Operator for Memcached by using the Operator SDK.

**Procedure**

1. **Create a project.**
   a. Create your project directory:
   ```
   $ mkdir memcached-operator
   ```
   b. Change into the project directory:
   ```
   $ cd memcached-operator
   ```
   c. Run the `operator-sdk init` command to initialize the project:
   ```
   $ operator-sdk init
   --domain=example.com
   --repo=github.com/example-inc/memcached-operator
   ```
   The command uses the Go plug-in by default.
   d. To enable your Go-based Operator to run on OpenShift Container Platform, edit the `config/manager/manager.yaml` file and replace the following line:
   ```yaml
   runAsUser: 65532
   ``
   with:
   ```yaml
   runAsNonRoot: true
   ```
   **NOTE**
   This step is a temporary workaround required for Go-based Operators only. For more information, see [BZ#1914406](#).

2. **Create an API.**
   Create a simple Memcached API:
   ```
   $ operator-sdk create api
   --resource=true
   --controller=true
   --group cache
   --version v1
   --kind Memcached
   ```

3. **Build and push the Operator image.**
Use the default Makefile targets to build and push your Operator. Set IMG with a pull spec for your image that uses a registry you can push to:

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

4. Run the Operator.
   a. Install the CRD:
      ```bash
      $ make install
      ```
   b. Deploy the project to the cluster. Set IMG to the image that you pushed:
      ```bash
      $ make deploy IMG=<registry>/<user>/<image_name>:<tag>
      ```

5. Create a sample custom resource (CR).
   a. Create a sample CR:
      ```bash
      $ oc apply -f config/samples/cache_v1_memcached.yaml
      -n memcached-operator-system
      ```
   b. Watch for the CR to reconcile the Operator:
      ```bash
      $ oc logs deployment.apps/memcached-operator-controller-manager
      -c manager
      -n memcached-operator-system
      ```

6. Clean up.
   Run the following command to clean up the resources that have been created as part of this procedure:

   ```bash
   $ make undeploy
   ```

5.3.1.3. Next steps

- See Operator SDK tutorial for Go-based Operators for a more in-depth walkthrough on building a Go-based Operator.

5.3.2. Operator SDK tutorial for Go-based Operators

Operator developers can take advantage of Go programming language support in the Operator SDK to build an example Go-based Operator for Memcached, a distributed key-value store, and manage its lifecycle.

This process is accomplished using two centerpieces of the Operator Framework:

**Operator SDK**

The operator-sdk CLI tool and controller-runtime library API

**Operator Lifecycle Manager (OLM)**

Installation, upgrade, and role-based access control (RBAC) of Operators on a cluster.
5.3.2.1. Prerequisites

- Operator SDK CLI installed
- OpenShift CLI (oc) v4.7+ installed
- Logged into an OpenShift Container Platform 4.7 cluster with oc with an account that has cluster-admin permissions
- To allow the cluster pull the image, the repository where you push your image must be set as public, or you must configure an image pull secret.

5.3.2.2. Creating a project

Use the Operator SDK CLI to create a project called memcached-operator.

Procedure

1. Create a directory for the project:

   $ mkdir -p $HOME/projects/memcached-operator

2. Change to the directory:

   $ cd $HOME/projects/memcached-operator

3. Activate support for Go modules:

   $ export GO111MODULE=on

4. Run the operator-sdk init command to initialize the project:

   $ operator-sdk init \
     --domain=example.com \
     --repo=github.com/example-inc/memcached-operator

   The operator-sdk init command uses the Go plug-in by default.

   The operator-sdk init command generates a go.mod file to be used with Go modules. The --repo flag is required when creating a project outside of $GOPATH/src/, because generated files require a valid module path.

5. To enable your Go-based Operator to run on OpenShift Container Platform, edit the config/manager/manager.yaml file and replace the following line:

   runAsUser: 65532
CHAPTER 5. DEVELOPING OPERATORS

with:
runAsNonRoot: true

NOTE
This step is a temporary workaround required for Go-based Operators only. For
more information, see BZ#1914406.
5.3.2.2.1. PROJECT file
Among the files generated by the operator-sdk init command is a Kubebuilder PROJECT file.
Subsequent operator-sdk commands, as well as help output, that are run from the project root read
this file and are aware that the project type is Go. For example:
domain: example.com
layout: go.kubebuilder.io/v3
projectName: memcached-operator
repo: github.com/example-inc/memcached-operator
version: 3-alpha
plugins:
manifests.sdk.operatorframework.io/v2: {}
scorecard.sdk.operatorframework.io/v2: {}
5.3.2.2.2. About the Manager
The main program for the Operator is the main.go file, which initializes and runs the Manager. The
Manager automatically registers the Scheme for all custom resource (CR) API definitions and sets up
and runs controllers and webhooks.
The Manager can restrict the namespace that all controllers watch for resources:
mgr, err := ctrl.NewManager(cfg, manager.Options{Namespace: namespace})
By default, the Manager watches the namespace where the Operator runs. To watch all namespaces,
you can leave the namespace option empty:
mgr, err := ctrl.NewManager(cfg, manager.Options{Namespace: ""})
You can also use the MultiNamespacedCacheBuilder function to watch a specific set of namespaces:
var namespaces []string 1
mgr, err := ctrl.NewManager(cfg, manager.Options{ 2
NewCache: cache.MultiNamespacedCacheBuilder(namespaces),
})
1

List of namespaces.

2

Creates a Cmd struct to provide shared dependencies and start components.

5.3.2.2.3. About multi-group APIs

139


Before you create an API and controller, consider whether your Operator requires multiple API groups. This tutorial covers the default case of a single group API, but to change the layout of your project to support multi-group APIs, you can run the following command:

```
$ operator-sdk edit --multigroup=true
```

This command updates the PROJECT file, which should look like the following example:

```
domain: example.com
layout: go.kubebuilder.io/v3
multigroup: true
...
```

For multi-group projects, the API Go type files are created in the `apis/<group>/<version>/` directory, and the controllers are created in the `controllers/<group>/` directory. The Dockerfile is then updated accordingly.

**Additional resource**

- For more details on migrating to a multi-group project, see the [Kubebuilder documentation](https://kubebuilder.io/docs/

### 5.3.2.3. Creating an API and controller

Use the Operator SDK CLI to create a custom resource definition (CRD) API and controller.

**Procedure**

1. Run the following command to create an API with group `cache`, version, `v1`, and kind `Memcached`:

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

2. When prompted, enter `y` for creating both the resource and controller:

   ```
   Create Resource [y/n]
   y
   Create Controller [y/n]
   y
   ```

**Example output**

```
Writing scaffold for you to edit...
api/v1/memcached_types.go
controllers/memcached_controller.go
...
```

This process generates the `Memcached` resource API at `api/v1/memcached_types.go` and the controller at `controllers/memcached_controller.go`.

### 5.3.2.3.1. Defining the API
Define the API for the Memcached custom resource (CR).

Procedure

1. Modify the Go type definitions at `api/v1/memcached_types.go` to have the following spec and status:

   ```go
   // MemcachedSpec defines the desired state of Memcached
type MemcachedSpec struct {
    // +kubebuilder:validation:Minimum=0
    // Size is the size of the memcached deployment
    Size int32 `json:"size"
}

   // MemcachedStatus defines the observed state of Memcached
type MemcachedStatus struct {
    // Nodes are the names of the memcached pods
    Nodes []string `json:"nodes"
}
```

2. Update the generated code for the resource type:

   ```bash
   $ make generate
   ```

   **TIP**

   After you modify a `*_types.go` file, you must run the `make generate` command to update the generated code for that resource type.

   The above Makefile target invokes the `controller-gen` utility to update the `api/v1/zz_generated deepcopy.go` file. This ensures your API Go type definitions implement the `runtime.Object` interface that all Kind types must implement.

5.3.2.3.2. Generating CRD manifests

After the API is defined with spec and status fields and custom resource definition (CRD) validation markers, you can generate CRD manifests.

Procedure

- Run the following command to generate and update CRD manifests:

  ```bash
  $ make manifests
  ```

  This Makefile target invokes the `controller-gen` utility to generate the CRD manifests in the `config/crd/bases/cache.example.com_memcacheds.yaml` file.

5.3.2.3.2.1. About OpenAPI validation

OpenAPIv3 schemas are added to CRD manifests in the `spec.validation` block when the manifests are generated. This validation block allows Kubernetes to validate the properties in a Memcached custom resource (CR) when it is created or updated.
Markers, or annotations, are available to configure validations for your API. These markers always have a `+kubebuilder:validation` prefix.

**Additional resources**

- For more details on the usage of markers in API code, see the following Kubebuilder documentation:
  - CRD generation
  - Markers
  - List of OpenAPIv3 validation markers
- For more details about OpenAPIv3 validation schemas in CRDs, see the Kubernetes documentation.

### 5.3.2.4. Implementing the controller

After creating a new API and controller, you can implement the controller logic.

**Procedure**

- For this example, replace the generated controller file `controllers/memcached_controller.go` with following example implementation:

```
/*
Copyright 2020.
Licensed under the Apache License, Version 2.0 (the "License");
you may not use this file except in compliance with the License.
You may obtain a copy of the License at
http://www.apache.org/licenses/LICENSE-2.0

Unless required by applicable law or agreed to in writing, software
distributed under the License is distributed on an "AS IS" BASIS,
WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
See the License for the specific language governing permissions and
limitations under the License.
*/

package controllers

import (
  appsv1 "k8s.io/api/apps/v1"
  corev1 "k8s.io/api/core/v1"
  "k8s.io/apimachinery/pkg/api/errors"
  metav1 "k8s.io/apimachinery/pkg/apis/meta/v1"
  "k8s.io/apimachinery/pkg/types"
  "reflect"
  "context"
```
// MemcachedReconciler reconciles a Memcached object

type MemcachedReconciler struct {
    client.Client
    Log    logr.Logger
    Scheme *runtime.Scheme
}

// Reconcile is part of the main kubernetes reconciliation loop which aims to
// move the current state of the cluster closer to the desired state.
// TODO(user): Modify the Reconcile function to compare the state specified by
// the Memcached object against the actual cluster state, and then
// perform operations to make the cluster state reflect the state specified by
// the user.
// 
// For more details, check Reconcile and its Result here:
// - https://pkg.go.dev/sigs.k8s.io/controller-runtime@v0.7.0/pkg/reconcile

func (r *MemcachedReconciler) Reconcile(ctx context.Context, req ctrl.Request) (ctrl.Result, error) {
    log := r.Log.WithValues("memcached", req.NamespacedName)

    // Fetch the Memcached instance
    memcached := &cachev1alpha1.Memcached{
        err := r.Get(ctx, req.NamespacedName, memcached)
        if err != nil {
            if errors.IsNotFound(err) {
                // Request object not found, could have been deleted after reconcile request.
                // Owned objects are automatically garbage collected. For additional cleanup logic use
                // finalizers.
                // Return and don't requeue
                log.Info("Memcached resource not found. Ignoring since object must be deleted")
                return ctrl.Result{}, nil
            } else {
                // Error reading the object - requeue the request.
                log.Error(err, "Failed to get Memcached")
            }
        }

        // Update the Memcached status with observed values
        // and apply finalizers if applicable
        // ... (update logic)
    }

    return ctrl.Result{}, nil
}
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)
    err = r.Create(ctx, dep)
    if err != nil {
        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 {
        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 {
    return ctrl.Result{}, err
}
podNames := getPodNames(podList.Items)

// Update status.Nodes if needed
if !reflect.DeepEqual(podNames, memcached.Status.Nodes) {
    memcached.Status.Nodes = podNames
}
err := r.Status().Update(ctx, memcached)
if err != nil {
    log.Error(err, "Failed to update Memcached status")
    return ctrl.Result{}, err
}

return ctrl.Result{}, nil

// deploymentForMemcached returns a memcached Deployment object
func (r *MemcachedReconciler) deploymentForMemcached(m *cachev1alpha1.Memcached) *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
The example controller runs the following reconciliation logic for each Memcached custom resource (CR):

- Create a Memcached deployment if it does not exist.
- Ensure that the deployment size is the same as specified by the Memcached CR spec.
- Update the Memcached CR status with the names of the memcached pods.

The next subsections explain how the controller in the example implementation watches resources and how the reconcile loop is triggered. You can skip these subsections to go directly to Running the Operator.

### 5.3.2.4.1. Resources watched by the controller

The SetupWithManager() function in controllers/memcached_controller.go specifies how the controller is built to watch a CR and other resources that are owned and managed by that controller.

```go
import (...
  appsv1 "k8s.io/api/apps/v1"
  ...
)

func (r *MemcachedReconciler) SetupWithManager(mgr ctrl.Manager) error {
  return ctrl.NewControllerManagedBy(mgr).
    For(&cachev1alpha1.Memcached{}).
    Owns(&appsv1.Deployment{}).
    Complete(r)
}
```

NewControllerManagedBy() provides a controller builder that allows various controller configurations.

For(&cachev1alpha1.Memcached{}) specifies the Memcached type as the primary resource to watch. For each Add, Update, or Delete event for a Memcached type, the reconcile loop is sent a reconcile Request argument, which consists of a namespace and name key, for that Memcached object.

Owns(&appsv1.Deployment{}) specifies the Deployment type as the secondary resource to watch.
For each **Deployment** type Add, Update, or Delete event, the event handler maps each event to a reconcile request for the owner of the deployment. In this case, the owner is the **Memcached** object for which the deployment was created.

### 5.3.2.4.2. Controller configurations

You can initialize a controller by using many other useful configurations. For example:

- Set the maximum number of concurrent reconciles for the controller by using the `MaxConcurrentReconciles` option, which defaults to 1:

  ```go
define (r *MemcachedReconciler) SetupWithManager(mgr ctrl.Manager) error {
    return ctrl.NewControllerManagedBy(mgr).
      For(&cachev1.Memcached{}).
      Owns(&appsv1.Deployment{}).
      WithOptions(controller.Options{
        MaxConcurrentReconciles: 2,
      }).
      Complete(r)
  }
```

- Filter watch events using predicates.

- Choose the type of **EventHandler** to change how a watch event translates to reconcile requests for the reconcile loop. For Operator relationships that are more complex than primary and secondary resources, you can use the **EnqueueRequestsFromMapFunc** handler to transform a watch event into an arbitrary set of reconcile requests.

For more details on these and other configurations, see the upstream **Builder** and **Controller** GoDocs.

### 5.3.2.4.3. Reconcile loop

Every controller has a reconciler object with a **Reconcile()** method that implements the reconcile loop. The reconcile loop is passed the **Request** argument, which is a namespace and name key used to find the primary resource object, **Memcached**, from the cache:

```go
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)
  ... 
}
```

Based on the return values, result, and error, the request might be requeued and the reconcile loop might be triggered again:

```go
// Reconcile successful - don’t requeue  
return ctrl.Result{}, nil
```
You can set the `Result.RequeueAfter` to requeue the request after a grace period as well:

```go
import "time"

// Reconcile for any reason other than an error after 5 seconds
return ctrl.Result{RequeueAfter: time.Second*5}, nil
```

For more on reconcilers, clients, and interacting with resource events, see the Controller Runtime Client API documentation.

### 5.3.2.4.4. Permissions and RBAC manifests

The controller requires certain RBAC permissions to interact with the resources it manages. These are specified using RBAC markers, such as the following:

```go
// +kubebuilder:rbac:groups=cache.example.com,resources=memcacheds,verbs=get;list;watch;create;update;patch;delete
// +kubebuilder:rbac:groups=cache.example.com,resources=memcacheds/status,verbs=get;update;patch
// +kubebuilder:rbac:groups=cache.example.com,resources=memcacheds/finalizers,verbs=update
// +kubebuilder:rbac:groups=apps,resources=deployments,verbs=get;list;watch;create;update;patch;delete
// +kubebuilder:rbac:groups=core,resources=pods,verbs=get;list;

func (r *MemcachedReconciler) Reconcile(ctx context.Context, req ctrl.Request) (ctrl.Result, error) {
  ...
}
```

The `ClusterRole` object manifest at `config/rbac/role.yaml` is generated from the previous markers by using the `controller-gen` utility whenever the `make manifests` command is run.

### 5.3.2.5. Running the Operator

There are three ways you can use the Operator SDK CLI to build and run your Operator:

- Run locally outside the cluster as a Go program.
- Run as a deployment on the cluster.
- Bundle your Operator and use Operator Lifecycle Manager (OLM) to deploy on the cluster.
NOTE

Before running your Go-based Operator as either a deployment on OpenShift Container Platform or as a bundle that uses OLM, ensure that your project has been updated to use supported images.

5.3.2.5.1. Running locally outside the cluster

You can run your Operator project as a Go program outside of the cluster. This is useful for development purposes to speed up deployment and testing.

Procedure

- Run the following command to install the custom resource definitions (CRDs) in the cluster configured in your ~/.kube/config file and run the Operator locally:

  $ make install run

Example output

```
```

5.3.2.5.2. Preparing your Operator to use supported images

Before running your Go-based Operator on OpenShift Container Platform, update your project to use supported images.

Procedure

1. Update the project root-level Dockerfile to use supported images. Change the default runner image reference from:

   FROM gcr.io/distroless/static:nonroot

   to:

   FROM registry.access.redhat.com/ubi8/ubi-minimal:latest

2. Depending on the Go project version, your Dockerfile might contain a USER 65532:65532.
Depending on the Go project version, your Dockerfile might contain a `USER 65532:65532` or `USER nonroot:nonroot` directive. In either case, remove the line, as it is not required by the supported runner image.

3. In the `config/default/manager_auth_proxy_patch.yaml` file, change the `image` value from:
   ```
gcr.io/kubebuilder/kube-rbac-proxy:<tag>
   ```
to use the supported image:
   ```
registry.redhat.io/openshift4/ose-kube-rbac-proxy:v4.7
   ```

## 5.3.2.5.3. Running as a deployment on the cluster

You can run your Operator project as a deployment on your cluster.

### Prerequisites

- Prepared your Go-based Operator to run on OpenShift Container Platform by updating the project to use supported images

### Procedure

1. Run the following `make` commands to build and push the Operator image. Modify the `IMG` argument in the following steps to reference a repository that you have access to. You can obtain an account for storing containers at repository sites such as Quay.io.
   
   a. Build the image:
      
      ```
      $ make docker-build IMG=<registry>/<user>/<image_name>:<tag>
      ```
   
   b. Push the image to a repository:
      
      ```
      $ make docker-push IMG=<registry>/<user>/<image_name>:<tag>
      ```

   **NOTE**

   The name and tag of the image, for example `IMG=<registry>/<user>/<image_name>:<tag>`, in both the commands can also be set in your Makefile. Modify the `IMG ?= controller:latest` value to set your default image name.

2. Run the following command to deploy the Operator:
   
   ```
   $ make deploy IMG=<registry>/<user>/<image_name>:<tag>
   ```

   By default, this command creates a namespace with the name of your Operator project in the form `<project_name>-system` and is used for the deployment. This command also installs the RBAC manifests from `config/rbac`.

3. Verify that the Operator is running:
   
   ```
   $ oc get deployment -n <project_name>-system
   ```
5.3.2.5.4. Bundling an Operator and deploying with Operator Lifecycle Manager

Operator Lifecycle Manager (OLM) helps you to install, update, and generally manage the lifecycle of Operators and their associated services on a Kubernetes cluster. OLM is installed by default on OpenShift Container Platform and runs as a Kubernetes extension so that you can use the web console and the OpenShift CLI (\texttt{oc}) for all Operator lifecycle management functions without any additional tools.

The Operator Bundle Format is the default packaging method for Operator SDK and OLM. You can get your Operator ready for OLM by using the Operator SDK to build, push, validate, and run a bundle image with OLM.

Prerequisites

- Operator SDK CLI installed on a development workstation
- OpenShift CLI (\texttt{oc}) v4.7+ installed
- Operator Lifecycle Manager (OLM) installed on a Kubernetes-based cluster (v1.16.0 or later if you use \texttt{apiextensions.k8s.io/v1} CRDs, for example OpenShift Container Platform 4.7)
- Logged into the cluster with \texttt{oc} using an account with \texttt{cluster-admin} permissions
- Operator project initialized by using the Operator SDK
- If your Operator is Go-based, your project must have been updated to use supported images for running on OpenShift Container Platform

Procedure

1. Run the following \texttt{make} commands in your Operator project directory to build and push your Operator image. Modify the \texttt{IMG} argument in the following steps to reference a repository that you have access to. You can obtain an account for storing containers at repository sites such as Quay.io.
   
a. Build the image:
   
   \[
   \texttt{$ make docker-build IMG=<registry>/<user>/<operator_image_name>:<tag>}
   \]

   b. Push the image to a repository:
   
   \[
   \texttt{$ make docker-push IMG=<registry>/<user>/<operator_image_name>:<tag>}
   \]

2. Update your \texttt{Makefile} by setting the \texttt{IMG} URL to your Operator image name and tag that you pushed:

   \[
   \texttt{$ # Image URL to use all building/pushing image targets}
   \texttt{IMG ?= <registry>/<user>/<operator_image_name>:<tag>}
   \]
This value is used for subsequent operations.

3. Create your Operator bundle manifest by running the `make bundle` command, which invokes several commands, including the Operator SDK `generate bundle` and `bundle validate` subcommands:

   ```
   $ make bundle
   ```

Bundle manifests for an Operator describe how to display, create, and manage an application. The `make bundle` command creates the following files and directories in your Operator project:

- A bundle manifests directory named `bundle/manifests` that contains a `ClusterServiceVersion` object
- A bundle metadata directory named `bundle/metadata`
- All custom resource definitions (CRDs) in a `config/crd` directory
- A Dockerfile `bundle.Dockerfile`

These files are then automatically validated by using `operator-sdk bundle validate` to ensure the on-disk bundle representation is correct.

4. Build and push your bundle image by running the following commands. OLM consumes Operator bundles using an index image, which reference one or more bundle images.

   a. Build the bundle image. Set `BUNDLE_IMAGE` with the details for the registry, user namespace, and image tag where you intend to push the image:

   ```
   $ make bundle-build
   BUNDLE_IMG=<registry>/<user>/<bundle_image_name>:<tag>
   ```

   b. Push the bundle image:

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

5. Check the status of OLM on your cluster by using the following Operator SDK command:

   ```
   $ operator-sdk olm status \
   --olm-namespace=openshift-operator-lifecycle-manager
   ```

6. Run the Operator on your cluster by using the OLM integration in Operator SDK:

   ```
   $ operator-sdk run bundle \ 
   [-n <namespace>] 1 \ 
   <registry>/<user>/<bundle_image_name>:<tag>
   ```

   1 By default, the command installs the Operator in the currently active project in your `~/.kube/config` file. You can add the `-n` flag to set a different namespace scope for the installation.

This command performs the following actions:

- Create an index image with your bundle image injected.
• Create a catalog source that points to your new index image, which enables OperatorHub to
  discover your Operator.

• Deploy your Operator to your cluster by creating an Operator group, subscription, install
  plan, and all other required objects, including RBAC.

5.3.2.6. Creating a custom resource

After your Operator is installed, you can test it by creating a custom resource (CR) that is now provided
on the cluster by the Operator.

Prerequisites

• Example Memcached Operator, which provides the Memcached CR, installed on a cluster

Procedure

1. Change to the namespace where your Operator is installed. For example, if you deployed the
   Operator using the make deploy command:

   $ oc project memcached-operator-system

2. Edit the sample Memcached CR manifest at config/samples/cache_v1_memcached.yaml to
   contain the following specification:

   ```yaml
   apiVersion: cache.example.com/v1
   kind: Memcached
   metadata:
     name: memcached-sample
   ...  
   spec:
     ... 
     size: 3
   ```

3. Create the CR:

   $ oc apply -f config/samples/cache_v1_memcached.yaml

4. Ensure that the Memcached Operator creates the deployment for the sample CR with the
   correct size:

   $ oc get deployments

<table>
<thead>
<tr>
<th>NAME</th>
<th>READY</th>
<th>UP-TO-DATE</th>
<th>AVAILABLE</th>
<th>AGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>memcached-operator-controller-manager</td>
<td>1/1</td>
<td>1</td>
<td>1</td>
<td>8m</td>
</tr>
<tr>
<td>memcached-sample</td>
<td>3/3</td>
<td>3</td>
<td>3</td>
<td>1m</td>
</tr>
</tbody>
</table>

5. Check the pods and CR status to confirm the status is updated with the Memcached pod
   names.

   a. Check the pods:

   ```
   ```
$ oc get pods

**Example output**

<table>
<thead>
<tr>
<th>NAME</th>
<th>READY</th>
<th>STATUS</th>
<th>RESTARTS</th>
<th>AGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>memcached-sample-6fd7c98d8-7dqdr</td>
<td>1/1</td>
<td>Running</td>
<td>0</td>
<td>1m</td>
</tr>
<tr>
<td>memcached-sample-6fd7c98d8-g5k7v</td>
<td>1/1</td>
<td>Running</td>
<td>0</td>
<td>1m</td>
</tr>
<tr>
<td>memcached-sample-6fd7c98d8-m7vn7</td>
<td>1/1</td>
<td>Running</td>
<td>0</td>
<td>1m</td>
</tr>
</tbody>
</table>

b. Check the CR status:

$ oc get memcached/memcached-sample -o yaml

**Example output**

```yaml
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. Update the deployment size.

a. Update `config/samples/cache_v1_memcached.yaml` file to change the `spec.size` field in the Memcached CR from 3 to 5:

```bash
$ oc patch memcached memcached-sample \
   -p '{"spec":{"size": 5}}' \
   --type=merge
```

b. Confirm that the Operator changes the deployment size:

$ oc get deployments

**Example output**

<table>
<thead>
<tr>
<th>NAME</th>
<th>READY</th>
<th>UP-TO-DATE</th>
<th>AVAILABLE</th>
<th>AGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>memcached-operator-controller-manager</td>
<td>1/1</td>
<td>1</td>
<td>1</td>
<td>10m</td>
</tr>
<tr>
<td>memcached-sample</td>
<td>5/5</td>
<td>5</td>
<td>5</td>
<td>3m</td>
</tr>
</tbody>
</table>

7. Clean up the resources that have been created as part of this tutorial.

- If you used the **make deploy** command to test the Operator, run the following command:

  $ make undeploy
If you used the `operator-sdk run bundle` command to test the Operator, run the following command:

```
$ operator-sdk cleanup <project_name>
```

### 5.3.2.7. Additional resources

- See [Project layout for Go-based Operators](#) to learn about the directory structures created by the Operator SDK.

