OpenShift Container Platform 4.5

Operators

Working with Operators in OpenShift Container Platform

Last Updated: 2021-07-06

OpenShift Container Platform 4.5 Operators

Working with Operators in OpenShift Container Platform

Legal Notice

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. UNDERSTANDING OPERATORS

1.1. 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, watching over 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.

1.1.1. 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.

1.1.2. 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 4.5.

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.

Operator Metering

Operator Metering collects operational metrics about Operators on the cluster for Day 2 management and aggregating usage metrics.

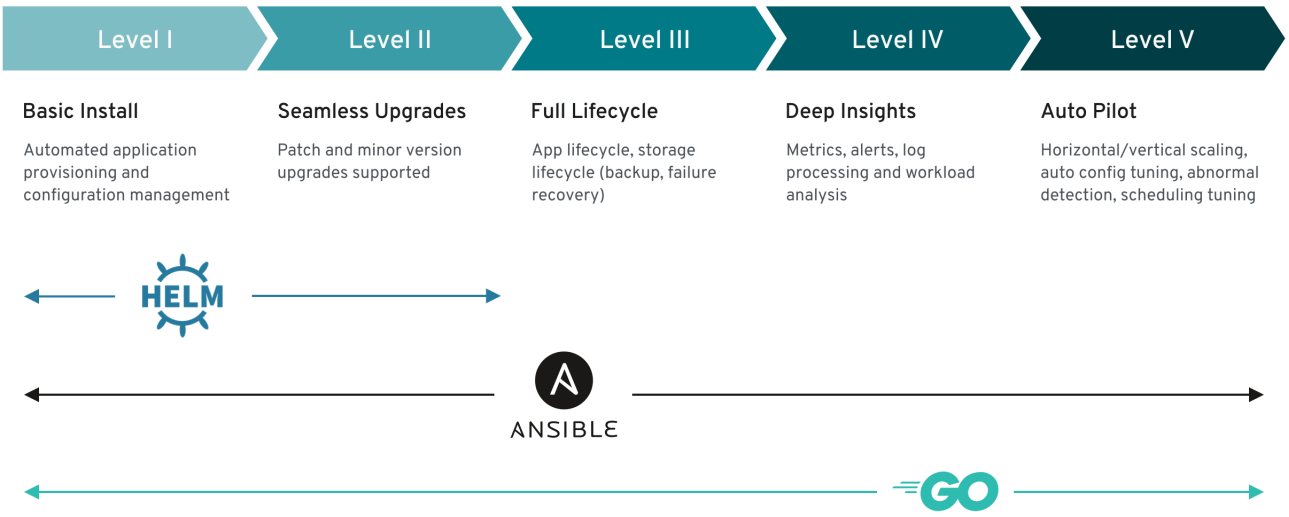
These tools are designed to be composable, so you can use any that are useful to you.

1.1.3. 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 1.1. 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.

1.2. 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.

1.2.1. Common Operator Framework terms

1.2.1.1. 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.

1.2.1.2. 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.

1.2.1.3. Catalog source

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

1.2.1.4. 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.

1.2.1.5. 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.

1.2.1.6. Channel head

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

1.2.1.7. 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.

1.2.1.8. 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.

1.2.1.9. 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.

1.2.1.10. Install plan

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

1.2.1.11. 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.

1.2.1.12. 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.

1.2.1.13. Registry

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

1.2.1.14. Subscription

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

1.2.1.15. 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.

1.3. OPERATOR FRAMEWORK PACKAGING FORMATS

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

1.3.1. 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

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

It also includes a **<name>.package.yaml** file that is the *package manifest* that defines the package name and channels details:

Example package manifest

```
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.

1.3.2. 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.

1.3.2.1. 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

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

Optional objects
The following object types can also be optionally included in the /**manifests** directory of a bundle:

Supported optional objects

- **Secrets**

- **ConfigMaps**

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.

1.3.2.2. 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

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

1. 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.

1.3.2.3. Dependencies

The dependencies of an Operator are listed in a **dependencies.yaml** file inside the bundle's **metadata/** folder. 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**: A package type means this is 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 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
      version: ">0.27.0"
  - type: olm.gvk
    value:
      group: etcd.database.coreos.com
      kind: EtcdCluster
      version: v1beta2
```

1.3.2.4. opm CLI

The new **opm** CLI tool is introduced alongside the new Bundle Format. This tool allows you to create and maintain catalogs of Operators from a list of bundles, called an *index*, that are equivalent to a "repository". 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 via an included API that is served when the container image is run. On OpenShift Container Platform, OLM can use the index image as a catalog by referencing it in a CatalogSource, which polls the image at regular intervals to enable frequent updates to installed Operators on the cluster.