### 5.3.3. Project layout for Go-based Operators

The `operator-sdk` CLI can generate, or scaffold, a number of packages and files for each Operator project.

#### 5.3.3.1. Go-based project layout

Go-based Operator projects, the default type, generated using the `operator-sdk init` command contain the following files and directories:

<table>
<thead>
<tr>
<th>File or directory</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>main.go</strong></td>
<td>Main program of the Operator. This instantiates a new manager that registers all custom resource definitions (CRDs) in the <code>apis/</code> directory and starts all controllers in the <code>controllers/</code> directory.</td>
</tr>
<tr>
<td><strong>apis/</strong></td>
<td>Directory tree that defines the APIs of the CRDs. You must edit the <code>apis/&lt;version&gt;/&lt;kind&gt;_types.go</code> files to define the API for each resource type and import these packages in your controllers to watch for these resource types.</td>
</tr>
<tr>
<td><strong>controllers/</strong></td>
<td>Controller implementations. Edit the <code>controller/&lt;kind&gt;_controller.go</code> files to define the reconcile logic of the controller for handling a resource type of the specified kind.</td>
</tr>
<tr>
<td><strong>config/</strong></td>
<td>Kubernetes manifests used to deploy your controller on a cluster, including CRDs, RBAC, and certificates.</td>
</tr>
<tr>
<td><strong>Makefile</strong></td>
<td>Targets used to build and deploy your controller.</td>
</tr>
<tr>
<td><strong>Dockerfile</strong></td>
<td>Instructions used by a container engine to build your Operator.</td>
</tr>
<tr>
<td><strong>manifests/</strong></td>
<td>Kubernetes manifests for registering CRDs, setting up RBAC, and deploying the Operator as a deployment.</td>
</tr>
</tbody>
</table>

### 5.4. ANSIBLE-BASED OPERATORS

#### 5.4.1. Getting started with Operator SDK for Ansible-based Operators
The Operator SDK includes options for generating an Operator project that leverages existing Ansible playbooks and modules to deploy Kubernetes resources as a unified application, without having to write any Go code.

To demonstrate the basics of setting up and running an Ansible-based Operator using tools and libraries provided by the Operator SDK, Operator developers can build an example Ansible-based Operator for Memcached, a distributed key-value store, and deploy it to a cluster.

5.4.1.1. Prerequisites

- Operator SDK CLI installed
- OpenShift CLI (oc) v4.7+ installed
- Ansible version v2.9.0
- Ansible Runner version v1.1.0+
- Ansible Runner HTTP Event Emitter plug-in version v1.0.0+
- OpenShift Python client version v0.11.2+
- Logged into an OpenShift Container Platform 4.7 cluster with oc with an account that has cluster-admin permissions
- To allow the cluster pull the image, the repository where you push your image must be set as public, or you must configure an image pull secret.

5.4.1.2. Creating and deploying Ansible-based Operators

You can build and deploy a simple Ansible-based Operator for Memcached by using the Operator SDK.

Procedure

1. Create a project.
   a. Create your project directory:
      
      $ mkdir memcached-operator

   b. Change into the project directory:
      
      $ cd memcached-operator

   c. Run the operator-sdk init command with the ansible plug-in to initialize the project:
      
      $ operator-sdk init \ 
         --plugins=ansible \ 
         --domain=example.com

2. Create an API.
   Create a simple Memcached API:
      
      $ operator-sdk create api \ 
         --group cache \ 

Generates an Ansible role for the API.

3. Build and push the Operator image.
   Use the default `Makefile` targets to build and push your Operator. Set `IMG` with a pull spec for your image that uses a registry you can push to:

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

4. Run the Operator.
   a. Install the CRD:
      ```
      $ make install
      ```
   
   b. Deploy the project to the cluster. Set `IMG` to the image that you pushed:
      ```
      $ make deploy IMG=<registry>/<user>/<image_name>:<tag>
      ```

5. Create a sample custom resource (CR).
   a. Create a sample CR:
      ```
      $ oc apply -f config/samples/cache_v1_memcached.yaml \
        -n memcached-operator-system
      ```
   
   b. Watch for the CR to reconcile the Operator:
      ```
      $ oc logs deployment.apps/memcached-operator-controller-manager \
        -c manager \
        -n memcached-operator-system
      ```

Example output

```
...  
I0205 17:48:45.881666       7 leaderelection.go:253] successfully acquired lease 
memcached-operator-system/memcached-operator 
["level":"info","ts":1612547325.8819902,"logger":"controller-runtime.manager.controller.memcached-controller","msg":"Starting EventSource","source":"kind source: cache.example.com/v1, Kind=Memcached"]  
["level":"info","ts":1612547325.98242,"logger":"controller-runtime.manager.controller.memcached-controller","msg":"Starting Controller"]  
["level":"info","ts":1612547325.9824686,"logger":"controller-runtime.manager.controller.memcached-controller","msg":"Starting workers","worker count":4]  
["level":"info","ts":1612547348.8311093,"logger":"runner","msg":"Ansible-runner exited successfully","job":"4037200794235010051","name":"memcached-sample","namespace":"memcached-operator-system"]
```
6. **Clean up.**
   Run the following command to clean up the resources that have been created as part of this procedure:

   ```
   $ make undeploy
   ```

5.4.1.3. Next steps

- See [Operator SDK tutorial for Ansible-based Operators](#) for a more in-depth walkthrough on building an Ansible-based Operator.

5.4.2. Operator SDK tutorial for Ansible-based Operators

Operator developers can take advantage of Ansible support in the Operator SDK to build an example Ansible-based Operator for Memcached, a distributed key-value store, and manage its lifecycle. This tutorial walks through the following process:

- Create a Memcached deployment
- Ensure that the deployment size is the same as specified by the Memcached custom resource (CR) spec
- Update the Memcached CR status using the status writer with the names of the memcached pods

This process is accomplished by using two centerpieces of the Operator Framework:

**Operator SDK**

The `operator-sdk` CLI tool and `controller-runtime` library API

**Operator Lifecycle Manager (OLM)**

Installation, upgrade, and role-based access control (RBAC) of Operators on a cluster

---

**NOTE**

This tutorial goes into greater detail than [Getting started with Operator SDK for Ansible-based Operators](#).

5.4.2.1. Prerequisites

- Operator SDK CLI installed
- OpenShift CLI (`oc`) v4.7+ installed
- Ansible version v2.9.0
- Ansible Runner version v1.1.0+
- Ansible Runner HTTP Event Emitter plug-in version v1.0.0+
- OpenShift Python client version v0.11.2+
- Logged into an OpenShift Container Platform 4.7 cluster with `oc` with an account that has `cluster-admin` permissions
To allow the cluster pull the image, the repository where you push your image must be set as public, or you must configure an image pull secret.

5.4.2.2. Creating a project

Use the Operator SDK CLI to create a project called `memcached-operator`.

**Procedure**

1. Create a directory for the project:
   ```bash
   $ mkdir -p $HOME/projects/memcached-operator
   ```
2. Change to the directory:
   ```bash
   $ cd $HOME/projects/memcached-operator
   ```
3. Run the `operator-sdk init` command with the `ansible` plug-in to initialize the project:
   ```bash
   $ operator-sdk init
   --plugins=ansible
   --domain=example.com
   ```

5.4.2.2.1. PROJECT file

Among the files generated by the `operator-sdk init` command is a Kubebuilder `PROJECT` file. Subsequent `operator-sdk` commands, as well as `help` output, that are run from the project root read this file and are aware that the project type is Ansible. For example:

```yaml
domain: example.com
layout: ansible.sdk.operatorframework.io/v1
projectName: memcached-operator
version: 3-alpha
```

5.4.2.3. Creating an API

Use the Operator SDK CLI to create a Memcached API.

**Procedure**

- Run the following command to create an API with group `cache`, version, `v1`, and kind `Memcached`:
  ```bash
  $ operator-sdk create api
  --group cache
  --version v1
  --kind Memcached
  --generate-role
  ```

  Generates an Ansible role for the API.

After creating the API, your Operator project updates with the following structure:
Memcached CRD

Includes a sample Memcached resource

Manager

Program that reconciles the state of the cluster to the desired state by using:

- A reconciler, either an Ansible role or playbook
- A `watches.yaml` file, which connects the Memcached resource to the memcached Ansible role

5.4.2.4. Modifying the manager

Update your Operator project to provide the reconcile logic, in the form of an Ansible role, which runs every time a Memcached resource is created, updated, or deleted.

Procedure

1. Update the `roles/memcached/tasks/main.yml` file with the following structure:

   ```yaml
   ---
   - name: start memcached
     community.kubernetes.k8s:
       definition:
         kind: Deployment
         apiVersion: apps/v1
         metadata:
           name: '{{ ansible_operator_meta.name }}-memcached'
           namespace: '{{ ansible_operator_meta.namespace }}'
         spec:
           replicas: '{{size}}'
           selector:
             matchLabels:
               app: memcached
           template:
             metadata:
               labels:
                 app: memcached
             spec:
               containers:
                 - name: memcached
                   command:
                     - memcached
                     -m=64
                     -o
                     - modern
                     -v
                   image: "docker.io/memcached:1.4.36-alpine"
                   ports:
                     - containerPort: 11211
   
   This memcached role ensures a memcached deployment exist and sets the deployment size.
   
2. Set default values for variables used in your Ansible role by editing the `roles/memcached/defaults/main.yml` file:
3. Update the Memcached sample resource in the `config/samples/cache_v1_memcached.yaml` file with the following structure:

```yaml
apiVersion: cache.example.com/v1
kind: Memcached
metadata:
  name: memcached-sample
spec:
  size: 3
```

The key-value pairs in the custom resource (CR) spec are passed to Ansible as extra variables.

**NOTE**

The names of all variables in the `spec` field are converted to snake case, meaning lowercase with an underscore, by the Operator before running Ansible. For example, `serviceAccount` in the spec becomes `service_account` in Ansible.

You can disable this case conversion by setting the `snakeCaseParameters` option to `false` in your `watches.yaml` file. It is recommended that you perform some type validation in Ansible on the variables to ensure that your application is receiving expected input.

5.4.2.5. Running the Operator

There are three ways you can use the Operator SDK CLI to build and run your Operator:

- Run locally outside the cluster as a Go program.
- Run as a deployment on the cluster.
- Bundle your Operator and use Operator Lifecycle Manager (OLM) to deploy on the cluster.

5.4.2.5.1. Running locally outside the cluster

You can run your Operator project as a Go program outside of the cluster. This is useful for development purposes to speed up deployment and testing.

**Procedure**

- Run the following command to install the custom resource definitions (CRDs) in the cluster configured in your `~/.kube/config` file and run the Operator locally:

  ```bash
  $ make install run
  ```

**Example output**

```json
...{
  "level": "info",
  "ts": 1612589622.7888272,
  "logger": "ansible-controller",
...}
```
5.4.2.5.2. Preparing your Operator to use supported images

Before running your Ansible-based Operator on OpenShift Container Platform, update your project to use supported images.

Procedure

1. Update the project root-level Dockerfile to use supported images. Change the default builder image reference from:

   FROM quay.io/operator-framework/ansible-operator:v1.3.0

   to:

   FROM registry.redhat.io/openshift4/ose-ansible-operator:v4.7

   IMPORTANT

   Use the builder image version that matches your Operator SDK version. Failure to do so can result in problems due to project layout, or "scaffolding" differences, particularly when mixing newer upstream versions of the Operator SDK with downstream OpenShift Container Platform builder images.

2. In the config/default/manager_auth_proxy_patch.yaml file, change the image value from:

   gcr.io/kubebuilder/kube-rbac-proxy:<tag>

   to use the supported image:

   registry.redhat.io/openshift4/ose-kube-rbac-proxy:v4.7

5.4.2.5.3. Running as a deployment on the cluster

You can run your Operator project as a deployment on your cluster.

Procedure
1. Run the following `make` commands to build and push the Operator image. Modify the `IMG` argument in the following steps to reference a repository that you have access to. You can obtain an account for storing containers at repository sites such as Quay.io.

   a. Build the image:
   
   ```
   $ make docker-build IMG=<registry>/<user>/<image_name>:<tag>
   ```

   b. Push the image to a repository:
   
   ```
   $ make docker-push IMG=<registry>/<user>/<image_name>:<tag>
   ```

   **NOTE**

   The name and tag of the image, for example `IMG=<registry>/<user>/<image_name>:<tag>`, in both the commands can also be set in your Makefile. Modify the `IMG?=controller:latest` value to set your default image name.

2. Run the following command to deploy the Operator:

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

   By default, this command creates a namespace with the name of your Operator project in the form `<project_name>-system` and is used for the deployment. This command also installs the RBAC manifests from `config/rbac`.

3. Verify that the Operator is running:

   ```
   $ oc get deployment -n <project_name>-system
   ```

   **Example output**

<table>
<thead>
<tr>
<th>NAME</th>
<th>READY</th>
<th>UP-TO-DATE</th>
<th>AVAILABLE</th>
<th>AGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;project_name&gt;-controller-manager</td>
<td>1/1</td>
<td>1</td>
<td>1</td>
<td>8m</td>
</tr>
</tbody>
</table>

---

5.4.2.5.4. Bundling an Operator and deploying with Operator Lifecycle Manager

Operator Lifecycle Manager (OLM) helps you to install, update, and generally manage the lifecycle of Operators and their associated services on a Kubernetes cluster. OLM is installed by default on OpenShift Container Platform and runs as a Kubernetes extension so that you can use the web console and the OpenShift CLI (`oc`) for all Operator lifecycle management functions without any additional tools.

The Operator Bundle Format is the default packaging method for Operator SDK and OLM. You can get your Operator ready for OLM by using the Operator SDK to build, push, validate, and run a bundle image with OLM.

**Prerequisites**

- Operator SDK CLI installed on a development workstation
- OpenShift CLI (`oc`) v4.7+ installed
Operator Lifecycle Manager (OLM) installed on a Kubernetes-based cluster (v1.16.0 or later if you use `apiextensions.k8s.io/v1` CRDs, for example OpenShift Container Platform 4.7)

- Logged into the cluster with `oc` using an account with `cluster-admin` permissions
- Operator project initialized by using the Operator SDK

**Procedure**

1. Run the following `make` commands in your Operator project directory to build and push your Operator image. Modify the `IMG` argument in the following steps to reference a repository that you have access to. You can obtain an account for storing containers at repository sites such as Quay.io.

   a. Build the image:

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

   b. Push the image to a repository:

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

2. Update your `Makefile` by setting the `IMG` URL to your Operator image name and tag that you pushed:

   ```
   # Image URL to use all building/pushing image targets
   IMG ?= <registry>/<user>/<operator_image_name>:<tag>
   ```

   This value is used for subsequent operations.

3. Create your Operator bundle manifest by running the `make bundle` command, which invokes several commands, including the Operator SDK `generate bundle` and `bundle validate` subcommands:

   ```
   $ make bundle
   ```

   Bundle manifests for an Operator describe how to display, create, and manage an application. The `make bundle` command creates the following files and directories in your Operator project:

   - A bundle manifests directory named `bundle/manifests` that contains a `ClusterServiceVersion` object
   - A bundle metadata directory named `bundle/metadata`
   - All custom resource definitions (CRDs) in a `config/crd` directory
   - A Dockerfile `bundle.Dockerfile`

   These files are then automatically validated by using `operator-sdk bundle validate` to ensure the on-disk bundle representation is correct.

4. Build and push your bundle image by running the following commands. OLM consumes Operator bundles using an index image, which reference one or more bundle images.

   a. Build the bundle image. Set `BUNDLE_IMAGE` with the details for the registry, user namespace, and image tag where you intend to push the image:
$ make bundle-build BUNDLE_IMG=<registry>/<user>/<bundle_image_name>:<tag>

b. Push the bundle image:

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

5. Check the status of OLM on your cluster by using the following Operator SDK command:

$ operator-sdk olm status \\
   --olm-namespace=openshift-operator-lifecycle-manager

6. Run the Operator on your cluster by using the OLM integration in Operator SDK:

$ operator-sdk run bundle \\
   [-n <namespace>] <registry>/<user>/<bundle_image_name>:<tag>

By default, the command installs the Operator in the currently active project in your 
~/.kube/config file. You can add the -n flag to set a different namespace scope for the
installation.

This command performs the following actions:

- Create an index image with your bundle image injected.
- Create a catalog source that points to your new index image, which enables OperatorHub to
discover your Operator.
- Deploy your Operator to your cluster by creating an Operator group, subscription, install
plan, and all other required objects, including RBAC.

5.4.2.6. Creating a custom resource

After your Operator is installed, you can test it by creating a custom resource (CR) that is now provided
on the cluster by the Operator.

Prerequisites

- Example Memcached Operator, which provides the Memcached CR, installed on a cluster

Procedure

1. Change to the namespace where your Operator is installed. For example, if you deployed the
Operator using the make deploy command:

   $ oc project memcached-operator-system

2. Edit the sample Memcached CR manifest at config/samples/cache_v1_memcached.yaml to
contain the following specification:

   apiVersion: cache.example.com/v1
   kind: Memcached
Create the CR:

```
$ oc apply -f config/samples/cache_v1_memcached.yaml
```

4. Ensure that the **Memcached** Operator creates the deployment for the sample CR with the correct size:

```
$ oc get deployments
```

**Example output**

<table>
<thead>
<tr>
<th>NAME</th>
<th>READY</th>
<th>UP-TO-DATE</th>
<th>AVAILABLE</th>
<th>AGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>memcached-operator-controller-manager</td>
<td>1/1</td>
<td>1</td>
<td>1</td>
<td>8m</td>
</tr>
<tr>
<td>memcached-sample</td>
<td>3/3</td>
<td>3</td>
<td>3</td>
<td>1m</td>
</tr>
</tbody>
</table>

5. Check the pods and CR status to confirm the status is updated with the Memcached pod names.

a. Check the pods:

```
$ oc get pods
```

**Example output**

<table>
<thead>
<tr>
<th>NAME</th>
<th>READY</th>
<th>STATUS</th>
<th>RESTARTS</th>
<th>AGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>memcached-sample-6fd7c98d8-7dqdr</td>
<td>1/1</td>
<td>Running</td>
<td>0</td>
<td>1m</td>
</tr>
<tr>
<td>memcached-sample-6fd7c98d8-g5k7v</td>
<td>1/1</td>
<td>Running</td>
<td>0</td>
<td>1m</td>
</tr>
<tr>
<td>memcached-sample-6fd7c98d8-m7vn7</td>
<td>1/1</td>
<td>Running</td>
<td>0</td>
<td>1m</td>
</tr>
</tbody>
</table>

b. Check the CR status:

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

**Example output**

```yaml
apiVersion: cache.example.com/v1
kind: Memcached
metadata:
  name: memcached-sample

spec:
  size: 3
status:
  nodes:
```
6. Update the deployment size.
   a. Update `config/samples/cache_v1_memcached.yaml` file to change the `spec.size` field in the `Memcached` CR from 3 to 5:

   ```bash
   $ oc patch memcached memcached-sample \
   -p '{"spec":{"size": 5}}' \
   --type=merge
   ```

   b. Confirm that the Operator changes the deployment size:

   ```bash
   $ oc get deployments
   ```

   **Example output**

<table>
<thead>
<tr>
<th>NAME</th>
<th>READY</th>
<th>UP-TO-DATE</th>
<th>AVAILABLE</th>
<th>AGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>memcached-operator-controller-manager</td>
<td>1/1</td>
<td>1</td>
<td>1</td>
<td>10m</td>
</tr>
<tr>
<td>memcached-sample</td>
<td>5/5</td>
<td>5</td>
<td>5</td>
<td>3m</td>
</tr>
</tbody>
</table>

7. Clean up the resources that have been created as part of this tutorial.
   - If you used the `make deploy` command to test the Operator, run the following command:

   ```bash
   $ make undeploy
   ```

   - If you used the `operator-sdk run bundle` command to test the Operator, run the following command:

   ```bash
   $ operator-sdk cleanup <project_name>
   ```

5.4.2.7. Additional resources
   - See [Project layout for Ansible-based Operators](#) to learn about the directory structures created by the Operator SDK.

5.4.3. Project layout for Ansible-based Operators

The `operator-sdk` CLI can generate, or `scaffold`, a number of packages and files for each Operator project.

5.4.3.1. Ansible-based project layout

Ansible-based Operator projects generated using the `operator-sdk init --plugins ansible` command contain the following directories and files:
<table>
<thead>
<tr>
<th>File or directory</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dockerfile</td>
<td>Dockerfile for building the container image for the Operator.</td>
</tr>
<tr>
<td>Makefile</td>
<td>Targets for building, publishing, deploying the container image that wraps the Operator binary, and targets for installing and uninstalling the custom resource definition (CRD).</td>
</tr>
<tr>
<td>PROJECT</td>
<td>YAML file containing metadata information for the Operator.</td>
</tr>
<tr>
<td>config/crd</td>
<td>Base CRD files and the kustomization.yaml file settings.</td>
</tr>
<tr>
<td>config/default</td>
<td>Collects all Operator manifests for deployment. Use by the make deploy command.</td>
</tr>
<tr>
<td>config/manager</td>
<td>Controller manager deployment.</td>
</tr>
<tr>
<td>config/prometheus</td>
<td>ServiceMonitor resource for monitoring the Operator.</td>
</tr>
<tr>
<td>config/rbac</td>
<td>Role and role binding for leader election and authentication proxy.</td>
</tr>
<tr>
<td>config/samples</td>
<td>Sample resources created for the CRDs.</td>
</tr>
<tr>
<td>config/testing</td>
<td>Sample configurations for testing.</td>
</tr>
<tr>
<td>playbooks/</td>
<td>A subdirectory for the playbooks to run.</td>
</tr>
<tr>
<td>roles/</td>
<td>Subdirectory for the roles tree to run.</td>
</tr>
<tr>
<td>watches.yaml</td>
<td>Group/version/kind (GVK) of the resources to watch, and the Ansible invocation method. New entries are added by using the create api command.</td>
</tr>
<tr>
<td>requirements.yml</td>
<td>YAML file containing the Ansible collections and role dependencies to install during a build.</td>
</tr>
<tr>
<td>molecule/</td>
<td>Molecule scenarios for end-to-end testing of your role and Operator.</td>
</tr>
</tbody>
</table>

### 5.4.4. Ansible support in Operator SDK

#### 5.4.4.1. Custom resource files

Operators use the Kubernetes extension mechanism, custom resource definitions (CRDs), so your custom resource (CR) looks and acts just like the built-in, native Kubernetes objects.

The CR file format is a Kubernetes resource file. The object has mandatory and optional fields:

<p>| Table 5.1. Custom resource fields |</p>
<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>apiVersion</td>
<td>Version of the CR to be created.</td>
</tr>
<tr>
<td>kind</td>
<td>Kind of the CR to be created.</td>
</tr>
<tr>
<td>metadata</td>
<td>Kubernetes-specific metadata to be created.</td>
</tr>
<tr>
<td>spec (optional)</td>
<td>Key-value list of variables which are passed to Ansible. This field is empty by default.</td>
</tr>
<tr>
<td>status</td>
<td>Summarizes the current state of the object. For Ansible-based Operators, the status subresource is enabled for CRDs and managed by the operator_sdk.util.k8s_status Ansible module by default, which includes condition information to the CR status.</td>
</tr>
<tr>
<td>annotations</td>
<td>Kubernetes-specific annotations to be appended to the CR.</td>
</tr>
</tbody>
</table>

The following list of CR annotations modify the behavior of the Operator:

**Table 5.2. Ansible-based Operator annotations**

<table>
<thead>
<tr>
<th>Annotation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ansible.operator-sdk/reconcile-period</td>
<td>Specifies the reconciliation interval for the CR. This value is parsed using the standard Golang package time. Specifically, ParseDuration is used which applies the default suffix of s, giving the value in seconds.</td>
</tr>
</tbody>
</table>

**Example Ansible-based Operator annotation**

```yaml
apiVersion: "test1.example.com/v1alpha1"
kind: "Test1"
metadata:
  name: "example"
annotations:
  ansible.operator-sdk/reconcile-period: "30s"
```

**5.4.4.2. watches.yaml file**

A group/version/kind (GVK) is a unique identifier for a Kubernetes API. The watches.yaml file contains a list of mappings from custom resources (CRs), identified by its GVK, to an Ansible role or playbook. The Operator expects this mapping file in a predefined location at /opt/ansible/watches.yaml.

**Table 5.3. watches.yaml file mappings**

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>group</td>
<td>Group of CR to watch.</td>
</tr>
</tbody>
</table>
Field | Description
--- | ---
version | Version of CR to watch.
kind | Kind of CR to watch
role (default) | Path to the Ansible role added to the container. For example, if your roles directory is at /opt/ansible/roles/ and your role is named busybox, this value would be /opt/ansible/roles/busybox. This field is mutually exclusive with the playbook field.
playbook | Path to the Ansible playbook added to the container. This playbook is expected to be a way to call roles. This field is mutually exclusive with the role field.
reconcilePeriod (optional) | The reconciliation interval, how often the role or playbook is run, for a given CR.
manageStatus (optional) | When set to true (default), the Operator manages the status of the CR generically. When set to false, the status of the CR is managed elsewhere, by the specified role or playbook or in a separate controller.

Example watches.yaml file

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

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

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

1 Simple example mapping Test1 to the test1 role.
2 Simple example mapping Test2 to a playbook.
3 More complex example for the Test3 kind. Disables re-queuing and managing the CR status in the playbook.

5.4.4.2.1. Advanced options
Advanced features can be enabled by adding them to your `watches.yaml` file per GVK. They can go below the `group, version, kind` and `playbook` or `role` fields.

Some features can be overridden per resource using an annotation on that CR. The options that can be overridden have the annotation specified below.

### Table 5.4. Advanced watches.yaml file options

<table>
<thead>
<tr>
<th>Feature</th>
<th>YAML key</th>
<th>Description</th>
<th>Annotation for override</th>
<th>Default value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reconcile period</td>
<td>reconcilePeriod</td>
<td>Time between reconcile runs for a particular CR.</td>
<td>ansible/operator-sdk/reconcile-period</td>
<td>1m</td>
</tr>
<tr>
<td>Manage status</td>
<td>manageStatus</td>
<td>Allows the Operator to manage the <code>conditions</code> section of each CR <code>status</code> section.</td>
<td></td>
<td>true</td>
</tr>
<tr>
<td>Watch dependent resources</td>
<td>watchDependentResources</td>
<td>Allows the Operator to dynamically watch resources that are created by Ansible.</td>
<td></td>
<td>true</td>
</tr>
<tr>
<td>Watch cluster-scoped resources</td>
<td>watchClusterScopedResources</td>
<td>Allows the Operator to watch cluster-scoped resources that are created by Ansible.</td>
<td></td>
<td>false</td>
</tr>
<tr>
<td>Max runner artifacts</td>
<td>maxRunnerArtifacts</td>
<td>Manages the number of artifact directories that Ansible Runner keeps in the Operator container for each individual resource.</td>
<td>ansible/operator-sdk/max-runner-artifacts</td>
<td>20</td>
</tr>
</tbody>
</table>

### Example watches.yml file with advanced options

```yaml
- version: v1alpha1
  group: app.example.com
  kind: AppService
  playbook: /opt/ansible/playbook.yml
  maxRunnerArtifacts: 30
  reconcilePeriod: 5s
  manageStatus: False
  watchDependentResources: False
```

### 5.4.4.3. Extra variables sent to Ansible

Extra variables can be sent to Ansible, which are then managed by the Operator. The `spec` section of the custom resource (CR) passes along the key-value pairs as extra variables. This is equivalent to extra variables passed in to the `ansible-playbook` command.
The Operator also passes along additional variables under the **meta** field for the name of the CR and the namespace of the CR.

For the following CR example:

```yaml
apiVersion: "app.example.com/v1alpha1"
kind: "Database"
metadata:
  name: "example"
spec:
  message: "Hello world 2"
  newParameter: "newParam"
```

The structure passed to Ansible as extra variables is:

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

The **message** and **newParameter** fields are set in the top level as extra variables, and **meta** provides the relevant metadata for the CR as defined in the Operator. The **meta** fields can be accessed using dot notation in Ansible, for example:

```yaml
- debug:
  msg: "name: {{ ansible_operator_meta.name }}, {{ ansible_operator_meta.namespace }}"
```

### 5.4.4.4. Ansible Runner directory

Ansible Runner keeps information about Ansible runs in the container. This is located at `/tmp/ansible-operator/runner/<group>/<version>/<kind>/<namespace>/<name>`.

**Additional resources**

- To learn more about the **runner** directory, see the [Ansible Runner documentation](https://docs.openshift.com/container-platform/4.7/ansible/ansible-runner.html).

### 5.4.5. Kubernetes Collection for Ansible

To manage the lifecycle of your application on Kubernetes using Ansible, you can use the [Kubernetes Collection for Ansible](https://github.com/kubernetes-incubator/kube-ansible). This collection of Ansible modules allows a developer to either leverage their existing Kubernetes resource files written in YAML or express the lifecycle management in native Ansible.

One of the biggest benefits of using Ansible in conjunction with existing Kubernetes resource files is the ability to use Jinja templating so that you can customize resources with the simplicity of a few variables in Ansible.
This section goes into detail on usage of the Kubernetes Collection. To get started, install the collection on your local workstation and test it using a playbook before moving on to using it within an Operator.

5.4.5.1. Installing the Kubernetes Collection for Ansible

You can install the Kubernetes Collection for Ansible on your local workstation.

**Procedure**

1. Install Ansible 2.9+:
   
   ```
   $ sudo dnf install ansible
   ```

2. Install the OpenShift python client package:

   ```
   $ pip3 install openshift
   ```

3. Install the Kubernetes Collection using one of the following methods:
   
   - You can install the collection directly from Ansible Galaxy:
     ```
     $ ansible-galaxy collection install community.kubernetes
     ```
   - If you have already initialized your Operator, you might have a `requirements.yml` file at the top level of your project. This file specifies Ansible dependencies that must be installed for your Operator to function. By default, this file installs the `community.kubernetes` collection as well as the `operator_sdk.util` collection, which provides modules and plug-ins for Operator-specific functions.
     To install the dependent modules from the `requirements.yml` file:
     ```
     $ ansible-galaxy collection install -r requirements.yml
     ```

5.4.5.2. Testing the Kubernetes Collection locally

Operator developers can run the Ansible code from their local machine as opposed to running and rebuilding the Operator each time.

**Prerequisites**

- Initialize an Ansible-based Operator project and create an API that has a generated Ansible role by using the Operator SDK
- Install the Kubernetes Collection for Ansible

**Procedure**

1. In your Ansible-based Operator project directory, modify the `roles/<kind>/tasks/main.yml` file with the Ansible logic that you want. The `roles/<kind>/` directory is created when you use the `--generate-role` flag while creating an API. The `<kind>` replaceable matches the kind that you specified for the API.
   The following example creates and deletes a config map based on the value of a variable named `state`:
1. Change this value if you want the config map to be created in a different namespace from `default`.

2. Setting `ignore_errors: true` ensures that deleting a nonexistent config map does not fail.

2. Modify the `roles/<kind>/defaults/main.yml` file to set `state` to `present` by default:

   ```yaml
   ---
   state: present
   ```

3. Create an Ansible playbook by creating a `playbook.yml` file in the top-level of your project directory, and include your `<kind>` role:

   ```yaml
   ---
   - hosts: localhost
     roles:
       - <kind>
   ```

4. Run the playbook:

   ```bash
   $ ansible-playbook playbook.yml
   ```

**Example output**

```
[WARNING]: provided hosts list is empty, only localhost is available. Note that the implicit localhost does not match `all`

PLAY [localhost] ***********************************************************
TASK [Gathering Facts] ***********************************************************
ok: [localhost]

TASK [memcached : set ConfigMap example-config to present] ***********************************************************
changed: [localhost]

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

5. Verify that the config map was created:
6. Rerun the playbook setting `state` to `absent`:

```
$ ansible-playbook playbook.yml --extra-vars state=absent
```

Example output

```
[WARNING]: provided hosts list is empty, only localhost is available. Note that the implicit localhost does not match `all`
PLAY [localhost] **********************************************************************************************
TASK [Gathering Facts]                                                                                       
*****************************************************************************
ok: [localhost]
TASK [memcached : set ConfigMap example-config to absent]                                                      
*****************************************************************************
changed: [localhost]
PLAY RECAP ****************************************************************************************************
localhost : ok=2    changed=1    unreachable=0    failed=0    skipped=0    rescued=0    ignored=0
```

7. Verify that the config map was deleted:

```
$ oc get configmaps
```

5.4.5.3. Next steps

- See Using Ansible inside an Operator for details on triggering your custom Ansible logic inside of an Operator when a custom resource (CR) changes.

5.4.6. Using Ansible inside an Operator

After you are familiar with using the Kubernetes Collection for Ansible locally, you can trigger the same Ansible logic inside of an Operator when a custom resource (CR) changes. This example maps an Ansible role to a specific Kubernetes resource that the Operator watches. This mapping is done in the `watches.yaml` file.

5.4.6.1. Custom resource files

Operators use the Kubernetes extension mechanism, custom resource definitions (CRDs), so your custom resource (CR) looks and acts just like the built-in, native Kubernetes objects.

The CR file format is a Kubernetes resource file. The object has mandatory and optional fields:
Table 5.5. Custom resource fields

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>apiVersion</td>
<td>Version of the CR to be created.</td>
</tr>
<tr>
<td>kind</td>
<td>Kind of the CR to be created.</td>
</tr>
<tr>
<td>metadata</td>
<td>Kubernetes-specific metadata to be created.</td>
</tr>
<tr>
<td>spec (optional)</td>
<td>Key-value list of variables which are passed to Ansible. This field is empty by default.</td>
</tr>
<tr>
<td>status</td>
<td>Summarizes the current state of the object. For Ansible-based Operators, the status subresource is enabled for CRDs and managed by the <code>operator_sdk.util.k8s_status</code> Ansible module by default, which includes condition information to the CR status.</td>
</tr>
<tr>
<td>annotations</td>
<td>Kubernetes-specific annotations to be appended to the CR.</td>
</tr>
</tbody>
</table>

The following list of CR annotations modify the behavior of the Operator:

Table 5.6. Ansible-based Operator annotations

<table>
<thead>
<tr>
<th>Annotation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ansible.operator-sdk/reconcile-period</td>
<td>Specifies the reconciliation interval for the CR. This value is parsed using the standard Golang package <code>time</code>. Specifically, <code>ParseDuration</code> is used which applies the default suffix of <code>s</code>, giving the value in seconds.</td>
</tr>
</tbody>
</table>

Example Ansible-based Operator annotation

```yaml
apiVersion: "test1.example.com/v1alpha1"
kind: "Test1"
metadata:
  name: "example"
annotations:
  ansible.operator-sdk/reconcile-period: "30s"
```

5.4.6.2. Testing an Ansible-based Operator locally

You can test the logic inside of an Ansible-based Operator running locally by using the `make run` command from the top-level directory of your Operator project. The `make run` Makefile target runs the `ansible-operator` binary locally, which reads from the `watches.yaml` file and uses your `~/.kube/config` file to communicate with a Kubernetes cluster just as the `k8s` modules do.
NOTE
You can customize the roles path by setting the environment variable `ANSIBLE_ROLES_PATH` or by using the `ansible-roles-path` flag. If the role is not found in the `ANSIBLE_ROLES_PATH` value, the Operator looks for it in `{{current directory}}/roles`.

Prerequisites

- **Ansible Runner** version v1.1.0+
- **Ansible Runner HTTP Event Emitter plug-in** version v1.0.0+
- Performed the previous steps for testing the Kubernetes Collection locally

Procedure

1. Install your custom resource definition (CRD) and proper role-based access control (RBAC) definitions for your custom resource (CR):

```
$ make install
```

**Example output**

```
/usr/bin/kustomize build config/crd | kubectl apply -f -
customresourcedefinition.apiextensions.k8s.io/memcacheds.cache.example.com created
```

2. Run the `make run` command:

```
$ make run
```

**Example output**

```
/home/user/memcached-operator/bin/ansible-operator run
("level":"info","ts":1612739145.2871568,"logger":"cmd","msg":"Version","Go Version":"go1.15.5","GOOS":"linux","GOARCH":"amd64","ansible-operator":"v1.3.0","commit":"1abf57985b43bf6a59dcd18147b3c574fa57d3f6")
...
("level":"info","ts":1612739148.347306,"logger":"controller-runtime.metrics","msg":"metrics server is starting to listen","addr":":8080")
{"level":"info","ts":1612739148.3488882,"logger":"watches","msg":"Environment variable not set; using default value","envVar":"ANSIBLE_VERBOSITY_MEMCACHED_CACHE_EXAMPLE_COM","default":2}
{"level":"info","ts":1612739148.3490262,"logger":"controller-runtime.metrics","msg":"Environment variable not set; using default value","Namespace":"","envVar":"ANSIBLE_DEBUG_LOGS","ANSIBLE_DEBUG_LOGS":false}
{"level":"info","ts":1612739148.350217,"logger":"proxy","msg":"Starting to serve","Address":"127.0.0.1:8888"}
{"level":"info","ts":1612739148.3506632,"logger":"controller-runtime.manager","msg":"starting
```
With the Operator now watching your CR for events, the creation of a CR will trigger your Ansible role to run.

```
NOTE

Consider an example config/samples/<gvk>.yaml CR manifest:

apiVersion: <group>.example.com/v1alpha1
kind: <kind>
metadata:
  name: "<kind>-sample"

Because the spec field is not set, Ansible is invoked with no extra variables. Passing extra variables from a CR to Ansible is covered in another section. It is important to set reasonable defaults for the Operator.
```

3. Create an instance of your CR with the default variable state set to present:

```
$ oc apply -f config/samples/<gvk>.yaml
```

4. Check that the example-config config map was created:

```
$ oc get configmaps
```

**Example output**

<table>
<thead>
<tr>
<th>NAME</th>
<th>STATUS</th>
<th>AGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>example-config</td>
<td>Active</td>
<td>3s</td>
</tr>
</tbody>
</table>

5. Modify your config/samples/<gvk>.yaml file to set the state field to absent. For example:

```
apiVersion: cache.example.com/v1
kind: Memcached
metadata:
  name: memcached-sample
spec:
  state: absent
```

6. Apply the changes:

```
$ oc apply -f config/samples/<gvk>.yaml
```
7. Confirm that the config map is deleted:

```
$ oc get configmap
```

5.4.6.3. Testing an Ansible-based Operator on the cluster

After you have tested your custom Ansible logic locally inside of an Operator, you can test the Operator inside of a pod on an OpenShift Container Platform cluster, which is preferred for production use.

You can run your Operator project as a deployment on your cluster.

Procedure

1. Run the following `make` commands to build and push the Operator image. Modify the `IMG` argument in the following steps to reference a repository that you have access to. You can obtain an account for storing containers at repository sites such as Quay.io.

   a. Build the image:

   ```
   $ make docker-build IMG=<registry>/<user>/<image_name>:<tag>
   
   b. Push the image to a repository:

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

   **NOTE**

   The name and tag of the image, for example `IMG= <registry>/<user>/<image_name>:<tag>`, in both the commands can also be set in your Makefile. Modify the `IMG ?= controller:latest` value to set your default image name.

2. Run the following command to deploy the Operator:

   ```
   $ make deploy IMG=<registry>/<user>/<image_name>:<tag>
   
   By default, this command creates a namespace with the name of your Operator project in the form `<project_name>-system` and is used for the deployment. This command also installs the RBAC manifests from `config/rbac`.

3. Verify that the Operator is running:

   ```
   $ oc get deployment -n <project_name>-system
   
   Example output

   ```
   NAME                                    READY   UP-TO-DATE   AVAILABLE   AGE
   <project_name>-controller-manager       1/1     1            1           8m
   ```

5.4.6.4. Ansible logs

Ansible-based Operators provide logs about the Ansible run, which can be useful for debugging your code.

```
$ oc get configmap
```

**Example output**

```
NAME                                    READY   UP-TO-DATE   AVAILABLE   AGE
<project_name>-controller-manager       1/1     1            1           8m
```
Ansible-based Operators provide logs about the Ansible run, which can be useful for debugging your Ansible tasks. The logs can also contain detailed information about the internals of the Operator and its interactions with Kubernetes.

5.4.6.4.1. Viewing Ansible logs

Prerequisites

- Ansible-based Operator running as a deployment on a cluster

Procedure

- To view logs from an Ansible-based Operator, run the following command:

  ```bash
  $ oc logs deployment/<project_name>-controller-manager \
  -c manager \1
  -n <namespace> \2
  
  1 View logs from the manager container.
  2 If you used the make deploy command to run the Operator as a deployment, use the <project_name>-system namespace.

Example output

```json
{"level":"info","ts":1612732105.0579333,"logger":"cmd","msg":"Version","Go Version":"go1.15.5","GOOS":"linux","GOARCH":"amd64","ansible-operator":"v1.3.0","commit":"1abf57985b43b6a59dcd18147b3c574fa5fd3f6"}
{"level":"info","ts":1612732105.0587437,"logger":"cmd","msg":"WATCH_NAMESPACE environment variable not set. Watching all namespaces.","Namespace":"

{"level":"info","ts":1612732107.768025,"logger":"controller-runtime.metrics","msg":"metrics server is starting to listen","addr":"127.0.0.1:8080"}
{"level":"info","ts":1612732107.768796,"logger":"watches","msg":"Environment variable not set; using default value","envVar":"ANSIBLE_VERBOSITY_MEMCACHED_CACHE_EXAMPLE_COM","default":2}
{"level":"info","ts":1612732107.7688773,"logger":"cmd","msg":"Environment variable not set; using default value","Namespace":"","envVar":"ANSIBLE_DEBUG_LOGS","ANSIBLE_DEBUG_LOGS":false}
{"level":"info","ts":1612732107.7688901,"logger":"ansible-controller","msg":"Starting to serve","Address":"127.0.0.1:8888"}
I0207 21:08:27.770185       7 leaderelection.go:243] attempting to acquire leader lease memcached-operator-system/memcached-operator...
{"level":"info","ts":1612732107.770202,"logger":"controller-runtime.manager","msg":"starting metrics server","path":"/metrics"}
{"level":"info","ts":1612732107.7850506,"logger":"controller-
5.4.6.4.2. Enabling full Ansible results in logs

You can set the environment variable `ANSIBLE_DEBUG_LOGS` to `True` to enable checking the full Ansible result in logs, which can be helpful when debugging.

Procedure

- Edit the `config/manager/manager.yaml` and `config/default/manager_auth_proxy_patch.yaml` files to include the following configuration:

```
containers:
  - name: manager
    env:
      - name: ANSIBLE_DEBUG_LOGS
        value: "True"
```

5.4.6.4.3. Enabling verbose debugging in logs

While developing an Ansible-based Operator, it can be helpful to enable additional debugging in logs.

Procedure

- Add the `ansible.sdk.operatorframework.io/verbosity` annotation to your custom resource to enable the verbosity level that you want. For example:

```
apiVersion: "cache.example.com/v1alpha1"
kind: "Memcached"
metadata:
  name: "example-memcached"
  annotations:
    "ansible.sdk.operatorframework.io/verbosity": "4"
spec:
  size: 4
```

5.4.7. Custom resource status management

5.4.7.1. About custom resource status in Ansible-based Operators

Ansible-based Operators automatically update custom resource (CR) `status` subresources with generic information about the previous Ansible run. This includes the number of successful and failed tasks and relevant error messages as shown:

```
status:
  conditions:
    - ansibleResult:
```
Ansible-based Operators also allow Operator authors to supply custom status values with the `k8s_status` Ansible module, which is included in the `operator_sdk.util` collection. This allows the author to update the `status` from within Ansible with any key-value pair as desired.

By default, Ansible-based Operators always include the generic Ansible run output as shown above. If you would prefer your application did not update the status with Ansible output, you can track the status manually from your application.

### 5.4.7.2. Tracking custom resource status manually

You can use the `operator_sdk.util` collection to modify your Ansible-based Operator to track custom resource (CR) status manually from your application.

**Prerequisites**

- Ansible-based Operator project created by using the Operator SDK

**Procedure**

1. Update the `watches.yaml` file with a `manageStatus` field set to `false`:

   ```yaml
   - version: v1
     group: api.example.com
     kind: <kind>
     role: <role>
     manageStatus: false
   ```

2. Use the `operator_sdk.util.k8s_status` Ansible module to update the subresource. For example, to update with key `test` and value `data`, `operator_sdk.util` can be used as shown:

   ```yaml
   - operator_sdk.util.k8s_status:
     api_version: app.example.com/v1
     kind: <kind>
     name: "\{ ansible_operator_meta.name \}"
     namespace: "\{ ansible_operator_meta.namespace \}"
     status:
       test: data
   ```
3. You can declare collections in the `meta/main.yml` file for the role, which is included for scaffolded Ansible-based Operators:

```yaml
collections:
  - operator_sdk.util

4. After declaring collections in the role meta, you can invoke the `k8s_status` module directly:

```yaml
k8s_status:
  ...
  status:
    key1: value1
```

### 5.5. HELM-BASED OPERATORS

#### 5.5.1. Getting started with Operator SDK for Helm-based Operators

The Operator SDK includes options for generating an Operator project that leverages existing Helm charts to deploy Kubernetes resources as a unified application, without having to write any Go code.