1.4. OPERATOR LIFECYCLE MANAGER (OLM)

1.4.1. Operator Lifecycle Manager concepts

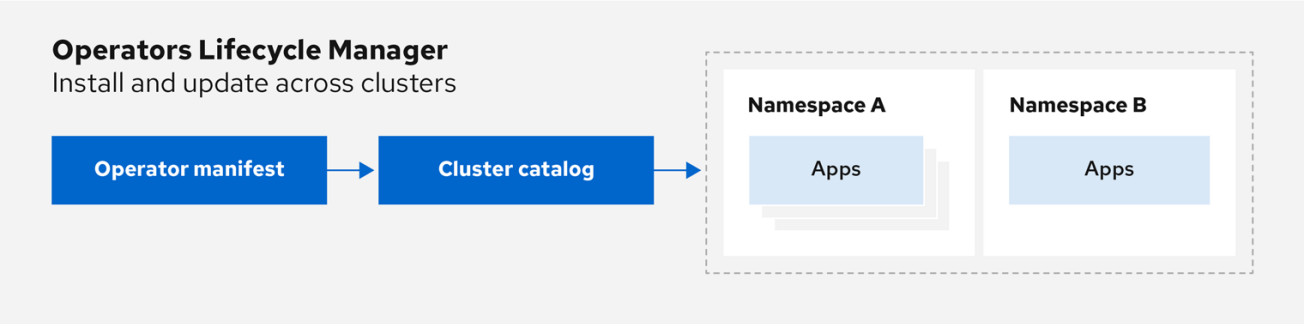
This guide provides an overview of the concepts that drive Operator Lifecycle Manager (OLM) in OpenShift Container Platform.

1.4.1.1. 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 1.2. Operator Lifecycle Manager workflow



OLM runs by default in OpenShift Container Platform 4.5, 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.

1.4.1.2. OLM resources

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

Table 1.1. CRDs managed by OLM and Catalog Operators

Resource	Short name	Description
ClusterServiceVersion (CSV)	**csv**	Application metadata. For example: name, version, icon, required resources.
CatalogSource	**catsrc**	A repository of CSVs, CRDs, and packages that define an application.
Subscription	**sub**	Keeps CSVs up to date by tracking a channel in a package.
InstallPlan	**ip**	Calculated list of resources to be created to automatically install or upgrade a CSV.
OperatorGroup	**og**	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.

1.4.1.2.1. 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.

1.4.1.2.2. Catalog source

A catalog source represents a store of metadata that OLM can query to discover and install Operators and their dependencies. 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.

There are three primary **sourceTypes** for a **CatalogSource** object:

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

The following example defines a catalog source for OperatorHub.io content:

Example CatalogSource object

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

The **name** of the **CatalogSource** object is used as input to a subscription, which instructs OLM where to look to find a requested Operator:

Example Subscription object referencing a catalog source

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

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

1.4.1.2.3. 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

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

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.

1.4.1.2.4. Install plan

An *install plan*, defined by an **InstallPlan** object, describes a set of resources to be created to install or upgrade to a specific version of an Operator, as defined by a cluster service version (CSV).

1.4.1.2.5. 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.

For more information, see the Operator groups guide.

1.4.2. Operator Lifecycle Manager architecture

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

1.4.2.1. 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 1.2. CRDs managed by OLM and Catalog Operators

Resource	Short name	Owner	Description
ClusterServiceVersion (CSV)	**csv**	OLM	Application metadata: name, version, icon, required resources, installation, and so on.
InstallPlan	**ip**	Catalog	Calculated list of resources to be created to automatically install or upgrade a CSV.
CatalogSource	**catsrc**	Catalog	A repository of CSVs, CRDs, and packages that define an application.
Subscription	**sub**	Catalog	Used to keep CSVs up to date by tracking a channel in a package.
OperatorGroup	**og**	OLM	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.

Each of these Operators is also responsible for creating the following resources:

Table 1.3. Resources created by OLM and Catalog Operators

Resource	Owner
Deployments	OLM
ServiceAccounts	
(Cluster)Roles	
(Cluster)RoleBindings	
CustomResourceDefinitions (CRDs)	Catalog
ClusterServiceVersions	

1.4.2.2. 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.

1.4.2.3. 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.

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.

1.4.2.4. 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.