To demonstrate the basics of setting up and running a Helm-based Operator using tools and libraries provided by the Operator SDK, Operator developers can build an example Helm-based Operator for Nginx and deploy it to a cluster.

#### 5.5.1.1. Prerequisites

- Operator SDK CLI installed
- OpenShift CLI (`oc`) v4.7+ installed
- Logged into an OpenShift Container Platform 4.7 cluster with `oc` with an account that has `cluster-admin` permissions
- To allow the cluster pull the image, the repository where you push your image must be set as public, or you must configure an image pull secret.

#### 5.5.1.2. Creating and deploying Helm-based Operators

You can build and deploy a simple Helm-based Operator for Nginx by using the Operator SDK.

**Procedure**

1. **Create a project.**
   a. Create your project directory:

   ```bash
   $ mkdir nginx-operator
   ```

   b. Change into the project directory:

   ```bash
   $ cd nginx-operator
   ```

   c. Run the `operator-sdk init` command with the `helm` plug-in to initialize the project:
2. Create an API.
Create a simple Nginx API:

   $ operator-sdk init \ 
   --plugins=helm

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

This API uses the built-in Helm chart boilerplate from the `helm create` command.

3. Build and push the Operator image.
Use the default Makefile targets to build and push your Operator. Set `IMG` with a pull spec for your image that uses a registry you can push to:

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

4. Run the Operator.
   a. Install the CRD:

      $ make install

   b. Deploy the project to the cluster. Set `IMG` to the image that you pushed:

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

5. Add a security context constraint (SCC).
The Nginx service account requires privileged access to run in OpenShift Container Platform. Add the following SCC to the service account for the `nginx-sample` pod:

   $ oc adm policy add-scc-to-user \ 
   anyuid system:serviceaccount:nginx-operator-system:nginx-sample

6. Create a sample custom resource (CR).
   a. Create a sample CR:

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

   b. Watch for the CR to reconcile the Operator:

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

7. Clean up.
Run the following command to clean up the resources that have been created as part of this procedure:
$ make undeploy

5.5.1.3. Next steps

- See Operator SDK tutorial for Helm-based Operators for a more in-depth walkthrough on building a Helm-based Operator.

5.5.2. Operator SDK tutorial for Helm-based Operators

Operator developers can take advantage of Helm support in the Operator SDK to build an example Helm-based Operator for Nginx and manage its lifecycle. This tutorial walks through the following process:

- Create a Nginx deployment
- Ensure that the deployment size is the same as specified by the Nginx custom resource (CR) spec
- Update the Nginx CR status using the status writer with the names of the nginx pods

This process is accomplished using two centerpieces of the Operator Framework:

**Operator SDK**

The `operator-sdk` CLI tool and `controller-runtime` library API

**Operator Lifecycle Manager (OLM)**

Installation, upgrade, and role-based access control (RBAC) of Operators on a cluster

**NOTE**

This tutorial goes into greater detail than Getting started with Operator SDK for Helm-based Operators.

5.5.2.1. Prerequisites

- **Operator SDK CLI installed**
- **OpenShift CLI (`oc`) v4.7+ installed**
- Logged into an OpenShift Container Platform 4.7 cluster with `oc` with an account that has `cluster-admin` permissions
- To allow the cluster pull the image, the repository where you push your image must be set as public, or you must configure an image pull secret.

5.5.2.2. Creating a project

Use the Operator SDK CLI to create a project called `nginx-operator`.

**Procedure**

1. Create a directory for the project:

   ```
   $ mkdir -p $HOME/projects/nginx-operator
   ```
2. Change to the directory:

```
$ cd $HOME/projects/nginx-operator
```

3. Run the `operator-sdk init` command with the `helm` plug-in to initialize the project:

```
$ operator-sdk init \n   --plugins=helm \n   --domain=example.com \n   --group=demo \n   --version=v1 \n   --kind=Nginx
```

**NOTE**

By default, the `helm` plug-in initializes a project using a boilerplate Helm chart. You can use additional flags, such as the `--helm-chart` flag, to initialize a project using an existing Helm chart.

The `init` command creates the `nginx-operator` project specifically for watching a resource with API version `example.com/v1` and kind `Nginx`.

4. For Helm-based projects, the `init` command generates the RBAC rules in the `config/rbac/role.yaml` file based on the resources that would be deployed by the default manifest for the chart. Verify that the rules generated in this file meet the permission requirements of the Operator.

### 5.5.2.2.1. Existing Helm charts

Instead of creating your project with a boilerplate Helm chart, you can alternatively use an existing chart, either from your local file system or a remote chart repository, by using the following flags:

- `--helm-chart`
- `--helm-chart-repo`
- `--helm-chart-version`

If the `--helm-chart` flag is specified, the `--group`, `--version`, and `--kind` flags become optional. If left unset, the following default values are used:

<table>
<thead>
<tr>
<th>Flag</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>--domain</td>
<td>my.domain</td>
</tr>
<tr>
<td>--group</td>
<td>charts</td>
</tr>
<tr>
<td>--version</td>
<td>v1</td>
</tr>
<tr>
<td>--kind</td>
<td>Deduced from the specified chart</td>
</tr>
</tbody>
</table>
If the `--helm-chart` flag specifies a local chart archive, for example `example-chart-1.2.0.tgz`, or directory, the chart is validated and unpacked or copied into the project. Otherwise, the Operator SDK attempts to fetch the chart from a remote repository.

If a custom repository URL is not specified by the `--helm-chart-repo` flag, the following chart reference formats are supported:

<table>
<thead>
<tr>
<th>Format</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>&lt;repo_name&gt;/&lt;chart_name&gt;</code></td>
<td>Fetch the Helm chart named <code>&lt;chart_name&gt;</code> from the helm chart repository named <code>&lt;repo_name&gt;</code>, as specified in the <code>$HELM_HOME/repositories/repositories.yaml</code> file. Use the <code>helm repo add</code> command to configure this file.</td>
</tr>
<tr>
<td><code>&lt;url&gt;</code></td>
<td>Fetch the Helm chart archive at the specified URL.</td>
</tr>
</tbody>
</table>

If a custom repository URL is specified by `--helm-chart-repo`, the following chart reference format is supported:

<table>
<thead>
<tr>
<th>Format</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>&lt;chart_name&gt;</code></td>
<td>Fetch the Helm chart named <code>&lt;chart_name&gt;</code> in the Helm chart repository specified by the <code>--helm-chart-repo</code> URL value.</td>
</tr>
</tbody>
</table>

If the `--helm-chart-version` flag is unset, the Operator SDK fetches the latest available version of the Helm chart. Otherwise, it fetches the specified version. The optional `--helm-chart-version` flag is not used when the chart specified with the `--helm-chart` flag refers to a specific version, for example when it is a local path or a URL.

For more details and examples, run:

```
$ operator-sdk init --plugins helm --help
```

### 5.5.2.2. PROJECT file

Among the files generated by the `operator-sdk init` command is a Kubebuilder PROJECT file. Subsequent `operator-sdk` commands, as well as `help` output, that are run from the project root read this file and are aware that the project type is Helm. For example:

```
domain: example.com
layout: helm.sdk.operatorframework.io/v1
projectName: helm-operator
resources:
  - group: demo
    kind: Nginx
    version: v1
    version: 3-alpha
```

### 5.5.2.3. Understanding the Operator logic
For this example, the **nginx-operator** project executes the following reconciliation logic for each **Nginx** custom resource (CR):

- Create an Nginx deployment if it does not exist.
- Create an Nginx service if it does not exist.
- Create an Nginx ingress if it is enabled and does not exist.
- Ensure that the deployment, service, and optional ingress match the desired configuration as specified by the **Nginx** CR, for example the replica count, image, and service type.

By default, the **nginx-operator** project watches **Nginx** resource events as shown in the **watches.yaml** file and executes Helm releases using the specified chart:

```yaml
# Use the 'create api' subcommand to add watches to this file.
- group: demo
  version: v1
  kind: Nginx
  chart: helm-charts/nginx
  # +kubebuilder:scaffold:watch
```

### 5.5.2.3.1. Sample Helm chart

When a Helm Operator project is created, the Operator SDK creates a sample Helm chart that contains a set of templates for a simple Nginx release.

For this example, templates are available for deployment, service, and ingress resources, along with a **NOTES.txt** template, which Helm chart developers use to convey helpful information about a release.

If you are not already familiar with Helm charts, review the [Helm developer documentation](#).

### 5.5.2.3.2. Modifying the custom resource spec

Helm uses a concept called **values** to provide customizations to the defaults of a Helm chart, which are defined in the **values.yaml** file.

You can override these defaults by setting the desired values in the custom resource (CR) spec. You can use the number of replicas as an example.

**Procedure**

1. The **helm-charts/nginx/values.yaml** file has a value called **replicaCount** set to 1 by default. To have two Nginx instances in your deployment, your CR spec must contain **replicaCount: 2**.
   
   Edit the **config/samples/demo_v1_nginx.yaml** file to set **replicaCount: 2**:

   ```yaml
   apiVersion: demo.example.com/v1
   kind: Nginx
   metadata:
     name: nginx-sample
   spec:
     replicaCount: 2
   ```
2. Similarly, the default service port is set to 80. To use 8080, edit the `config/samples/demo_v1_nginx.yaml` file to set `spec.port: 8080`, which adds the service port override:

```yaml
apiVersion: demo.example.com/v1
kind: Nginx
metadata:
  name: nginx-sample
spec:
  replicaCount: 2
  service:
    port: 8080
```

The Helm Operator applies the entire spec as if it was the contents of a values file, just like the `helm install -f ./overrides.yaml` command.

### 5.5.2.4. Running the Operator

There are three ways you can use the Operator SDK CLI to build and run your Operator:

- Run locally outside the cluster as a Go program.
- Run as a deployment on the cluster.
- Bundle your Operator and use Operator Lifecycle Manager (OLM) to deploy on the cluster.

#### 5.5.2.4.1. Running locally outside the cluster

You can run your Operator project as a Go program outside of the cluster. This is useful for development purposes to speed up deployment and testing.

**Procedure**

- Run the following command to install the custom resource definitions (CRDs) in the cluster configured in your `~/.kube/config` file and run the Operator locally:

  ```bash
  $ make install run
  ```

**Example output**

```json
...
{
  "level": "info",
  "ts": "1612652419.9289865",
  "logger": "controller-runtime.metrics",
  "msg": "metrics server is starting to listen", "addr": ":8080"
}
{
  "level": "info",
  "ts": "1612652419.9296563",
  "logger": "helm.controller",
  "msg": "Watching resource", "apiVersion": "demo.example.com/v1",
  "kind": "Nginx",
  "namespace": "",
  "reconcilePeriod": "1m0s"
}
{
  "level": "info",
  "ts": "1612652419.929983",
  "logger": "controller-runtime.manager",
  "msg": "starting metrics server", "path": "/metrics"
}
{
  "level": "info",
  "ts": "1612652419.930015",
  "logger": "controller-runtime.manager.controller.nginx-controller",
  "msg": "Starting EventSource", "source": "kind source: demo.example.com/v1, Kind=nginx"
}
{
  "level": "info",
  "ts": "1612652420.2307851",
  "logger": "controller-runtime.manager.controller.nginx-
```
5.5.2.4.2. Preparing your Operator to use supported images

Before running your Helm-based Operator on OpenShift Container Platform, update your project to use supported images.

Procedure

1. Update the project root-level Dockerfile to use supported images. Change the default builder image reference from:

```
FROM quay.io/operator-framework/helm-operator:v1.3.0
```

to:

```
FROM registry.redhat.io/openshift4/ose-helm-operator:v4.7
```

**IMPORTANT**

Use the builder image version that matches your Operator SDK version. Failure to do so can result in problems due to project layout, or scaffolding, differences, particularly when mixing newer upstream versions of the Operator SDK with downstream OpenShift Container Platform builder images.

2. In the `config/default/manager_auth_proxy_patch.yaml` file, change the `image` value from:

```
gcr.io/kubebuilder/kube-rbac-proxy:<tag>
```

to use the supported image:

```
registry.redhat.io/openshift4/ose-kube-rbac-proxy:v4.7
```

5.5.2.4.3. Running as a deployment on the cluster

You can run your Operator project as a deployment on your cluster.

Procedure

1. Run the following `make` commands to build and push the Operator image. Modify the `IMG` argument in the following steps to reference a repository that you have access to. You can obtain an account for storing containers at repository sites such as Quay.io.

   a. Build the image:

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

   b. Push the image to a repository:

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

The name and tag of the image, for example IMG=
<registry>/<user>/<image_name>:<tag>, in both the commands can also be set in your Makefile. Modify the IMG ?= controller:latest value to set your default image name.

2. Run the following command to deploy the Operator:

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

   By default, this command creates a namespace with the name of your Operator project in the form <project_name>-system and is used for the deployment. This command also installs the RBAC manifests from config/rbac.

3. Verify that the Operator is running:

   $ oc get deployment -n <project_name>-system

   Example output

<table>
<thead>
<tr>
<th>NAME</th>
<th>READY</th>
<th>UP-TO-DATE</th>
<th>AVAILABLE</th>
<th>AGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;project_name&gt;-controller-manager</td>
<td>1/1</td>
<td>1</td>
<td>1</td>
<td>8m</td>
</tr>
</tbody>
</table>

5.5.2.4.4. Bundling an Operator and deploying with Operator Lifecycle Manager

Operator Lifecycle Manager (OLM) helps you to install, update, and generally manage the lifecycle of Operators and their associated services on a Kubernetes cluster. OLM is installed by default on OpenShift Container Platform and runs as a Kubernetes extension so that you can use the web console and the OpenShift CLI (oc) for all Operator lifecycle management functions without any additional tools.

The Operator Bundle Format is the default packaging method for Operator SDK and OLM. You can get your Operator ready for OLM by using the Operator SDK to build, push, validate, and run a bundle image with OLM.

Prerequisites

- Operator SDK CLI installed on a development workstation
- OpenShift CLI (oc) v4.7+ installed
- Operator Lifecycle Manager (OLM) installed on a Kubernetes-based cluster (v1.16.0 or later if you use apiextensions.k8s.io/v1 CRDs, for example OpenShift Container Platform 4.7)
- Logged into the cluster with oc using an account with cluster-admin permissions
- Operator project initialized by using the Operator SDK

Procedure

1. Run the following make commands in your Operator project directory to build and push your Operator image. Modify the IMG argument in the following steps to reference a repository that you have access to. You can obtain an account for storing containers at repository sites such as
Quay.io.

a. Build the image:

$ make docker-build IMG=<registry>/<user>/<operator_image_name>:<tag>

b. Push the image to a repository:

$ make docker-push IMG=<registry>/<user>/<operator_image_name>:<tag>

2. Update your Makefile by setting the IMG URL to your Operator image name and tag that you pushed:

$ # Image URL to use all building/pushing image targets
IMG ?= <registry>/<user>/<operator_image_name>:<tag>

This value is used for subsequent operations.

3. Create your Operator bundle manifest by running the make bundle command, which invokes several commands, including the Operator SDK generate bundle and bundle validate subcommands:

$ make bundle

Bundle manifests for an Operator describe how to display, create, and manage an application. The make bundle command creates the following files and directories in your Operator project:

- A bundle manifests directory named bundle/manifests that contains a ClusterServiceVersion object
- A bundle metadata directory named bundle/metadata
- All custom resource definitions (CRDs) in a config/crd directory
- A Dockerfile bundle.Dockerfile

These files are then automatically validated by using operator-sdk bundle validate to ensure the on-disk bundle representation is correct.

4. Build and push your bundle image by running the following commands. OLM consumes Operator bundles using an index image, which reference one or more bundle images.

a. Build the bundle image. Set BUNDLE_IMAGE with the details for the registry, user namespace, and image tag where you intend to push the image:

$ make bundle-build BUNDLE_IMG=<registry>/<user>/<bundle_image_name>:<tag>

b. Push the bundle image:

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

5. Check the status of OLM on your cluster by using the following Operator SDK command:

$ operator-sdk olm status \
   --olm-namespace=openshift-operator-lifecycle-manager
6. Run the Operator on your cluster by using the OLM integration in Operator SDK:

```bash
$ operator-sdk run bundle \n   [-n <namespace>] \n   <registry>/<user>/<bundle_image_name>:<tag>
```

By default, the command installs the Operator in the currently active project in your
`~/.kube/config` file. You can add the `-n` flag to set a different namespace scope for the
installation.

This command performs the following actions:

- Create an index image with your bundle image injected.
- Create a catalog source that points to your new index image, which enables OperatorHub to
discover your Operator.
- Deploy your Operator to your cluster by creating an Operator group, subscription, install
plan, and all other required objects, including RBAC.

### 5.5.2.5. Creating a custom resource

After your Operator is installed, you can test it by creating a custom resource (CR) that is now provided
on the cluster by the Operator.

**Prerequisites**

- Example Nginx Operator, which provides the **Nginx** CR, installed on a cluster

**Procedure**

1. Change to the namespace where your Operator is installed. For example, if you deployed the
Operator using the `make deploy` command:

```bash
$ oc project nginx-operator-system
```

2. Edit the sample **Nginx** CR manifest at `config/samples/demo_v1_nginx.yaml` to contain the
following specification:

```yaml
apiVersion: demo.example.com/v1
kind: Nginx
metadata:
  name: nginx-sample
...
spec:
  ...
  replicaCount: 3
```

3. The Nginx service account requires privileged access to run in OpenShift Container Platform.
Add the following security context constraint (SCC) to the service account for the **nginx-
sample** pod:
4. Create the CR:

```
$ oc adm policy add-scc-to-user \
 anyuid system:serviceaccount:nginx-operator-system:nginx-sample
```

5. Ensure that the **Nginx** Operator creates the deployment for the sample CR with the correct size:

```
$ oc get deployments
```

**Example output**

<table>
<thead>
<tr>
<th>NAME</th>
<th>READY</th>
<th>UP-TO-DATE</th>
<th>AVAILABLE</th>
<th>AGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>nginx-operator-controller-manager</td>
<td>1/1</td>
<td>1</td>
<td>1</td>
<td>8m</td>
</tr>
<tr>
<td>nginx-sample</td>
<td>3/3</td>
<td>3</td>
<td>3</td>
<td>1m</td>
</tr>
</tbody>
</table>

6. Check the pods and CR status to confirm the status is updated with the Nginx pod names.

   a. Check the pods:

   ```
   $ oc get pods
   ```

   **Example output**

<table>
<thead>
<tr>
<th>NAME</th>
<th>READY</th>
<th>STATUS</th>
<th>RESTARTS</th>
<th>AGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>nginx-sample-6fd7c98d8-7dqdr</td>
<td>1/1</td>
<td>Running</td>
<td>0</td>
<td>1m</td>
</tr>
<tr>
<td>nginx-sample-6fd7c98d8-g5k7v</td>
<td>1/1</td>
<td>Running</td>
<td>0</td>
<td>1m</td>
</tr>
<tr>
<td>nginx-sample-6fd7c98d8-m7vn7</td>
<td>1/1</td>
<td>Running</td>
<td>0</td>
<td>1m</td>
</tr>
</tbody>
</table>

   b. Check the CR status:

   ```
   $ oc get nginx/nginx-sample -o yaml
   ```

   **Example output**

   ```yaml
   apiVersion: demo.example.com/v1
   kind: Nginx
   metadata:
   ...
   - name: nginx-sample
   ...
   spec:
   replicaCount: 3
   status:
   nodes:
   - nginx-sample-6fd7c98d8-7dqdr
   - nginx-sample-6fd7c98d8-g5k7v
   - nginx-sample-6fd7c98d8-m7vn7
   ```

7. Update the deployment size.
a. Update `config/samples/demo_v1_nginx.yaml` file to change the `spec.size` field in the Nginx CR from 3 to 5:

```
$ oc patch nginx nginx-sample -p \'"spec":{"replicaCount": 5}\' --type=merge
```

b. Confirm that the Operator changes the deployment size:

```
$ oc get deployments
```

Example output

<table>
<thead>
<tr>
<th>NAME</th>
<th>READY</th>
<th>UP-TO-DATE</th>
<th>AVAILABLE</th>
<th>AGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>nginx-operator-controller-manager</td>
<td>1/1</td>
<td>1</td>
<td>1</td>
<td>10m</td>
</tr>
<tr>
<td>nginx-sample</td>
<td>5/5</td>
<td>5</td>
<td>5</td>
<td>3m</td>
</tr>
</tbody>
</table>

8. Clean up the resources that have been created as part of this tutorial.

- If you used the `make deploy` command to test the Operator, run the following command:

```
$ make undeploy
```

- If you used the `operator-sdk run bundle` command to test the Operator, run the following command:

```
$ operator-sdk cleanup <project_name>
```

5.5.2.6. Additional resources

- See Project layout for Helm-based Operators to learn about the directory structures created by the Operator SDK.

5.5.3. Project layout for Helm-based Operators

The `operator-sdk` CLI can generate, or `scaffold`, a number of packages and files for each Operator project.

5.5.3.1. Helm-based project layout

Helm-based Operator projects generated using the `operator-sdk init --plugins helm` command contain the following directories and files:

<table>
<thead>
<tr>
<th>File/folders</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>config</td>
<td>Kustomize manifests for deploying the Operator on a Kubernetes cluster.</td>
</tr>
<tr>
<td>helm-charts/</td>
<td>Helm chart initialized with the <code>operator-sdk create api</code> command.</td>
</tr>
<tr>
<td>Dockerfile</td>
<td>Used to build the Operator image with the <code>make docker-build</code> command.</td>
</tr>
</tbody>
</table>
### File/folders

<table>
<thead>
<tr>
<th>File/folders</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>watches.yaml</code></td>
<td>Group/version/kind (GVK) and Helm chart location.</td>
</tr>
<tr>
<td>Makefile</td>
<td>Targets used to manage the project.</td>
</tr>
<tr>
<td>PROJECT</td>
<td>YAML file containing metadata information for the Operator.</td>
</tr>
</tbody>
</table>

### 5.5.4. Helm support in Operator SDK

#### 5.5.4.1. Helm charts

One of the Operator SDK options for generating an Operator project includes leveraging an existing Helm chart to deploy Kubernetes resources as a unified application, without having to write any Go code. Such Helm-based Operators are designed to excel at stateless applications that require very little logic when rolled out, because changes should be applied to the Kubernetes objects that are generated as part of the chart. This may sound limiting, but can be sufficient for a surprising amount of use-cases as shown by the proliferation of Helm charts built by the Kubernetes community.

The main function of an Operator is to read from a custom object that represents your application instance and have its desired state match what is running. In the case of a Helm-based Operator, the `spec` field of the object is a list of configuration options that are typically described in the Helm `values.yaml` file. Instead of setting these values with flags using the Helm CLI (for example, `helm install -f values.yaml`), you can express them within a custom resource (CR), which, as a native Kubernetes object, enables the benefits of RBAC applied to it and an audit trail.

For an example of a simple CR called **Tomcat**:

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

The `replicaCount` value, 2 in this case, is propagated into the template of the chart where the following is used:

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

After an Operator is built and deployed, you can deploy a new instance of an app by creating a new instance of a CR, or list the different instances running in all environments using the `oc` command:

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

There is no requirement use the Helm CLI or install Tiller; Helm-based Operators import code from the Helm project. All you have to do is have an instance of the Operator running and register the CR with a custom resource definition (CRD). Because it obeys RBAC, you can more easily prevent production changes.

### 5.6. DEFINING CLUSTER SERVICE VERSIONS (CSVs)
A *cluster service version* (CSV), defined by a `ClusterServiceVersion` object, is a YAML manifest created from Operator metadata that assists Operator Lifecycle Manager (OLM) in running the Operator in a cluster. It is the metadata that accompanies an Operator container image, used to populate user interfaces with information such as its logo, description, and version. It is also a source of technical information that is required to run the Operator, like the RBAC rules it requires and which custom resources (CRs) it manages or depends on.

The Operator SDK includes the CSV generator to generate a CSV for the current Operator project, customized using information contained in YAML manifests and Operator source files.

A CSV-generating command removes the responsibility of Operator authors having in-depth OLM knowledge in order for their Operator to interact with OLM or publish metadata to the Catalog Registry. Further, because the CSV spec will likely change over time as new Kubernetes and OLM features are implemented, the Operator SDK is equipped to easily extend its update system to handle new CSV features going forward.

5.6.1. How CSV generation works

Operator bundle manifests, which include cluster service versions (CSVs), describe how to display, create, and manage an application with Operator Lifecycle Manager (OLM). The CSV generator in the Operator SDK, called by the `generate bundle` subcommand, is the first step towards publishing your Operator to a catalog and deploying it with OLM. The subcommand requires certain input manifests to construct a CSV manifest; all inputs are read when the command is invoked, along with a CSV base, to idempotently generate or regenerate a CSV.

Typically, the `generate kustomize manifests` subcommand would be run first to generate the input Kustomize bases that are consumed by the `generate bundle` subcommand. However, the Operator SDK provides the `make bundle` command, which automates several tasks, including running the following subcommands in order:

1. `generate kustomize manifests`
2. `generate bundle`
3. `bundle validate`

Additional resources

- See [Bundling an Operator and deploying with Operator Lifecycle Manager](#) for a full procedure that includes generating a bundle and CSV.

5.6.1.1. Generated files and resources

The `make bundle` command creates the following files and directories in your Operator project:

- A bundle manifests directory named `bundle/manifests` that contains a `ClusterServiceVersion` (CSV) object
- A bundle metadata directory named `bundle/metadata`
- All custom resource definitions (CRDs) in a `config/crd` directory
- A Dockerfile `bundle.Dockerfile`

The following resources are typically included in a CSV:
Role
Defines Operator permissions within a namespace.

ClusterRole
Defines cluster-wide Operator permissions.

Deployment
Defines how an Operand of an Operator is run in pods.

CustomResourceDefinition (CRD)
Defines custom resources that your Operator reconciles.

Custom resource examples
Examples of resources adhering to the spec of a particular CRD.

5.6.1.2. Version management
The --version flag for the generate bundle subcommand supplies a semantic version for your bundle when creating one for the first time and when upgrading an existing one.

By setting the VERSION variable in your Makefile, the --version flag is automatically invoked using that value when the generate bundle subcommand is run by the make bundle command. The CSV version is the same as the Operator version, and a new CSV is generated when upgrading Operator versions.

5.6.2. Manually-defined CSV fields
Many CSV fields cannot be populated using generated, generic manifests that are not specific to Operator SDK. These fields are mostly human-written metadata about the Operator and various custom resource definitions (CRDs).

Operator authors must directly modify their cluster service version (CSV) YAML file, adding personalized data to the following required fields. The Operator SDK gives a warning during CSV generation when a lack of data in any of the required fields is detected.

The following tables detail which manually-defined CSV fields are required and which are optional.

Table 5.7. Required

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>metadata.name</td>
<td>A unique name for this CSV. Operator version should be included in the name to ensure uniqueness, for example app-operator.v0.1.1.</td>
</tr>
<tr>
<td>metadata.capabilities</td>
<td>The capability level according to the Operator maturity model. Options include Basic Install, Seamless Upgrades, Full Lifecycle, Deep Insights, and Auto Pilot.</td>
</tr>
<tr>
<td>spec.displayName</td>
<td>A public name to identify the Operator.</td>
</tr>
<tr>
<td>spec.description</td>
<td>A short description of the functionality of the Operator.</td>
</tr>
<tr>
<td>spec.keywords</td>
<td>Keywords describing the Operator.</td>
</tr>
</tbody>
</table>
### Field | Description
--- | ---
`spec.maintainers` | Human or organizational entities maintaining the Operator, with a **name** and **email**.

`spec.provider` | The provider of the Operator (usually an organization), with a **name**.

`spec.labels` | Key-value pairs to be used by Operator internals.

`spec.version` | Semantic version of the Operator, for example **0.1.1**.

`spec.customresourcedefinitions` | Any CRDs the Operator uses. This field is populated automatically by the Operator SDK if any CRD YAML files are present in `deploy/`. However, several fields not in the CRD manifest spec require user input:
- **description**: description of the CRD.
- **resources**: any Kubernetes resources leveraged by the CRD, for example **Pod** and **StatefulSet** objects.
- **specDescriptors**: UI hints for inputs and outputs of the Operator.

### Table 5.8. Optional

#### Field | Description
--- | ---
`spec.replaces` | The name of the CSV being replaced by this CSV.

`spec.links` | URLs (for example, websites and documentation) pertaining to the Operator or application being managed, each with a **name** and **url**.

`spec.selector` | Selectors by which the Operator can pair resources in a cluster.

`spec.icon` | A base64-encoded icon unique to the Operator, set in a **base64data** field with a **mediatype**.

`spec.maturity` | The level of maturity the software has achieved at this version. Options include **planning**, **pre-alpha**, **alpha**, **beta**, **stable**, **mature**, **inactive**, and **deprecated**.

Further details on what data each field above should hold are found in the CSV spec.

**NOTE**

Several YAML fields currently requiring user intervention can potentially be parsed from Operator code.

Additional resources
5.6.2.1. Operator metadata annotations

Operator developers can manually define certain annotations in the metadata of a cluster service version (CSV) to enable features or highlight capabilities in user interfaces (UIs), such as OperatorHub.

The following table lists Operator metadata annotations that can be manually defined using `metadata.annotations` fields.

Table 5.9. Annotations

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>alm-examples</code></td>
<td>Provide custom resource definition (CRD) templates with a minimum set of configuration. Compatible UIs pre-fill this template for users to further customize.</td>
</tr>
<tr>
<td><code>operatorframework.io/initialization-resource</code></td>
<td>Specify a single required custom resource that must be created at the time that the Operator is installed. Must include a template that contains a complete YAML definition.</td>
</tr>
<tr>
<td><code>operatorframework.io/suggested-namespace</code></td>
<td>Set a suggested namespace where the Operator should be deployed.</td>
</tr>
<tr>
<td>Field</td>
<td>Description</td>
</tr>
<tr>
<td>-------</td>
<td>-------------</td>
</tr>
</tbody>
</table>
| `operators.openshift.io/infrastructure-features` | Infrastructure features supported by the Operator. Users can view and filter by these features when discovering Operators through OperatorHub in the web console. Valid, case-sensitive values:  
  - **disconnected**: Operator supports being mirrored into disconnected catalogs, including all dependencies, and does not require Internet access. All related images required for mirroring are listed by the Operator.  
  - **cnf**: Operator provides a Cloud-native Network Functions (CNF) Kubernetes plug-in.  
  - **cni**: Operator provides a Container Network Interface (CNI) Kubernetes plug-in.  
  - **csi**: Operator provides a Container Storage Interface (CSI) Kubernetes plug-in.  
  - **fips**: Operator accepts the FIPS mode of the underlying platform and works on nodes that are booted into FIPS mode.  
  
  **IMPORTANT**  
  The use of FIPS Validated / Modules in Process cryptographic libraries is only supported on OpenShift Container Platform deployments on the `x86_64` architecture.  
  - **proxy-aware**: Operator supports running on a cluster behind a proxy. Operator accepts the standard proxy environment variables `HTTP_PROXY` and `HTTPS_PROXY`, which Operator Lifecycle Manager (OLM) provides to the Operator automatically when the cluster is configured to use a proxy. Required environment variables are passed down to Operands for managed workloads. |
<p>| <code>operators.openshift.io/valid-subscription</code> | Free-form array for listing any specific subscriptions that are required to use the Operator. For example, <code>&quot;[&quot;3Scale Commercial License&quot;, &quot;Red Hat Managed Integration&quot;]&quot;</code>. |</p>
<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>operators.operatorframework.io/internal-objects</td>
<td>Hides CRDs in the UI that are not meant for user manipulation.</td>
</tr>
</tbody>
</table>

**Example use cases**

**Operator supports disconnected and proxy-aware**

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

**Operator requires an OpenShift Container Platform license**

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

**Operator requires a 3scale license**

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

**Operator supports disconnected and proxy-aware, and requires an OpenShift Container Platform license**

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

**Additional resources**

- CRD templates
- Initializing required custom resources
- Setting a suggested namespace
- Enabling your Operator for restricted network environments (disconnected mode)
- Hiding internal objects
- Support for FIPS crytography

**5.6.3. Enabling your Operator for restricted network environments**

As an Operator author, your Operator must meet additional requirements to run properly in a restricted network, or disconnected, environment.

**Operator requirements for supporting disconnected mode**

- In the cluster service version (CSV) of your Operator:
  - List any related images, or other container images that your Operator might require to perform their functions.
Reference all specified images by a digest (SHA) and not by a tag.

- All dependencies of your Operator must also support running in a disconnected mode.
- Your Operator must not require any off-cluster resources.

For the CSV requirements, you can make the following changes as the Operator author.

Prerequisites

- An Operator project with a CSV.

Procedure

1. Use SHA references to related images in two places in the CSV for your Operator:

   a. Update `spec.relatedImages`:

   ```yaml
   spec:
     relatedImages:
     - name: etcd-operator
       image: quay.io/etcd-operator/operator@sha256:d134a9865524c29f75bbc4469013bc38d8a15cb5f41acfd6db6b9e492f556e4
     - name: etcd-image
       image: quay.io/etcd-operator/etcd@sha256:13348c15263bd8838ec1d5fc4550ede9860fcb0f843e48cbceec07810e6bb68
   ...
   ```

   1. Create a `relatedImages` section and set the list of related images.
   2. Specify a unique identifier for the image.
   3. Specify each image by a digest (SHA), not by an image tag.

   b. Update the `env` section in the deployment when declaring environment variables that inject the image that the Operator should use:

   ```yaml
   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@sha256:13348c15263bd8838ec1d5fc4550ede9860fcb0f843e48cbcc07810e5b68
    - name: ETCD_LOG_LEVEL
      value: INFO
    image: quay.io/etcd-operator/operator@sha256:d134a9865524c29f575bb4469013bc38d8a15cb5f41acfdb629e492f55e4
    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. Inject the images referenced by the Operator by using environment variables.
2. Specify each image by a digest (SHA), not by an image tag.
3. Also reference the Operator container image by a digest (SHA), not by an image tag.

NOTE

When configuring probes, the `timeoutSeconds` value must be lower than the `periodSeconds` value. The `timeoutSeconds` default value is 1. The `periodSeconds` default value is 10.

2. Add the `disconnected` annotation, which indicates that the Operator works in a disconnected environment:
Operators can be filtered in OperatorHub by this infrastructure feature.

### 5.6.4. Enabling your Operator for multiple architectures and operating systems

Operator Lifecycle Manager (OLM) assumes that all Operators run on Linux hosts. However, as an Operator author, you can specify whether your Operator supports managing workloads on other architectures, if worker nodes are available in the OpenShift Container Platform cluster.

If your Operator supports variants other than AMD64 and Linux, you can add labels to the cluster service version (CSV) that provides the Operator to list the supported variants. Labels indicating supported architectures and operating systems are defined by the following:

```plaintext
Set <arch> to a supported string.
Set <os> to a supported string.
```

**NOTE**

Only the labels on the channel head of the default channel are considered for filtering package manifests by label. This means, for example, that providing an additional architecture for an Operator in the non-default channel is possible, but that architecture is not available for filtering in the PackageManifest API.

If a CSV does not include an `os` label, it is treated as if it has the following Linux support label by default:

```plaintext
operatorframework.io/os.linux: supported
```

If a CSV does not include an `arch` label, it is treated as if it has the following AMD64 support label by default:

```plaintext
operatorframework.io/arch.amd64: supported
```

If an Operator supports multiple node architectures or operating systems, you can add multiple labels, as well.

**Prerequisites**

- An Operator project with a CSV.
- To support listing multiple architectures and operating systems, your Operator image referenced in the CSV must be a manifest list image.
For the Operator to work properly in restricted network, or disconnected, environments, the image referenced must also be specified using a digest (SHA) and not by a tag.

**Procedure**

- Add a label in the `metadata.labels` of your CSV for each supported architecture and operating system that your Operator supports:

```bash
labels:
  operatorframework.io/arch.s390x: supported
  operatorframework.io/os.zos: supported
  operatorframework.io/os.linux: supported
  operatorframework.io/arch.amd64: supported
```

After you add a new architecture or operating system, you must also now include the default `os.linux` and `arch.amd64` variants explicitly.

**Additional resources**

- See the [Image Manifest V 2, Schema 2](#) specification for more information on manifest lists.

### 5.6.4.1. Architecture and operating system support for Operators

The following strings are supported in Operator Lifecycle Manager (OLM) on OpenShift Container Platform when labeling or filtering Operators that support multiple architectures and operating systems:

<table>
<thead>
<tr>
<th>Architecture</th>
<th>String</th>
</tr>
</thead>
<tbody>
<tr>
<td>AMD64</td>
<td>amd64</td>
</tr>
<tr>
<td>64-bit PowerPC little-endian</td>
<td>ppc64le</td>
</tr>
<tr>
<td>IBM Z</td>
<td>s390x</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Operating system</th>
<th>String</th>
</tr>
</thead>
<tbody>
<tr>
<td>Linux</td>
<td>linux</td>
</tr>
<tr>
<td>z/OS</td>
<td>zos</td>
</tr>
</tbody>
</table>

**NOTE**

Different versions of OpenShift Container Platform and other Kubernetes-based distributions might support a different set of architectures and operating systems.
5.6.5. Setting a suggested namespace

Some Operators must be deployed in a specific namespace, or with ancillary resources in specific namespaces, to work properly. If resolved from a subscription, Operator Lifecycle Manager (OLM) defaults the namespaced resources of an Operator to the namespace of its subscription.

As an Operator author, you can instead express a desired target namespace as part of your cluster service version (CSV) to maintain control over the final namespaces of the resources installed for their Operators. When adding the Operator to a cluster using OperatorHub, this enables the web console to autopopulate the suggested namespace for the cluster administrator during the installation process.

Procedure

- In your CSV, set the `operatorframework.io/suggested-namespace` annotation to your suggested namespace:

  ```yaml
  metadata:
    annotations:
      operatorframework.io/suggested-namespace: <namespace>
  ```

  (1) Set your suggested namespace.

5.6.6. Enabling Operator conditions

Operator Lifecycle Manager (OLM) provides Operators with a channel to communicate complex states that influence OLM behavior while managing the Operator. By default, OLM creates an `OperatorCondition` custom resource definition (CRD) when it installs an Operator. Based on the conditions set in the `OperatorCondition` custom resource (CR), the behavior of OLM changes accordingly.

To support Operator conditions, an Operator must be able to read the `OperatorCondition` CR created by OLM and have the ability to:

- Get the specific condition.
- Set the status of a specific condition.

This can be accomplished by using the `operator-lib` library. An Operator author can provide a `controller-runtime` client in their Operator for the library to access the `OperatorCondition` CR owned by the Operator in the cluster.

The library provides a generic `Conditions` interface, which has the following methods to `Get` and `Set` a `conditionType` in the `OperatorCondition` CR:

**Get**

To get the specific condition, the library uses the `client.Get` function from `controller-runtime`, which requires an `ObjectKey` of type `types.NamespacedName` present in `conditionAccessor`.

**Set**

To update the status of the specific condition, the library uses the `client.Update` function from `controller-runtime`. An error occurs if the `conditionType` is not present in the CRD.

The Operator is allowed to modify only the `status` subresource of the CR. Operators can either delete or update the `status.conditions` array to include the condition. For more details on the format and description of the fields present in the conditions, see the upstream `Condition GoDocs`. 
NOTE

Operator SDK v1.3.0 supports **operator-lib** v0.3.0.

**Prerequisites**

- An Operator project generated using the Operator SDK.

**Procedure**

To enable Operator conditions in your Operator project:

1. In the `go.mod` file of your Operator project, add `operator-framework/operator-lib` as a required library:

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

2. Write your own constructor in your Operator logic that:
   - Accepts a `controller-runtime` client.
   - Accepts a `conditionType`.
   - Returns a `Condition` interface to update or add conditions.

Because OLM currently supports the **Upgradeable** condition, you can create an interface that has methods to access the **Upgradeable** condition. For example:

```go
import (  
  ...  
  apiv1 "github.com/operator-framework/api/pkg/operators/v1"  
)

func NewUpgradeable(cl client.Client) (Condition, error) {  
  return NewCondition(cl, "apiv1.OperatorUpgradeable")
}

cond, err := NewUpgradeable(cl);
```

In this example, the `NewUpgradeable` constructor is further used to create a variable `cond` of type `Condition`. The `cond` variable would in turn have `Get` and `Set` methods, which can be used for handling the OLM **Upgradeable** condition.

**Additional resources**

- Operator conditions
5.6.7. Defining webhooks

Webhooks allow Operator authors to intercept, modify, and accept or reject resources before they are saved to the object store and handled by the Operator controller. Operator Lifecycle Manager (OLM) can manage the lifecycle of these webhooks when they are shipped alongside your Operator.

The cluster service version (CSV) resource of an Operator can include a `webhookdefinitions` section to define the following types of webhooks:

- Admission webhooks (validating and mutating)
- Conversion webhooks

Procedure

- Add a `webhookdefinitions` section to the `spec` section of the CSV of your Operator and include any webhook definitions using a `type` of `ValidatingAdmissionWebhook`, `MutatingAdmissionWebhook`, or `ConversionWebhook`. The following example contains all three types of webhooks:

```yaml
apiVersion: operators.coreos.com/v1alpha1
kind: ClusterServiceVersion
metadata:
  name: webhook-operator.v0.1
spec:
  customresourcedefinitions:
  - kind: WebhookTest
    name: webhooktests.webhook.operators.coreos.io
    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
        admissionReviewVersions:
        - v1beta1
        - v1
        containerPort: 443
        targetPort: 4343
```

CHAPTER 5. DEVELOPING OPERATORS
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
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
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

...
4. A conversion webhook.

5. The `spec.PreserveUnknownFields` property of each CRD must be set to `false` or `nil`.

Additional resources

- Types of webhook admission plug-ins
- Kubernetes documentation:
  - Validating admission webhooks
  - Mutating admission webhooks
  - Conversion webhooks

5.6.7.1. Webhook considerations for OLM

When deploying an Operator with webhooks using Operator Lifecycle Manager (OLM), you must define the following:

- The `type` field must be set to either `ValidatingAdmissionWebhook`, `MutatingAdmissionWebhook`, or `ConversionWebhook`, or the CSV will be placed in a failed phase.

- The CSV must contain a deployment whose name is equivalent to the value supplied in the `deploymentName` field of the `webhookdefinition`.

When the webhook is created, OLM ensures that the webhook only acts upon namespaces that match the Operator group that the Operator is deployed in.

Certificate authority constraints
OLM is configured to provide each deployment with a single certificate authority (CA). The logic that generates and mounts the CA into the deployment was originally used by the API service lifecycle logic. As a result:

- The TLS certificate file is mounted to the deployment at `/apiserver.local.config/certificates/apiserver.crt`.
- The TLS key file is mounted to the deployment at `/apiserver.local.config/certificates/apiserver.key`.

Admission webhook rules constraints
To prevent an Operator from configuring the cluster into an unrecoverable state, OLM places the CSV in the failed phase if the rules defined in an admission webhook intercept any of the following requests:

- Requests that target all groups
- Requests that target the `operators.coreos.com` group
- Requests that target the `ValidatingWebhookConfigurations` or `MutatingWebhookConfigurations` resources

Conversion webhook constraints
OLM places the CSV in the failed phase if a conversion webhook definition does not adhere to the following constraints:
CSVs featuring a conversion webhook can only support the AllNamespaces install mode.

- The CRD targeted by the conversion webhook must have its spec.preserveUnknownFields field set to false or nil.
- The conversion webhook defined in the CSV must target an owned CRD.
- There can only be one conversion webhook on the entire cluster for a given CRD.

### 5.6.8. Understanding your custom resource definitions (CRDs)

There are two types of custom resource definitions (CRDs) that your Operator can use: ones that are owned by it and ones that it depends on, which are required.

#### 5.6.8.1. Owned CRDs

The custom resource definitions (CRDs) owned by your Operator are the most important part of your CSV. This establishes the link between your Operator and the required RBAC rules, dependency management, and other Kubernetes concepts.

It is common for your Operator to use multiple CRDs to link together concepts, such as top-level database configuration in one object and a representation of replica sets in another. Each one should be listed out in the CSV file.

### Table 5.12. Owned CRD fields

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
<th>Required/optional</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name</td>
<td>The full name of your CRD.</td>
<td>Required</td>
</tr>
<tr>
<td>Version</td>
<td>The version of that object API.</td>
<td>Required</td>
</tr>
<tr>
<td>Kind</td>
<td>The machine readable name of your CRD.</td>
<td>Required</td>
</tr>
<tr>
<td>DisplayName</td>
<td>A human readable version of your CRD name, for example MongoDB Standalone.</td>
<td>Required</td>
</tr>
<tr>
<td>Description</td>
<td>A short description of how this CRD is used by the Operator or a description of the functionality provided by the CRD.</td>
<td>Required</td>
</tr>
<tr>
<td>Group</td>
<td>The API group that this CRD belongs to, for example database.example.com.</td>
<td>Optional</td>
</tr>
</tbody>
</table>
### Resources
Your CRDs own one or more types of Kubernetes objects. These are listed in the **resources** section to inform your users of the objects they might need to troubleshoot or how to connect to the application, such as the service or ingress rule that exposes a database.

It is recommended to only list out the objects that are important to a human, not an exhaustive list of everything you orchestrate. For example, do not list config maps that store internal state that are not meant to be modified by a user.

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
<th>Required/optional</th>
</tr>
</thead>
<tbody>
<tr>
<td>Resources</td>
<td>Your CRDs own one or more types of Kubernetes objects. These are listed in the <strong>resources</strong> section to inform your users of the objects they might need to troubleshoot or how to connect to the application, such as the service or ingress rule that exposes a database. It is recommended to only list out the objects that are important to a human, not an exhaustive list of everything you orchestrate. For example, do not list config maps that store internal state that are not meant to be modified by a user.</td>
<td>Optional</td>
</tr>
</tbody>
</table>

### SpecDescriptors, StatusDescriptors, and ActionDescriptors
These descriptors are a way to hint UIs with certain inputs or outputs of your Operator that are most important to an end user. If your CRD contains the name of a secret or config map that the user must provide, you can specify that here. These items are linked and highlighted in compatible UIs.

There are three types of descriptors:

- **SpecDescriptors**: A reference to fields in the `spec` block of an object.
- **StatusDescriptors**: A reference to fields in the `status` block of an object.
- **ActionDescriptors**: A reference to actions that can be performed on an object.

All descriptors accept the following fields:

- **DisplayName**: A human readable name for the Spec, Status, or Action.
- **Description**: A short description of the Spec, Status, or Action and how it is used by the Operator.
- **Path**: A dot-delimited path of the field on the object that this descriptor describes.
- **X-Descriptors**: Used to determine which "capabilities" this descriptor has and which UI component to use. See the openshift/console project for a canonical list of React UI X-Descriptors for OpenShift Container Platform.

Also see the openshift/console project for more information on Descriptors in general.

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
<th>Required/optional</th>
</tr>
</thead>
</table>
| SpecDescriptors, StatusDescriptors, and ActionDescriptors | These descriptors are a way to hint UIs with certain inputs or outputs of your Operator that are most important to an end user. If your CRD contains the name of a secret or config map that the user must provide, you can specify that here. These items are linked and highlighted in compatible UIs. There are three types of descriptors:  
- **SpecDescriptors**: A reference to fields in the `spec` block of an object.  
- **StatusDescriptors**: A reference to fields in the `status` block of an object.  
- **ActionDescriptors**: A reference to actions that can be performed on an object.  
All descriptors accept the following fields:  
- **DisplayName**: A human readable name for the Spec, Status, or Action.  
- **Description**: A short description of the Spec, Status, or Action and how it is used by the Operator.  
- **Path**: A dot-delimited path of the field on the object that this descriptor describes.  
- **X-Descriptors**: Used to determine which "capabilities" this descriptor has and which UI component to use. See the openshift/console project for a canonical list of React UI X-Descriptors for OpenShift Container Platform. | Optional |

The following example depicts a **MongoDB Standalone** CRD that requires some user input in the form of a secret and config map, and orchestrates services, stateful sets, pods and config maps:

**Example owned CRD**

- displayName: MongoDB Standalone
5.6.8.2. Required CRDs

Relying on other required CRDs is completely optional and only exists to reduce the scope of individual Operators and provide a way to compose multiple Operators together to solve an end-to-end use case.

An example of this is an Operator that might set up an application and install an etcd cluster (from an etcd Operator) to use for distributed locking and a Postgres database (from a Postgres Operator) for data storage.

Operator Lifecycle Manager (OLM) checks against the available CRDs and Operators in the cluster to fulfill these requirements. If suitable versions are found, the Operators are started within the desired namespace and a service account created for each Operator to create, watch, and modify the Kubernetes resources required.
Table 5.13. Required CRD fields

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
<th>Required/optional</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name</td>
<td>The full name of the CRD you require.</td>
<td>Required</td>
</tr>
<tr>
<td>Version</td>
<td>The version of that object API.</td>
<td>Required</td>
</tr>
<tr>
<td>Kind</td>
<td>The Kubernetes object kind.</td>
<td>Required</td>
</tr>
<tr>
<td>DisplayName</td>
<td>A human readable version of the CRD.</td>
<td>Required</td>
</tr>
<tr>
<td>Description</td>
<td>A summary of how the component fits in your larger architecture.</td>
<td>Required</td>
</tr>
</tbody>
</table>

Example required CRD

```yaml
required:
- name: etcdclusters.etcd.database.coreos.com
  version: v1beta2
  kind: EtcdCluster
  displayName: etcd Cluster
  description: Represents a cluster of etcd nodes.
```

5.6.8.3. CRD upgrades

OLM upgrades a custom resource definition (CRD) immediately if it is owned by a singular cluster service version (CSV). If a CRD is owned by multiple CSVs, then the CRD is upgraded when it has satisfied all of the following backward compatible conditions:

- All existing serving versions in the current CRD are present in the new CRD.
- All existing instances, or custom resources, that are associated with the serving versions of the CRD are valid when validated against the validation schema of the new CRD.

5.6.8.3.1. Adding a new CRD version

Procedure

To add a new version of a CRD to your Operator:

1. Add a new entry in the CRD resource under the `versions` section of your CSV.
   For example, if the current CRD has a version `v1alpha1` and you want to add a new version `v1beta1` and mark it as the new storage version, add a new entry for `v1beta1`:

   ```yaml
   versions:
   - name: v1alpha1
     served: true
     storage: false
   - name: v1beta1
     served: true
     storage: true
   ```
1. New entry.

2. Ensure the referencing version of the CRD in the owned section of your CSV is updated if the CSV intends to use the new version:

```yaml
customresourcedefinitions:
  owned:
    - name: cluster.example.com
      version: v1beta1
      kind: cluster
      displayName: Cluster
```

1. Update the version.

3. Push the updated CRD and CSV to your bundle.

### 5.6.8.3.2. Deprecating or removing a CRD version

Operator Lifecycle Manager (OLM) does not allow a serving version of a custom resource definition (CRD) to be removed right away. Instead, a deprecated version of the CRD must be first disabled by setting the served field in the CRD to false. Then, the non-serving version can be removed on the subsequent CRD upgrade.

**Procedure**

To deprecate and remove a specific version of a CRD:

1. Mark the deprecated version as non-serving to indicate this version is no longer in use and may be removed in a subsequent upgrade. For example:

```yaml
versions:
  - name: v1alpha1
    served: false
    storage: true
```

1. Set to false.

2. Switch the storage version to a serving version if the version to be deprecated is currently the storage version. For example:

```yaml
versions:
  - name: v1alpha1
    served: false
    storage: false
  - name: v1beta1
    served: true
    storage: true
```

1 2 Update the storage fields accordingly.
NOTE

To remove a specific version that is or was the storage version from a CRD, that version must be removed from the storedVersion in the status of the CRD. OLM will attempt to do this for you if it detects a stored version no longer exists in the new CRD.

3. Upgrade the CRD with the above changes.

4. In subsequent upgrade cycles, the non-serving version can be removed completely from the CRD. For example:

```yaml
versions:
  - name: v1beta1
    served: true
    storage: true
```

5. Ensure the referencing CRD version in the owned section of your CSV is updated accordingly if that version is removed from the CRD.

5.6.8.4. CRD templates

Users of your Operator must be made aware of which options are required versus optional. You can provide templates for each of your custom resource definitions (CRDs) with a minimum set of configuration as an annotation named alm-examples. Compatible UIs will pre-fill this template for users to further customize.

The annotation consists of a list of the kind, for example, the CRD name and the corresponding metadata and spec of the Kubernetes object.

The following full example provides templates for EtcdCluster, EtcdBackup and EtcdRestore:

```yaml
metadata:
  annotations:
    alm-examples: >-
      [{"apiVersion":"etcd.database.coreos.com/v1beta2","kind":"EtcdCluster","metadata":
        {"name":"example","namespace":"default"},{"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>}}]
```

5.6.8.5. Hiding internal objects

It is common practice for Operators to use custom resource definitions (CRDs) internally to accomplish a task. These objects are not meant for users to manipulate and can be confusing to users of the Operator. For example, a database Operator might have a Replication CRD that is created whenever a user creates a Database object with replication: true.

As an Operator author, you can hide any CRDs in the user interface that are not meant for user manipulation by adding the operators.operatorframework.io/internal-objects annotation to the cluster service version (CSV) of your Operator.
Procedure

1. Before marking one of your CRDs as internal, ensure that any debugging information or configuration that might be required to manage the application is reflected on the status or `spec` block of your CR, if applicable to your Operator.

2. Add the `operators.operatorframework.io/internal-objects` annotation to the CSV of your Operator to specify any internal objects to hide in the user interface:

   **Internal object annotation**

   ```yaml
   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 Set any internal CRDs as an array of strings.
   
5.6.8.6. Initializing required custom resources

An Operator might require the user to instantiate a custom resource before the Operator can be fully functional. However, it can be challenging for a user to determine what is required or how to define the resource.

As an Operator developer, you can specify a single required custom resource that must be created at the time that the Operator is installed by adding the `operatorframework.io/initialization-resource` annotation to the cluster service version (CSV). The annotation must include a template that contains a complete YAML definition that is required to initialize the resource during installation.

If this annotation is defined, after installing the Operator from the OpenShift Container Platform web console, the user is prompted to create the resource using the template provided in the CSV.

Procedure

- Add the `operatorframework.io/initialization-resource` annotation to the CSV of your Operator to specify a required custom resource. For example, the following annotation requires the creation of a `StorageCluster` resource and provides a full YAML definition:

   **Initialization resource annotation**

   ```yaml
   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": {
   ```
5.6.9. Understanding your API services

As with CRDs, there are two types of API services that your Operator may use: owned and required.

5.6.9.1. Owned API services

When a CSV owns an API service, it is responsible for describing the deployment of the extension api-server that backs it and the group/version/kind (GVK) it provides.

An API service is uniquely identified by the group/version it provides and can be listed multiple times to denote the different kinds it is expected to provide.
Table 5.14. Owned API service fields

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
<th>Required/optional</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Group</strong></td>
<td>Group that the API service provides, for example database.example.com.</td>
<td>Required</td>
</tr>
<tr>
<td><strong>Version</strong></td>
<td>Version of the API service, for example v1alpha1.</td>
<td>Required</td>
</tr>
<tr>
<td><strong>Kind</strong></td>
<td>A kind that the API service is expected to provide.</td>
<td>Required</td>
</tr>
<tr>
<td><strong>Name</strong></td>
<td>The plural name for the API service provided.</td>
<td>Required</td>
</tr>
<tr>
<td><strong>DeploymentName</strong></td>
<td>Name of the deployment defined by your CSV that corresponds to your API service (required for owned API services). During the CSV pending phase, the OLM Operator searches the InstallStrategy of your CSV for a Deployment spec with a matching name, and if not found, does not transition the CSV to the &quot;Install Ready&quot; phase.</td>
<td>Required</td>
</tr>
<tr>
<td><strong>DisplayName</strong></td>
<td>A human readable version of your API service name, for example MongoDB Standalone.</td>
<td>Required</td>
</tr>
<tr>
<td><strong>Description</strong></td>
<td>A short description of how this API service is used by the Operator or a description of the functionality provided by the API service.</td>
<td>Required</td>
</tr>
<tr>
<td><strong>Resources</strong></td>
<td>Your API services own one or more types of Kubernetes objects. These are listed in the resources section to inform your users of the objects they might need to troubleshoot or how to connect to the application, such as the service or ingress rule that exposes a database. It is recommended to only list out the objects that are important to a human, not an exhaustive list of everything you orchestrate. For example, do not list config maps that store internal state that are not meant to be modified by a user.</td>
<td>Optional</td>
</tr>
<tr>
<td><strong>SpecDescriptors</strong>, <strong>StatusDescriptors</strong>, and <strong>ActionDescriptors</strong></td>
<td>Essentially the same as for owned CRDs.</td>
<td>Optional</td>
</tr>
</tbody>
</table>

5.6.9.1.1. API service resource creation

Operator Lifecycle Manager (OLM) is responsible for creating or replacing the service and API service resources for each unique owned API service:

- Service pod selectors are copied from the CSV deployment matching the DeploymentName field of the API service description.
5.6.9.1.2. API service serving certificates

OLM handles generating a serving key/certificate pair whenever an owned API service is being installed. The serving certificate has a common name (CN) containing the hostname of the generated Service resource and is signed by the private key of the CA bundle embedded in the corresponding API service resource.

The certificate is stored as a type `kubernetes.io/tls` secret in the deployment namespace, and a volume named `apiservice-cert` is automatically appended to the volumes section of the deployment in the CSV matching the `DeploymentName` field of the API service description.

If one does not already exist, a volume mount with a matching name is also appended to all containers of that deployment. This allows users to define a volume mount with the expected name to accommodate any custom path requirements. The path of the generated volume mount defaults to `/apiserver.local.config/certificates` and any existing volume mounts with the same path are replaced.

5.6.9.2. Required API services

OLM ensures all required CSVs have an API service that is available and all expected GVKs are discoverable before attempting installation. This allows a CSV to rely on specific kinds provided by API services it does not own.

Table 5.15. Required API service fields

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
<th>Required/optional</th>
</tr>
</thead>
<tbody>
<tr>
<td>Group</td>
<td>Group that the API service provides, for example <code>database.example.com</code>.</td>
<td>Required</td>
</tr>
<tr>
<td>Version</td>
<td>Version of the API service, for example <code>v1alpha1</code>.</td>
<td>Required</td>
</tr>
<tr>
<td>Kind</td>
<td>A kind that the API service is expected to provide.</td>
<td>Required</td>
</tr>
<tr>
<td>DisplayName</td>
<td>A human readable version of your API service name, for example <code>MongoDB Standalone</code>.</td>
<td>Required</td>
</tr>
<tr>
<td>Description</td>
<td>A short description of how this API service is used by the Operator or a description of the functionality provided by the API service.</td>
<td>Required</td>
</tr>
</tbody>
</table>

5.7. WORKING WITH BUNDLE IMAGES

You can use the Operator SDK to package, deploy, and upgrade Operators in the Bundle Format on Operator Lifecycle Manager (OLM).

5.7.1. Bundling an Operator and deploying with Operator Lifecycle Manager

Operator Lifecycle Manager (OLM) helps you to install, update, and generally manage the lifecycle of Operators and their associated services on a Kubernetes cluster. OLM is installed by default on
OpenShift Container Platform and runs as a Kubernetes extension so that you can use the web console and the OpenShift CLI (oc) for all Operator lifecycle management functions without any additional tools.

The Operator Bundle Format is the default packaging method for Operator SDK and OLM. You can get your Operator ready for OLM by using the Operator SDK to build, push, validate, and run a bundle image with OLM.

Prerequisites

- Operator SDK CLI installed on a development workstation
- OpenShift CLI (oc) v4.7+ installed
- Operator Lifecycle Manager (OLM) installed on a Kubernetes-based cluster (v1.16.0 or later if you use apiextensions.k8s.io/v1 CRDs, for example OpenShift Container Platform 4.7)
- Logged into the cluster with oc using an account with cluster-admin permissions
- Operator project initialized by using the Operator SDK
- If your Operator is Go-based, your project must have been updated to use supported images for running on OpenShift Container Platform

Procedure

1. Run the following make commands in your Operator project directory to build and push your Operator image. Modify the IMG argument in the following steps to reference a repository that you have access to. You can obtain an account for storing containers at repository sites such as Quay.io.
   a. Build the image:

   $$ $ \textit{make docker-build IMG=registry}\textbackslash user\textbackslash operator\textunderscore image\textunderscore name\textbackslash tag}$$

   b. Push the image to a repository:

   $$ $ \textit{make docker-push IMG=registry}\textbackslash user\textbackslash operator\textunderscore image\textunderscore name\textbackslash tag}$$

2. Update your Makefile by setting the IMG URL to your Operator image name and tag that you pushed:

   $$ $ \# \text{ Image URL to use all building/pushing image targets}$
   \texttt{IMG} \?= \texttt{registry}\textbackslash user\textbackslash operator\textunderscore image\textunderscore name\textbackslash tag}$$

   This value is used for subsequent operations.

3. Create your Operator bundle manifest by running the make bundle command, which invokes several commands, including the Operator SDK generate bundle and bundle validate subcommands:

   $$ $ \textit{make bundle}$$

Bundle manifests for an Operator describe how to display, create, and manage an application. The make bundle command creates the following files and directories in your Operator project:
- A bundle manifests directory named `bundle/manifests` that contains a `ClusterServiceVersion` object

- A bundle metadata directory named `bundle/metadata`

- All custom resource definitions (CRDs) in a `config/crd` directory

- A Dockerfile `bundle.Dockerfile`

These files are then automatically validated by using `operator-sdk bundle validate` to ensure the on-disk bundle representation is correct.

4. Build and push your bundle image by running the following commands. OLM consumes Operator bundles using an index image, which reference one or more bundle images.

   a. Build the bundle image. Set `BUNDLE_IMAGE` with the details for the registry, user namespace, and image tag where you intend to push the image:

      ```
      $ make bundle-build BUNDLE_IMG=<registry>/<user>/<bundle_image_name>:<tag>
      ```

   b. Push the bundle image:

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

5. Check the status of OLM on your cluster by using the following Operator SDK command:

   ```
   $ operator-sdk olm status \
   --olm-namespace=openshift-operator-lifecycle-manager
   ```

6. Run the Operator on your cluster by using the OLM integration in Operator SDK:

   ```
   $ operator-sdk run bundle \
   [-n <namespace>] \
   <registry>/<user>/<bundle_image_name>:<tag>
   ```

   By default, the command installs the Operator in the currently active project in your `~/.kube/config` file. You can add the `-n` flag to set a different namespace scope for the installation.

This command performs the following actions:

- Create an index image with your bundle image injected.

- Create a catalog source that points to your new index image, which enables OperatorHub to discover your Operator.

- Deploy your Operator to your cluster by creating an Operator group, subscription, install plan, and all other required objects, including RBAC.

5.7.2. Testing an Operator upgrade on Operator Lifecycle Manager

You can quickly test upgrading your Operator by using Operator Lifecycle Manager (OLM) integration in the Operator SDK, without requiring you to manually manage index images and catalog sources.
The **run bundle-upgrade** subcommand automates triggering an installed Operator to upgrade to a later version by specifying a bundle image for the later version.

**Prerequisites**

- Operator installed with OLM by using the **run bundle** subcommand
- A bundle image that represents a later version of the installed Operator

**Procedure**

1. If your Operator has not already been installed on OLM with the **run bundle** subcommand, install the earlier version of your Operator by specifying the bundle image. For example, for a Memcached Operator:

   ```
   $ operator-sdk run bundle <registry>/<user>/memcached-operator:v0.0.1
   ```

   **Example output**

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

2. Upgrade the installed Operator by specifying the bundle image for the later Operator version:

   ```
   $ operator-sdk run bundle-upgrade <registry>/<user>/memcached-operator:v0.0.2
   ```

   **Example output**

   ```
   INFO[0002] Found existing subscription with name memcached-operator-v0-0-1-sub and namespace my-project
   INFO[0002] Found existing catalog source with name memcached-operator-catalog and namespace my-project
   INFO[0009] Updated catalog source memcached-operator-catalog with address and annotations
   INFO[0010] Deleted previous registry pod with name "quay-io-demo-memcached-operator-v0-0-1"
   INFO[0041] Approved InstallPlan install-gvcjh for the Subscription: memcached-operator-v0-0-1-sub
   INFO[0042] Waiting for ClusterServiceVersion "my-project/memcached-operator.v0.0.2" to reach 'Succeeded' phase
   INFO[0042]   Found ClusterServiceVersion "my-project/memcached-operator.v0.0.2" phase: InstallReady
   INFO[0043]   Found ClusterServiceVersion "my-project/memcached-operator.v0.0.2" phase:
Clean up the installed Operators:

```
$ operator-sdk cleanup memcached-operator
```

### 5.7.3. Additional resources

- See [Operator Framework packaging formats](#) for details on the Bundle Format.
- See [Managing custom catalogs](#) for details on adding bundle images to index images by using the `opm` command.
- See [Operator Lifecycle Manager workflow](#) for details on how upgrades work for installed Operators.

### 5.8. VALIDATING OPERATORS USING THE SCORECARD TOOL

As an Operator author, you can use the scorecard tool in the Operator SDK to do the following tasks:

- Validate that your Operator project is free of syntax errors and packaged correctly
- Review suggestions about ways you can improve your Operator

#### 5.8.1. About the scorecard tool

While the Operator SDK `bundle validate` subcommand can validate local bundle directories and remote bundle images for content and structure, you can use the `scorecard` command to run tests on your Operator based on a configuration file and test images. These tests are implemented within test images that are configured and constructed to be executed by the scorecard.

The scorecard assumes it is run with access to a configured Kubernetes cluster, such as OpenShift Container Platform. The scorecard runs each test within a pod, from which pod logs are aggregated and test results are sent to the console. The scorecard has built-in basic and Operator Lifecycle Manager (OLM) tests and also provides a means to execute custom test definitions.

**Scorecard workflow**

1. Create all resources required by any related custom resources (CRs) and the Operator
2. Create a proxy container in the deployment of the Operator to record calls to the API server and run tests
3. Examine parameters in the CRs

The scorecard tests make no assumptions as to the state of the Operator being tested. Creating Operators and CRs for an Operators are beyond the scope of the scorecard itself. Scorecard tests can, however, create whatever resources they require if the tests are designed for resource creation.

**scorecard command syntax**
$ operator-sdk scorecard <bundle_dir_or_image> [flags]

The scorecard requires a positional argument for either the on-disk path to your Operator bundle or the name of a bundle image.

For further information about the flags, run:

$ operator-sdk scorecard -h

5.8.2. Scorecard configuration

The scorecard tool uses a configuration that allows you to configure internal plug-ins, as well as several global configuration options. Tests are driven by a configuration file named `config.yaml`, which is generated by the `make bundle` command, located in your `bundle/` directory:

```
./bundle
...
  tests
    └── scorecard
        └── config.yaml
```

Example scorecard configuration file

```
kind: Configuration
apiVersion: scorecard.operatorframework.io/v1alpha3
metadata:
  name: config
stages:
  - parallel: true
tests:
  - image: quay.io/operator-framework/scorecard-test:v1.3.0
    entrypoint:
      - scorecard-test
      - basic-check-spec
    labels:
      suite: basic
      test: basic-check-spec-test
  - image: quay.io/operator-framework/scorecard-test:v1.3.0
    entrypoint:
      - scorecard-test
      - olm-bundle-validation
    labels:
      suite: olm
      test: olm-bundle-validation-test
```

The configuration file defines each test that scorecard can execute. The following fields of the scorecard configuration file define the test as follows:

<table>
<thead>
<tr>
<th>Configuration field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>image</td>
<td>Test container image name that implements a test</td>
</tr>
</tbody>
</table>
5.8.3. Built-in scorecard tests

The scorecard ships with pre-defined tests that are arranged into suites: the basic test suite and the Operator Lifecycle Manager (OLM) suite.

<table>
<thead>
<tr>
<th>Configuration field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>entrypoint</strong></td>
<td>Command and arguments that are invoked in the test image to execute a test</td>
</tr>
<tr>
<td><strong>labels</strong></td>
<td>Scorecard-defined or custom labels that select which tests to run</td>
</tr>
</tbody>
</table>

**Table 5.16. Basic test suite**

<table>
<thead>
<tr>
<th>Test</th>
<th>Description</th>
<th>Short name</th>
</tr>
</thead>
<tbody>
<tr>
<td>Spec Block Exists</td>
<td>This test checks the custom resource (CR) created in the cluster to make sure that all CRs have a <code>spec</code> block.</td>
<td><strong>basic-check-spec-test</strong></td>
</tr>
</tbody>
</table>

**Table 5.17. OLM test suite**

<table>
<thead>
<tr>
<th>Test</th>
<th>Description</th>
<th>Short name</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bundle Validation</td>
<td>This test validates the bundle manifests found in the bundle that is passed into scorecard. If the bundle contents contain errors, then the test result output includes the validator log as well as error messages from the validation library.</td>
<td><strong>olm-bundle-validation-test</strong></td>
</tr>
<tr>
<td>Provided APIs Have Validation</td>
<td>This test verifies that the custom resource definitions (CRDs) for the provided CRs contain a validation section and that there is validation for each <code>spec</code> and <code>status</code> field detected in the CR.</td>
<td><strong>olm-crds-have-validation-test</strong></td>
</tr>
<tr>
<td>Owned CRDs Have Resources Listed</td>
<td>This test makes sure that the CRDs for each CR provided via the <code>cr-manifest</code> option have a <code>resources</code> subsection in the <code>owned</code> CRDs section of the ClusterServiceVersion (CSV). If the test detects used resources that are not listed in the resources section, it lists them in the suggestions at the end of the test. Users are required to fill out the resources section after initial code generation for this test to pass.</td>
<td><strong>olm-crds-have-resources-test</strong></td>
</tr>
<tr>
<td>Spec Fields With Descriptors</td>
<td>This test verifies that every field in the CRs <code>spec</code> sections has a corresponding descriptor listed in the CSV.</td>
<td><strong>olm-spec-descriptors-test</strong></td>
</tr>
<tr>
<td>Status Fields With Descriptors</td>
<td>This test verifies that every field in the CRs <code>status</code> sections have a corresponding descriptor listed in the CSV.</td>
<td><strong>olm-status-descriptors-test</strong></td>
</tr>
</tbody>
</table>
5.8.4. Running the scorecard tool

A default set of Kustomize files are generated by the Operator SDK after running the init command. The default bundle/tests/scorecard/config.yaml file that is generated can be immediately used to run the scorecard tool against your Operator, or you can modify this file to your test specifications.

Prerequisites

- Operator project generated by using the Operator SDK

Procedure

1. Generate or regenerate your bundle manifests and metadata for your Operator:

   ```shell
   $ make bundle
   ```
   
   This command automatically adds scorecard annotations to your bundle metadata, which is used by the `scorecard` command to run tests.

2. Run the scorecard against the on-disk path to your Operator bundle or the name of a bundle image:

   ```shell
   $ operator-sdk scorecard <bundle_dir_or_image>
   ```

5.8.5. Scorecard output

The `--output` flag for the `scorecard` command specifies the scorecard results output format: either text or json.

Example 5.2. Example JSON output snippet

```json
{
   "apiVersion": "scorecard.operatorframework.io/v1alpha3",
   "kind": "TestList",
   "items": [
   {
      "kind": "Test",
      "apiVersion": "scorecard.operatorframework.io/v1alpha3",
      "spec": {
         "image": "quay.io/operator-framework/scorecard-test:v1.3.0",
         "entrypoint": [
            "scorecard-test",
            "olm-bundle-validation"
         ],
         "labels": {
            "suite": "olm",
            "test": "olm-bundle-validation-test"
         }
      },
      "status": {
         "results": [
         {
            "name": "olm-bundle-validation",
            "log": "time="2020-06-10T19:02:49Z" level=debug msg=\"Found manifests directory\"
         }
         ]
      }
   }
}
```
Example 5.3. Example text output snippet

```
---
Image:      quay.io/operator-framework/scorecard-test:v1.3.0
Entrypoint: [scorecard-test olm-bundle-validation]
Labels:
  "suite": "olm"
  "test": "olm-bundle-validation-test"
Results:
  Name: olm-bundle-validation
  State: pass
Log:
  time="2020-07-15T03:19:02Z" level=debug msg="Found manifests directory" name=bundle-test
  time="2020-07-15T03:19:02Z" level=debug msg="Found metadata directory" name=bundle-test
  time="2020-07-15T03:19:02Z" level=debug msg="Getting mediaType info from manifests directory" name=bundle-test
  time="2020-07-15T03:19:02Z" level=info msg="Found annotations file" name=bundle-test
  time="2020-07-15T03:19:02Z" level=info msg="Could not find optional dependencies file" name=bundle-test
---
```

NOTE

The output format spec matches the Test type layout.

### 5.8.6. Selecting tests

Scorecard tests are selected by setting the `--selector` CLI flag to a set of label strings. If a selector flag is not supplied, then all the tests within the scorecard configuration file are run.

Tests are run serially with test results being aggregated by the scorecard and written to standard output, or `stdout`.

**Procedure**

1. To select a single test, for example `basic-check-spec-test`, specify the test by using the `--selector` flag:
2. To select a suite of tests, for example **olm**, specify a label that is used by all of the OLM tests:

```
$ operator-sdk scorecard <bundle_dir_or_image> \
   -o text \n   --selector=suite=olm
```

3. To select multiple tests, specify the test names by using the **selector** flag using the following syntax:

```
$ operator-sdk scorecard <bundle_dir_or_image> \
   -o text \n   --selector='test in (basic-check-spec-test,olm-bundle-validation-test)'
```

### 5.8.7. Enabling parallel testing

As an Operator author, you can define separate stages for your tests using the scorecard configuration file. Stages run sequentially in the order they are defined in the configuration file. A stage contains a list of tests and a configurable **parallel** setting.

By default, or when a stage explicitly sets **parallel** to **false**, tests in a stage are run sequentially in the order they are defined in the configuration file. Running tests one at a time is helpful to guarantee that no two tests interact and conflict with each other.

However, if tests are designed to be fully isolated, they can be parallelized.

**Procedure**

- To run a set of isolated tests in parallel, include them in the same stage and set **parallel** to **true**:

```yaml
apiVersion: scorecard.operatorframework.io/v1alpha3
kind: Configuration
metadata:
  name: config
stages:
- parallel: true 1
  tests:
  - entrypoint:
    - scorecard-test
    - basic-check-spec
    image: quay.io/operator-framework/scorecard-test:v1.3.0
    labels:
      suite: basic
      test: basic-check-spec-test
  - entrypoint:
    - scorecard-test
    - olm-bundle-validation
    image: quay.io/operator-framework/scorecard-test:v1.3.0
    labels:
      suite: olm
      test: olm-bundle-validation-test
```
Enables parallel testing

All tests in a parallel stage are executed simultaneously, and scorecard waits for all of them to finish before proceeding to the next stage. This can make your tests run much faster.

5.8.8. Custom scorecard tests

The scorecard tool can run custom tests that follow these mandated conventions:

- Tests are implemented within a container image
- Tests accept an entrypoint which include a command and arguments
- Tests produce `v1alpha3` scorecard output in JSON format with no extraneous logging in the test output
- Tests can obtain the bundle contents at a shared mount point of `/bundle`
- Tests can access the Kubernetes API using an in-cluster client connection

Writing custom tests in other programming languages is possible if the test image follows the above guidelines.

The following example shows a custom test image written in Go:

```
package main

import (
    "encoding/json"
    "fmt"
    "log"
    "os"

scapiv1alpha3 "github.com/operator-framework/api/pkg/apis/scorecard/v1alpha3"
apimanifests "github.com/operator-framework/api/pkg/manifests"
)

// This is the custom scorecard test example binary
// As with the Redhat scorecard test image, the bundle that is under
```
// test is expected to be mounted so that tests can inspect the
// bundle contents as part of their test implementations.
// The actual test to be run is named and that name is passed
// as an argument to this binary. This argument mechanism allows
// this binary to run various tests all from within a single
// test image.

const PodBundleRoot = "/bundle"

func main() {
    entrypoint := os.Args[1:]
    if len(entrypoint) == 0 {
        log.Fatal("Test name argument is required")
    }

    // Read the pod's untar'd bundle from a well-known path.
    cfg, err := apimanifests.GetBundleFromDir(PodBundleRoot)
    if err != nil {
        log.Fatal(err.Error())
    }

    var result scapiv1alpha3.TestStatus

    // Names of the custom tests which would be passed in the
    // `operator-sdk` command.
    switch entrypoint[0] {
    case CustomTest1Name:
        result = CustomTest1(cfg)
    case CustomTest2Name:
        result = CustomTest2(cfg)
    default:
        result = printValidTests()
    }

    // Convert scapiv1alpha3.TestResult to json.
    prettyJSON, err := json.MarshalIndent(result, "", "    ")
    if err != nil {
        log.Fatal("Failed to generate json", err)
    }
    fmt.Printf("%s
", string(prettyJSON))
}

// printValidTests will print out full list of test names to give a hint to the end user on what the valid
tests are.
func printValidTests() scapiv1alpha3.TestStatus {
    result := scapiv1alpha3.TestResult{}
    result.State = scapiv1alpha3.FailState
    result.Errors = make([]string, 0)
    result.Suggestions = make([]string, 0)

    str := fmt.Sprintf("Valid tests for this image include: %s %s", CustomTest1Name, CustomTest2Name)
    result.Errors = append(result.Errors, str)
    return scapiv1alpha3.TestStatus{

5.9. CONFIGURING BUILT-IN MONITORING WITH PROMETHEUS

This guide describes the built-in monitoring support provided by the Operator SDK using the Prometheus Operator and details usage for Operator authors.

5.9.1. Prometheus Operator support
Prometheus is an open-source systems monitoring and alerting toolkit. The Prometheus Operator creates, configures, and manages Prometheus clusters running on Kubernetes-based clusters, such as OpenShift Container Platform.

Helper functions exist in the Operator SDK by default to automatically set up metrics in any generated Go-based Operator for use on clusters where the Prometheus Operator is deployed.

5.9.2. Metrics helper

In Go-based Operators generated using the Operator SDK, the following function exposes general metrics about the running program:

```go
```

These metrics are inherited from the controller-runtime library API. By default, the metrics are served on `0.0.0.0:8383/metrics`.

A `Service` object is created with the metrics port exposed, which can be then accessed by Prometheus. The `Service` object is garbage collected when the leader pod’s `root` owner is deleted.

The following example is present in the `cmd/manager/main.go` file in all Operators generated using the Operator SDK:

```go
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 The host that the metrics are exposed on.
2 The port that the metrics are exposed on.
5.9.2.1. Modifying the metrics port
Operator authors can modify the port that metrics are exposed on.

Prerequisites
- Go-based Operator generated using the Operator SDK
- Kubernetes-based cluster with the Prometheus Operator deployed

Procedure
- In the `cmd/manager/main.go` file of the generated Operator, change the value of `metricsPort` in the following line:

```
var metricsPort int32 = 8383
```

5.9.3. Service monitors

A `ServiceMonitor` is a custom resource provided by the Prometheus Operator that discovers the `Endpoints` in `Service` objects and configures Prometheus to monitor those pods.

In Go-based Operators generated using the Operator SDK, the `GenerateServiceMonitor()` helper function can take a `Service` object and generate a `ServiceMonitor` object based on it.

Additional resources
- See the Prometheus Operator documentation for more information about the `ServiceMonitor` custom resource definition (CRD).

5.9.3.1. Creating service monitors

Operator authors can add service target discovery of created monitoring services using the `metrics.CreateServiceMonitor()` helper function, which accepts the newly created service.

Prerequisites
- Go-based Operator generated using the Operator SDK
- Kubernetes-based cluster with the Prometheus Operator deployed

Procedure
- Add the `metrics.CreateServiceMonitor()` helper function to your Operator code:

```
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.
```
5.10. CONFIGURING LEADER ELECTION

During the lifecycle of an Operator, it is possible that there may be more than one instance running at any given time, for example when rolling out an upgrade for the Operator. In such a scenario, it is necessary to avoid contention between multiple Operator instances using leader election. This ensures only one leader instance handles the reconciliation while the other instances are inactive but ready to take over when the leader steps down.

There are two different leader election implementations to choose from, each with its own trade-off:

**Leader-for-life**

The leader pod only gives up leadership, using garbage collection, when it is deleted. This implementation precludes the possibility of two instances mistakenly running as leaders, a state also known as split brain. However, this method can be subject to a delay in electing a new leader. For example, when the leader pod is on an unresponsive or partitioned node, the pod-eviction-timeout dictates long how it takes for the leader pod to be deleted from the node and step down, with a default of 5m. See the Leader-for-life Go documentation for more.

**Leader-with-lease**

The leader pod periodically renews the leader lease and gives up leadership when it cannot renew the lease. This implementation allows for a faster transition to a new leader when the existing leader is isolated, but there is a possibility of split brain in certain situations. See the Leader-with-lease Go documentation for more.

By default, the Operator SDK enables the Leader-for-life implementation. Consult the related Go documentation for both approaches to consider the trade-offs that make sense for your use case.

5.10.1. Operator leader election examples

The following examples illustrate how to use the two leader election options for an Operator, Leader-for-life and Leader-with-lease.

5.10.1.1. Leader-for-life election

With the Leader-for-life election implementation, a call to `leader.Become()` blocks the Operator as it retries until it can become the leader by creating the config map named `memcached-operator-lock`.

```go
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.
s := "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, s, services)
if err != nil {
    // Handle errors here.
}
...```
If the Operator is not running inside a cluster, `leader.Become()` simply returns without error to skip the leader election since it cannot detect the name of the Operator.

### 5.10.1.2. Leader-with-lease election

The Leader-with-lease implementation can be enabled using the Manager Options for leader election:

```go
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)
  ...
}
```

When the Operator is not running in a cluster, the Manager returns an error when starting because it cannot detect the namespace of the Operator to create the config map for leader election. You can override this namespace by setting the `LeaderElectionNamespace` option for the Manager.

### 5.11. OPERATOR SDK CLI REFERENCE

The Operator SDK command-line interface (CLI) is a development kit designed to make writing Operators easier.

**Operator SDK CLI syntax**

```
$ operator-sdk <command> [subcommand] [argument] [flags]
```

Operator authors with cluster administrator access to a Kubernetes-based cluster (such as OpenShift Container Platform) can use the Operator SDK CLI to develop their own Operators based on Go, Ansible, or Helm. Kubebuilder is embedded into the Operator SDK as the scaffolding solution for Go-
based Operators, which means existing Kubebuilder projects can be used as is with the Operator SDK and continue to work.

### 5.11.1. bundle

The operator-sdk bundle command manages Operator bundle metadata.

#### 5.11.1.1. validate

The bundle validate subcommand validates an Operator bundle.

**Table 5.18. bundle validate flags**

<table>
<thead>
<tr>
<th>Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-h, --help</td>
<td>Help output for the bundle validate subcommand.</td>
</tr>
<tr>
<td>--index-builder (string)</td>
<td>Tool to pull and unpack bundle images. Only used when validating a bundle image. Available options are docker, which is the default, podman, or none.</td>
</tr>
<tr>
<td>--list-optional</td>
<td>List all optional validators available. When set, no validators are run.</td>
</tr>
<tr>
<td>--select-optional (string)</td>
<td>Label selector to select optional validators to run. When run with the --list-optional flag, lists available optional validators.</td>
</tr>
</tbody>
</table>

### 5.11.2. cleanup

The operator-sdk cleanup command destroys and removes resources that were created for an Operator that was deployed with the run command.

**Table 5.19. cleanup flags**

<table>
<thead>
<tr>
<th>Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-h, --help</td>
<td>Help output for the run bundle subcommand.</td>
</tr>
<tr>
<td>--kubeconfig (string)</td>
<td>Path to the kubeconfig file to use for CLI requests.</td>
</tr>
<tr>
<td>n, --namespace (string)</td>
<td>If present, namespace in which to run the CLI request.</td>
</tr>
<tr>
<td>--timeout &lt;duration&gt;</td>
<td>Time to wait for the command to complete before failing. The default value is 2m0s.</td>
</tr>
</tbody>
</table>

### 5.11.3. completion

The operator-sdk completion command generates shell completions to make issuing CLI commands quicker and easier.

**Table 5.20. completion subcommands**

238
<table>
<thead>
<tr>
<th>Subcommand</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>bash</td>
<td>Generate bash completions.</td>
</tr>
<tr>
<td>zsh</td>
<td>Generate zsh completions.</td>
</tr>
</tbody>
</table>

Table 5.21. completion flags

<table>
<thead>
<tr>
<th>Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-h, --help</td>
<td>Usage help output.</td>
</tr>
</tbody>
</table>

For example:

```bash
$ operator-sdk completion bash
```

Example output

```
# bash completion for operator-sdk
# * shell-script -*-
...
# ex: ts=4 sw=4 et filetype=sh
```

5.11.4. create

The **operator-sdk create** command is used to create, or **scaffold**, a Kubernetes API.

5.11.4.1. api

The **create api** subcommand scaffolds a Kubernetes API. The subcommand must be run in a project that was initialized with the **init** command.

Table 5.22. create api flags

<table>
<thead>
<tr>
<th>Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-h, --help</td>
<td>Help output for the <strong>run bundle</strong> subcommand.</td>
</tr>
</tbody>
</table>

5.11.5. generate

The **operator-sdk generate** command invokes a specific generator to generate code or manifests.

5.11.5.1. bundle

The **generate bundle** subcommand generates a set of bundle manifests, metadata, and a **bundle.Dockerfile** file for your Operator project.
Typically, you run the `generate kustomize manifests` subcommand first to generate the input Kustomize bases that are used by the `generate bundle` subcommand. However, you can use the `make bundle` command in an initialized project to automate running these commands in sequence.

Table 5.23. generate bundle flags

<table>
<thead>
<tr>
<th>Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>--channels</code> (string)</td>
<td>Comma-separated list of channels to which the bundle belongs. The default value is <code>alpha</code>.</td>
</tr>
<tr>
<td><code>--crds-dir</code> (string)</td>
<td>Root directory for <code>CustomResourceDefinition</code> manifests.</td>
</tr>
<tr>
<td><code>--default-channel</code> (string)</td>
<td>The default channel for the bundle.</td>
</tr>
<tr>
<td><code>--deploy-dir</code> (string)</td>
<td>Root directory for Operator manifests, such as deployments and RBAC. This directory is different from the directory passed to the <code>--input-dir</code> flag.</td>
</tr>
<tr>
<td><code>-h, --help</code></td>
<td>Help for <code>generate bundle</code></td>
</tr>
<tr>
<td><code>--input-dir</code> (string)</td>
<td>Directory from which to read an existing bundle. This directory is the parent of your bundle <code>manifests</code> directory and is different from the <code>--deploy-dir</code> directory.</td>
</tr>
<tr>
<td><code>--kustomize-dir</code> (string)</td>
<td>Directory containing Kustomize bases and a <code>kustomization.yaml</code> file for bundle manifests. The default path is <code>config/manifests</code>.</td>
</tr>
<tr>
<td><code>--manifests</code></td>
<td>Generate bundle manifests.</td>
</tr>
<tr>
<td><code>--metadata</code></td>
<td>Generate bundle metadata and Dockerfile.</td>
</tr>
<tr>
<td><code>--output-dir</code> (string)</td>
<td>Directory to write the bundle to.</td>
</tr>
<tr>
<td><code>--overwrite</code></td>
<td>Overwrite the bundle metadata and Dockerfile if they exist. The default value is <code>true</code>.</td>
</tr>
<tr>
<td><code>--package</code> (string)</td>
<td>Package name for the bundle.</td>
</tr>
<tr>
<td><code>-q, --quiet</code></td>
<td>Run in quiet mode.</td>
</tr>
<tr>
<td><code>--stdout</code></td>
<td>Write bundle manifest to standard out.</td>
</tr>
<tr>
<td><code>--version</code> (string)</td>
<td>Semantic version of the Operator in the generated bundle. Set only when creating a new bundle or upgrading the Operator.</td>
</tr>
</tbody>
</table>
Additional resources

- See Bundling an Operator and deploying with Operator Lifecycle Manager for a full procedure that includes using the `make bundle` command to call the `generate bundle` subcommand.

5.11.5.2. kustomize

The `generate kustomize` subcommand contains subcommands that generate Kustomize data for the Operator.

5.11.5.2.1. manifests

The `generate kustomize manifests` subcommand generates or regenerates Kustomize bases and a `kustomization.yaml` file in the `config/manifests` directory, which are used to build bundle manifests by other Operator SDK commands. This command interactively asks for UI metadata, an important component of manifest bases, by default unless a base already exists or you set the `--interactive=false` flag.

Table 5.24. `generate kustomize manifests` flags

<table>
<thead>
<tr>
<th>Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>--apis-dir</code> (string)</td>
<td>Root directory for API type definitions.</td>
</tr>
<tr>
<td><code>-h</code>, <code>--help</code></td>
<td>Help for <code>generate kustomize manifests</code>.</td>
</tr>
<tr>
<td><code>--input-dir</code> (string)</td>
<td>Directory containing existing Kustomize files.</td>
</tr>
<tr>
<td><code>--interactive</code></td>
<td>When set to <code>false</code>, if no Kustomize base exists, an interactive command prompt is presented to accept custom metadata.</td>
</tr>
<tr>
<td><code>--output-dir</code> (string)</td>
<td>Directory where to write Kustomize files.</td>
</tr>
<tr>
<td><code>--package</code> (string)</td>
<td>Package name.</td>
</tr>
<tr>
<td><code>-q</code>, <code>--quiet</code></td>
<td>Run in quiet mode.</td>
</tr>
</tbody>
</table>

5.11.6. init

The `operator-sdk init` command initializes an Operator project and generates, or scaffolds, a default project directory layout for the given plug-in.

This command writes the following files:

- Boilerplate license file
- `PROJECT` file with the domain and repository
- `Makefile` to build the project
- `go.mod` file with project dependencies
- `kustomization.yaml` file for customizing manifests
- Patch file for customizing images for manager manifests
- Patch file for enabling Prometheus metrics
- **main.go** file to run

### Table 5.25. **init** flags

<table>
<thead>
<tr>
<th>Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>--help, -h</td>
<td>Help output for the <strong>init</strong> command.</td>
</tr>
<tr>
<td>--plugins (string)</td>
<td>Name and optionally version of the plug-in to initialize the project with. Available plug-ins are ansible.sdk.operatorframework.io/v1, go.kubebuilder.io/v2, go.kubebuilder.io/v3, and helm.sdk.operatorframework.io/v1.</td>
</tr>
<tr>
<td>--project-version</td>
<td>Project version. Available values are 2 and 3-alpha, which is the default.</td>
</tr>
</tbody>
</table>

#### 5.11.7. **run**

The **operator-sdk run** command provides options that can launch the Operator in various environments.

#### 5.11.7.1. **bundle**

The **run bundle** subcommand deploys an Operator in the bundle format with Operator Lifecycle Manager (OLM).

### Table 5.26. **run bundle** flags

<table>
<thead>
<tr>
<th>Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>--index-image (string)</td>
<td>Index image in which to inject a bundle. The default image is quay.io/operator-framework/upstream-opm-builder:latest.</td>
</tr>
<tr>
<td>--install-mode &lt;install_mode_value&gt;</td>
<td>Install mode supported by the cluster service version (CSV) of the Operator, for example AllNamespaces or SingleNamespace.</td>
</tr>
<tr>
<td>--timeout &lt;duration&gt;</td>
<td>Install timeout. The default value is 2m0s.</td>
</tr>
<tr>
<td>--kubeconfig (string)</td>
<td>Path to the kubeconfig file to use for CLI requests.</td>
</tr>
<tr>
<td>--namespace (string)</td>
<td>If present, namespace in which to run the CLI request.</td>
</tr>
<tr>
<td>-h, --help</td>
<td>Help output for the <strong>run bundle</strong> subcommand.</td>
</tr>
</tbody>
</table>

**Additional resources**
See Operator group membership for details on possible install modes.

5.11.7.2. bundle-upgrade

The `run bundle-upgrade` subcommand upgrades an Operator that was previously installed in the bundle format with Operator Lifecycle Manager (OLM).

### Table 5.27. `run bundle-upgrade` flags

<table>
<thead>
<tr>
<th>Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>--timeout &lt;duration&gt;</code></td>
<td>Upgrade timeout. The default value is <code>2m0s</code>.</td>
</tr>
<tr>
<td><code>--kubeconfig (string)</code></td>
<td>Path to the <code>kubeconfig</code> file to use for CLI requests.</td>
</tr>
<tr>
<td><code>n. --namespace (string)</code></td>
<td>If present, namespace in which to run the CLI request.</td>
</tr>
<tr>
<td><code>-h. --help</code></td>
<td>Help output for the <code>run bundle</code> subcommand.</td>
</tr>
</tbody>
</table>

5.11.8. scorecard

The `operator-sdk scorecard` command runs the scorecard tool to validate an Operator bundle and provide suggestions for improvements. The command takes one argument, either a bundle image or directory containing manifests and metadata. If the argument holds an image tag, the image must be present remotely.

### Table 5.28. `scorecard` flags

<table>
<thead>
<tr>
<th>Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>-c. --config (string)</code></td>
<td>Path to scorecard configuration file. The default path is <code>bundle/tests/scorecard/config.yaml</code>.</td>
</tr>
<tr>
<td><code>-h. --help</code></td>
<td>Help output for the <code>scorecard</code> command.</td>
</tr>
<tr>
<td><code>--kubeconfig (string)</code></td>
<td>Path to <code>kubeconfig</code> file.</td>
</tr>
<tr>
<td><code>-L. --list</code></td>
<td>List which tests are available to run.</td>
</tr>
<tr>
<td><code>-n. --namespace (string)</code></td>
<td>Namespace in which to run the test images.</td>
</tr>
<tr>
<td><code>-o. --output (string)</code></td>
<td>Output format for results. Available values are <code>text</code>, which is the default, and <code>json</code>.</td>
</tr>
<tr>
<td><code>-l. --selector (string)</code></td>
<td>Label selector to determine which tests are run.</td>
</tr>
<tr>
<td><code>-s. --service-account (string)</code></td>
<td>Service account to use for tests. The default value is <code>default</code>.</td>
</tr>
<tr>
<td>Flag</td>
<td>Description</td>
</tr>
<tr>
<td>-----------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td><code>-x, --skip-cleanup</code></td>
<td>Disable resource cleanup after tests are run.</td>
</tr>
<tr>
<td><code>-w, --wait-time &lt;duration&gt;</code></td>
<td>Seconds to wait for tests to complete, for example <code>35s</code>. The default value is <code>30s</code>.</td>
</tr>
</tbody>
</table>

**Additional resources**

- See [Validating Operators using the scorecard tool](#) for details about running the scorecard tool.
CHAPTER 6. PLATFORM OPERATORS REFERENCE

This reference guide indexes the platform Operators, also known as cluster Operators, shipped by Red Hat that serve as the architectural foundation for OpenShift Container Platform. Platform Operators are installed by default, unless otherwise noted, and are managed by the Cluster Version Operator (CVO). For more details on the control plane architecture, see Operators in OpenShift Container Platform.

Cluster administrators can view platform Operators in the OpenShift Container Platform web console from the Administration → Cluster Settings page.

NOTE

Platform operators are not managed by Operator Lifecycle Manager (OLM) and OperatorHub. OLM and OperatorHub are part of the Operator Framework used in OpenShift Container Platform for installing and running optional add-on Operators.

6.1. CLOUD CREDENTIAL OPERATOR

Purpose
The Cloud Credential Operator (CCO) manages cloud provider credentials as Kubernetes custom resource definitions (CRDs). The CCO syncs on CredentialsRequest custom resources (CRs) to allow OpenShift Container Platform components to request cloud provider credentials with the specific permissions that are required for the cluster to run.

By setting different values for the credentialsMode parameter in the install-config.yaml file, the CCO can be configured to operate in several different modes. If no mode is specified, or the credentialsMode parameter is set to an empty string (""), the CCO operates in its default mode.

Project
openshift-cloud-credential-operator

CRDs

- credentialsrequests.cloudcredential.openshift.io
  - Scope: Namespaced
  - CR: CredentialsRequest
    - Validation: Yes

Configuration objects
No configuration required.

Additional resources

- CredentialsRequest custom resource
- About the Cloud Credential Operator

6.2. CLUSTER AUTHENTICATION OPERATOR

Purpose
The Cluster Authentication Operator installs and maintains the Authentication custom resource in a cluster and can be viewed with:
6.3. CLUSTER AUTOSCALER OPERATOR

Purpose
The Cluster Autoscaler Operator manages deployments of the OpenShift Cluster Autoscaler using the cluster-api provider.

Project
cluster-autoscaler-operator

CRDs
- **ClusterAutoscaler**: This is a singleton resource, which controls the configuration autoscaler instance for the cluster. The Operator only responds to the ClusterAutoscaler resource named default in the managed namespace, the value of the WATCH_NAMESPACE environment variable.

- **MachineAutoscaler**: This resource targets a node group and manages the annotations to enable and configure autoscaling for that group, the min and max size. Currently only MachineSet objects can be targeted.

6.4. CLUSTER CONFIG OPERATOR

Purpose
The Cluster Config Operator performs the following tasks related to config.openshift.io:

- Creates CRDs.
- Renders the initial custom resources.
- Handles migrations.

Project
cluster-config-operator

6.5. CLUSTER CSI SNAPSHOT CONTROLLER OPERATOR

Purpose
The Cluster CSI Snapshot Controller Operator installs and maintains the CSI Snapshot Controller. The CSI Snapshot Controller is responsible for watching the VolumeSnapshot CRD objects and manages the creation and deletion lifecycle of volume snapshots.

Project
cluster-csi-snapshot-controller-operator

6.6. CLUSTER IMAGE REGISTRY OPERATOR

Purpose
The Cluster Image Registry Operator manages a singleton instance of the OpenShift Container Platform registry. It manages all configuration of the registry, including creating storage.

$ oc get clusteroperator authentication -o yaml
On initial start up, the Operator creates a default `image-registry` resource instance based on the configuration detected in the cluster. This indicates what cloud storage type to use based on the cloud provider.

If insufficient information is available to define a complete `image-registry` resource, then an incomplete resource is defined and the Operator updates the resource status with information about what is missing.

The Cluster Image Registry Operator runs in the `openshift-image-registry` namespace and it also manages the registry instance in that location. All configuration and workload resources for the registry reside in that namespace.

Project
`cluster-image-registry-operator`

### 6.7. CLUSTER MACHINE APPROVER OPERATOR

#### Purpose
The Cluster Machine Approver Operator automatically approves the CSRs requested for a new worker node after cluster installation.

**NOTE**
For the control plane node, the `approve-csr` service on the bootstrap node automatically approves all CSRs during the cluster bootstrapping phase.

Project
`cluster-machine-approver-operator`

### 6.8. CLUSTER MONITORING OPERATOR

#### Purpose
The Cluster Monitoring Operator manages and updates the Prometheus-based cluster monitoring stack deployed on top of OpenShift Container Platform.

Project
`openshift-monitoring`

**CRDs**

- `alertmanagers.monitoring.coreos.com`
  - Scope: Namespaced
  - CR: `alertmanager`
  - Validation: Yes

- `prometheuses.monitoring.coreos.com`
  - Scope: Namespaced
  - CR: `prometheus`
  - Validation: Yes
Configuration objects

$ oc -n openshift-monitoring edit cm cluster-monitoring-config

6.9. CLUSTER NETWORK OPERATOR

Purpose
The Cluster Network Operator installs and upgrades the networking components on an OpenShift Container Platform cluster.

6.10. OPENSHIFT CONTROLLER MANAGER OPERATOR

Purpose
The OpenShift Controller Manager Operator installs and maintains the OpenShiftControllerManager custom resource in a cluster and can be viewed with:

$ oc get clusteroperator openshift-controller-manager -o yaml

Project
cluster-openshift-controller-manager-operator

6.11. CLUSTER SAMPLES OPERATOR

Purpose
The Cluster Samples Operator manages the sample image streams and templates stored in the openshift namespace.

On initial start up, the Operator creates the default samples configuration resource to initiate the creation of the image streams and templates. The configuration object is a cluster scoped object with the key cluster and type configs.samples.
The image streams are the Red Hat Enterprise Linux CoreOS (RHCOS)-based OpenShift Container Platform image streams pointing to images on registry.redhat.io. Similarly, the templates are those categorized as OpenShift Container Platform templates.

The Cluster Samples Operator deployment is contained within the openshift-cluster-samples-operator namespace. On start up, the install pull secret is used by the image stream import logic in the internal registry and API server to authenticate with registry.redhat.io. An administrator can create any additional secrets in the openshift namespace if they change the registry used for the sample image streams. If created, those secrets contain the content of a config.json for docker needed to facilitate image import.

The image for the Cluster Samples Operator contains image stream and template definitions for the associated OpenShift Container Platform release. After the Cluster Samples Operator creates a sample, it adds an annotation that denotes the OpenShift Container Platform version that it is compatible with. The Operator uses this annotation to ensure that each sample matches the compatible release version. Samples outside of its inventory are ignored, as are skipped samples.

Modifications to any samples that are managed by the Operator are allowed as long as the version annotation is not modified or deleted. However, on an upgrade, as the version annotation will change, those modifications can get replaced as the sample will be updated with the newer version. The Jenkins images are part of the image payload from the installation and are tagged into the image streams directly.

The samples resource includes a finalizer, which cleans up the following upon its deletion:

- Operator-managed image streams
- Operator-managed templates
- Operator-generated configuration resources
- Cluster status resources

Upon deletion of the samples resource, the Cluster Samples Operator recreates the resource using the default configuration.

Project
cluster-samples-operator

6.12. CLUSTER STORAGE OPERATOR

Purpose

Project
cluster-storage-operator

Configuration
No configuration is required.

Notes
- The Cluster Storage Operator supports Amazon Web Services (AWS) and Red Hat OpenStack Platform (RHOSP).
The created storage class can be made non-default by editing its annotation, but the storage class cannot be deleted as long as the Operator runs.

### 6.13. CLUSTER VERSION OPERATOR

**Purpose**

**Project**  
cluster-version-operator

### 6.14. CONSOLE OPERATOR

**Purpose**

The Console Operator installs and maintains the OpenShift Container Platform web console on a cluster.

**Project**  
console-operator

### 6.15. DNS OPERATOR

**Purpose**

The DNS Operator deploys and manages CoreDNS to provide a name resolution service to pods that enables DNS-based Kubernetes Service discovery in OpenShift Container Platform.

The Operator creates a working default deployment based on the cluster’s configuration.

- The default cluster domain is `cluster.local`.
- Configuration of the CoreDNS Corefile or Kubernetes plug-in is not yet supported.

The DNS Operator manages CoreDNS as a Kubernetes daemon set exposed as a service with a static IP. CoreDNS runs on all nodes in the cluster.

**Project**  
cluster-dns-operator

### 6.16. ETCD CLUSTER OPERATOR

**Purpose**

The etcd cluster Operator automates etcd cluster scaling, enables etcd monitoring and metrics, and simplifies disaster recovery procedures.

**Project**  
cluster-etcd-operator

**CRDs**

- `etcds.operator.openshift.io`
  - Scope: Cluster
  - CR: `etcd`
  - Validation: Yes
6.17. INGRESS OPERATOR

Purpose
The Ingress Operator configures and manages the OpenShift Container Platform router.

Project
openshift-ingress-operator

CRDs

- clusteringresses.ingress.openshift.io
  - Scope: Namespaced
  - CR: clusteringresses
  - Validation: No

Configuration objects

- Cluster config
  - Type Name: clusteringresses.ingress.openshift.io
  - Instance Name: default
  - View Command:

```bash
$ oc get clusteringresses.ingress.openshift.io -n openshift-ingress-operator default -o yaml
```

Notes
The Ingress Operator sets up the router in the openshift-ingress project and creates the deployment for the router:

```bash
$ oc get deployment -n openshift-ingress
```

The Ingress Operator uses the `clusterNetwork[].cidr` from the `network/cluster` status to determine what mode (IPv4, IPv6, or dual stack) the managed ingress controller (router) should operate in. For example, if `clusterNetwork` contains only a v6 `cidr`, then the ingress controller operate in IPv6-only mode.

In the following example, ingress controllers managed by the Ingress Operator will run in IPv4-only mode because only one cluster network exists and the network is an IPv4 `cidr`:

```bash
$ oc get network/cluster -o jsonpath='{.status.clusterNetwork[*]}'
```

Example output

```
map[cidr:10.128.0.0/14 hostPrefix:23]
```
6.18. INSIGHTS OPERATOR

Purpose
The Insights Operator gathers OpenShift Container Platform configuration data and sends it to Red Hat. The data is used to produce proactive insights recommendations about potential issues that a cluster might be exposed to. These insights are communicated to cluster administrators through Insights Advisor on console.redhat.com.

Project
insights-operator

Configuration
No configuration is required.

Notes
Insights Operator compliments OpenShift Container Platform Telemetry.

Additional resources
- See About remote health monitoring for details about Insights Operator and Telemetry.

6.19. KUBERNETES API SERVER OPERATOR

Purpose
The Kubernetes API Server Operator manages and updates the Kubernetes API server deployed on top of OpenShift Container Platform. The Operator is based on the OpenShift library-go framework and it is installed using the Cluster Version Operator (CVO).

Project
openshift-kube-apiserver-operator

CRDs
- kubeapiservers.operator.openshift.io
  - Scope: Cluster
  - CR: kubeapiserver
  - Validation: Yes

Configuration objects

$ oc edit kubeapiserver

6.20. KUBERNETES CONTROLLER MANAGER OPERATOR

Purpose
The Kubernetes Controller Manager Operator manages and updates the Kubernetes Controller Manager deployed on top of OpenShift Container Platform. The Operator is based on OpenShift library-go framework and it is installed via the Cluster Version Operator (CVO).

It contains the following components:

- Operator
Bootstrap manifest renderer

Installer based on static pods

Configuration observer

By default, the Operator exposes Prometheus metrics through the `metrics` service.

**Project**

`cluster-kube-controller-manager-operator`

### 6.21. KUBERNETES SCHEDULER OPERATOR

**Purpose**
The Kubernetes Scheduler Operator manages and updates the Kubernetes Scheduler deployed on top of OpenShift Container Platform. The Operator is based on the OpenShift Container Platform `library-go` framework and it is installed with the Cluster Version Operator (CVO).

The Kubernetes Scheduler Operator contains the following components:

- Operator
- Bootstrap manifest renderer
- Installer based on static pods
- Configuration observer

By default, the Operator exposes Prometheus metrics through the metrics service.

**Project**

`cluster-kube-scheduler-operator`

**Configuration**
The configuration for the Kubernetes Scheduler is the result of merging:

- a default configuration.
- an observed configuration from the spec `.schedulers.config.openshift.io`.

All of these are sparse configurations, invalidated JSON snippets which are merged to form a valid configuration at the end.

### 6.22. KUBERNETES STORAGE VERSION MIGRATOR OPERATOR

**Purpose**
The Kubernetes Storage Version Migrator Operator detects changes of the default storage version, creates migration requests for resource types when the storage version changes, and processes migration requests.

**Project**

`cluster-kube-storage-version-migrator-operator`

### 6.23. MACHINE API OPERATOR
Purpose
The Machine API Operator manages the lifecycle of specific purpose custom resource definitions (CRD), controllers, and RBAC objects that extend the Kubernetes API. This declares the desired state of machines in a cluster.

Project
machine-api-operator

CRDs
- MachineSet
- Machine
- MachineHealthCheck

6.24. MACHINE CONFIG OPERATOR

Purpose
The Machine Config Operator manages and applies configuration and updates of the base operating system and container runtime, including everything between the kernel and kubelet.

There are four components:

- machine-config-server: Provides Ignition configuration to new machines joining the cluster.
- machine-config-controller: Coordinates the upgrade of machines to the desired configurations defined by a MachineConfig object. Options are provided to control the upgrade for sets of machines individually.
- machine-config-daemon: Applies new machine configuration during update. Validates and verifies the state of the machine to the requested machine configuration.
- machine-config: Provides a complete source of machine configuration at installation, first start up, and updates for a machine.

Project
openshift-machine-config-operator

6.25. MARKETPLACE OPERATOR

Purpose
The Marketplace Operator is a conduit to bring off-cluster Operators to your cluster.

Project
operator-marketplace

6.26. NODE TUNING OPERATOR

Purpose
The Node Tuning Operator helps you manage node-level tuning by orchestrating the Tuned daemon. The majority of high-performance applications require some level of kernel tuning. The Node Tuning Operator provides a unified management interface to users of node-level sysctls and more flexibility to add custom tuning specified by user needs.
The Operator manages the containerized Tuned daemon for OpenShift Container Platform as a Kubernetes daemon set. It ensures the custom tuning specification is passed to all containerized Tuned daemons running in the cluster in the format that the daemons understand. The daemons run on all nodes in the cluster, one per node.

Node-level settings applied by the containerized Tuned daemon are rolled back on an event that triggers a profile change or when the containerized Tuned daemon is terminated gracefully by receiving and handling a termination signal.

The Node Tuning Operator is part of a standard OpenShift Container Platform installation in version 4.1 and later.

Project  
cluster-node-tuning-operator

### 6.27. OPENShift API Server Operator

**Purpose**
The OpenShift API Server Operator installs and maintains the `openshift-apiserver` on a cluster.

**Project**
openshift-apiserver-operator

**CRDs**
- `openshifapiservers.operator.openshift.io`
  - Scope: Cluster
  - CR: `openshifapiserver`
  - Validation: Yes

### 6.28. OPERATOR LIFECYCLE MANAGER OPERATORS

**Purpose**
Operator Lifecycle Manager (OLM) helps users install, update, and manage the lifecycle of Kubernetes native applications (Operators) and their associated services running across their OpenShift Container Platform clusters. It is part of the Operator Framework, an open source toolkit designed to manage Operators in an effective, automated, and scalable way.

**Figure 6.1. Operator Lifecycle Manager workflow**
OLM runs by default in OpenShift Container Platform 4.7, which aids cluster administrators in installing, upgrading, and granting access to Operators running on their cluster. The OpenShift Container Platform web console provides management screens for cluster administrators to install Operators, as well as grant specific projects access to use the catalog of Operators available on the cluster.

For developers, a self-service experience allows provisioning and configuring instances of databases, monitoring, and big data services without having to be subject matter experts, because the Operator has that knowledge baked into it.

CRDs
Operator Lifecycle Manager (OLM) is composed of two Operators: the OLM Operator and the Catalog Operator.

Each of these Operators is responsible for managing the custom resource definitions (CRDs) that are the basis for the OLM framework:

Table 6.1. CRDs managed by OLM and Catalog Operators

<table>
<thead>
<tr>
<th>Resource</th>
<th>Short name</th>
<th>Owner</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ClusterServiceVersion (CSV)</td>
<td>csv</td>
<td>OLM</td>
<td>Application metadata: name, version, icon, required resources, installation, and so on.</td>
</tr>
<tr>
<td>InstallPlan</td>
<td>ip</td>
<td>Catalog</td>
<td>Calculated list of resources to be created to automatically install or upgrade a CSV.</td>
</tr>
<tr>
<td>CatalogSource</td>
<td>catsrc</td>
<td>Catalog</td>
<td>A repository of CSVs, CRDs, and packages that define an application.</td>
</tr>
<tr>
<td>Subscription</td>
<td>sub</td>
<td>Catalog</td>
<td>Used to keep CSVs up to date by tracking a channel in a package.</td>
</tr>
<tr>
<td>OperatorGroup</td>
<td>og</td>
<td>OLM</td>
<td>Configures all Operators deployed in the same namespace as the OperatorGroup object to watch for their custom resource (CR) in a list of namespaces or cluster-wide.</td>
</tr>
</tbody>
</table>

Each of these Operators is also responsible for creating the following resources:

Table 6.2. Resources created by OLM and Catalog Operators

<table>
<thead>
<tr>
<th>Resource</th>
<th>Owner</th>
</tr>
</thead>
<tbody>
<tr>
<td>Deployments</td>
<td>OLM</td>
</tr>
<tr>
<td>ServiceAccounts</td>
<td></td>
</tr>
<tr>
<td>(Cluster)Roles</td>
<td></td>
</tr>
</tbody>
</table>
### OLM Operator

The OLM Operator is responsible for deploying applications defined by CSV resources after the required resources specified in the CSV are present in the cluster.

The OLM Operator is not concerned with the creation of the required resources; you can choose to manually create these resources using the CLI or using the Catalog Operator. This separation of concern allows users incremental buy-in in terms of how much of the OLM framework they choose to leverage for their application.

The OLM Operator uses the following workflow:

1. Watch for cluster service versions (CSVs) in a namespace and check that requirements are met.

2. If requirements are met, run the install strategy for the CSV.

   **NOTE**

   A CSV must be an active member of an Operator group for the install strategy to run.

### Catalog Operator

The Catalog Operator is responsible for resolving and installing cluster service versions (CSVs) and the required resources they specify. It is also responsible for watching catalog sources for updates to packages in channels and upgrading them, automatically if desired, to the latest available versions.

To track a package in a channel, you can create a **Subscription** object configuring the desired package, channel, and the **CatalogSource** object you want to use for pulling updates. When updates are found, an appropriate **InstallPlan** object is written into the namespace on behalf of the user.

The Catalog Operator uses the following workflow:

1. Connect to each catalog source in the cluster.

2. Watch for unresolved install plans created by a user, and if found:
   a. Find the CSV matching the name requested and add the CSV as a resolved resource.
   b. For each managed or required CRD, add the CRD as a resolved resource.
   c. For each required CRD, find the CSV that manages it.

3. Watch for resolved install plans and create all of the discovered resources for it, if approved by a user or automatically.

4. Watch for catalog sources and subscriptions and create install plans based on them.
Catalog Registry
The Catalog Registry stores CSVs and CRDs for creation in a cluster and stores metadata about packages and channels.

A package manifest is an entry in the Catalog Registry that associates a package identity with sets of CSVs. Within a package, channels point to a particular CSV. Because CSVs explicitly reference the CSV that they replace, a package manifest provides the Catalog Operator with all of the information that is required to update a CSV to the latest version in a channel, stepping through each intermediate version.

Additional resources
- For more information, see the sections on understanding Operator Lifecycle Manager (OLM).

6.29. OPENSHEET SERVICE CA OPERATOR

Purpose
The OpenShift Service CA Operator mints and manages serving certificates for Kubernetes services.

Project
openshift-service-ca-operator

6.30. VSPHERE PROBLEM DETECTOR OPERATOR

Purpose
The vSphere Problem Detector Operator checks clusters that are deployed on vSphere for common installation and misconfiguration issues that are related to storage.

NOTE
The vSphere Problem Detector Operator is only started by the Cluster Storage Operator when the Cluster Storage Operator detects that the cluster is deployed on vSphere.

Configuration
No configuration is required.

Notes
- The Operator supports OpenShift Container Platform installations on vSphere.
- The Operator uses the vsphere-cloud-credentials to communicate with vSphere.
- The Operator performs checks that are related to storage.

Additional resources
- For more details, see Using the vSphere Problem Detector Operator.