1.4.3. Operator Lifecycle Manager workflow

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

1.4.3.1. 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:

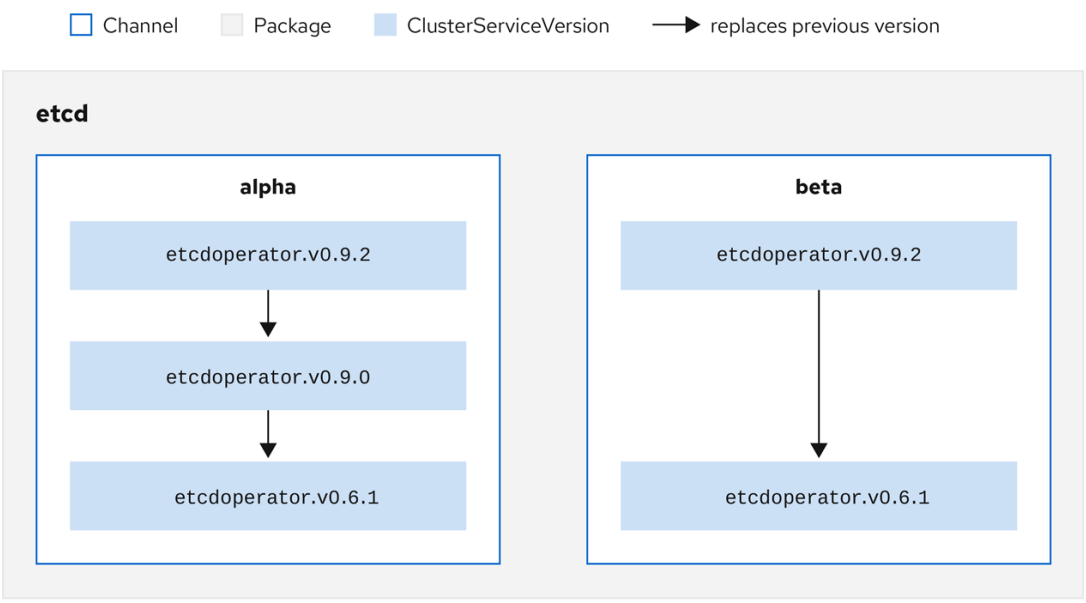
- **ClusterServiceVersion** (CSV)
- **CatalogSource**
- **Subscription**

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.

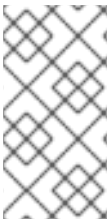
Figure 1.3. Catalog source overview

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 1.4. 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.

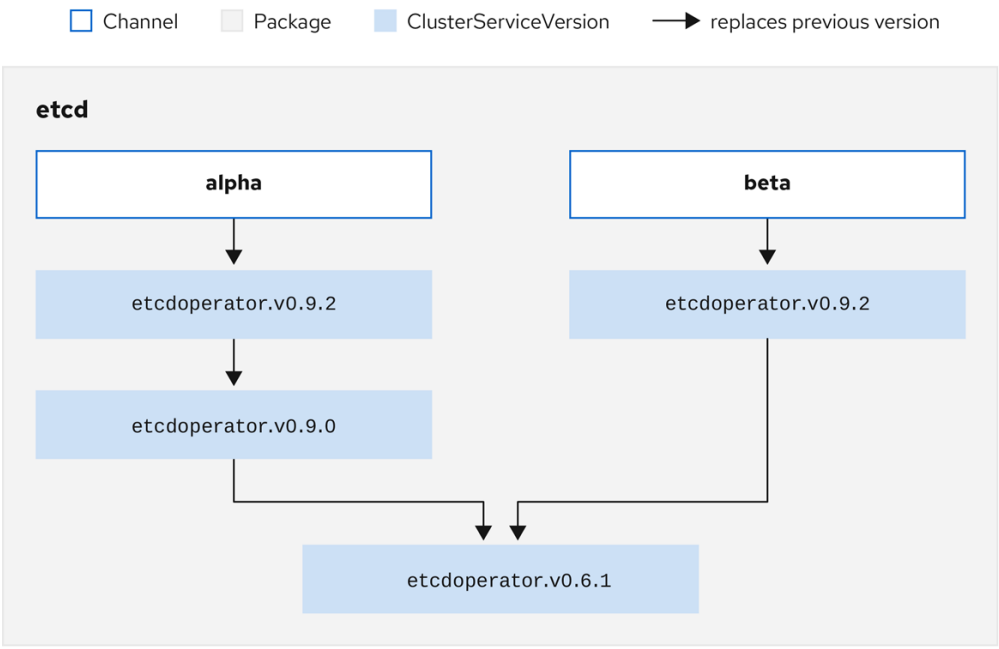


NOTE

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:

Figure 1.5. OLM graph of available channel updates



Example channels in a package

```
packageName: example
channels:
- 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.

1.4.3.1.1. 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.

1.4.3.1.2. 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.

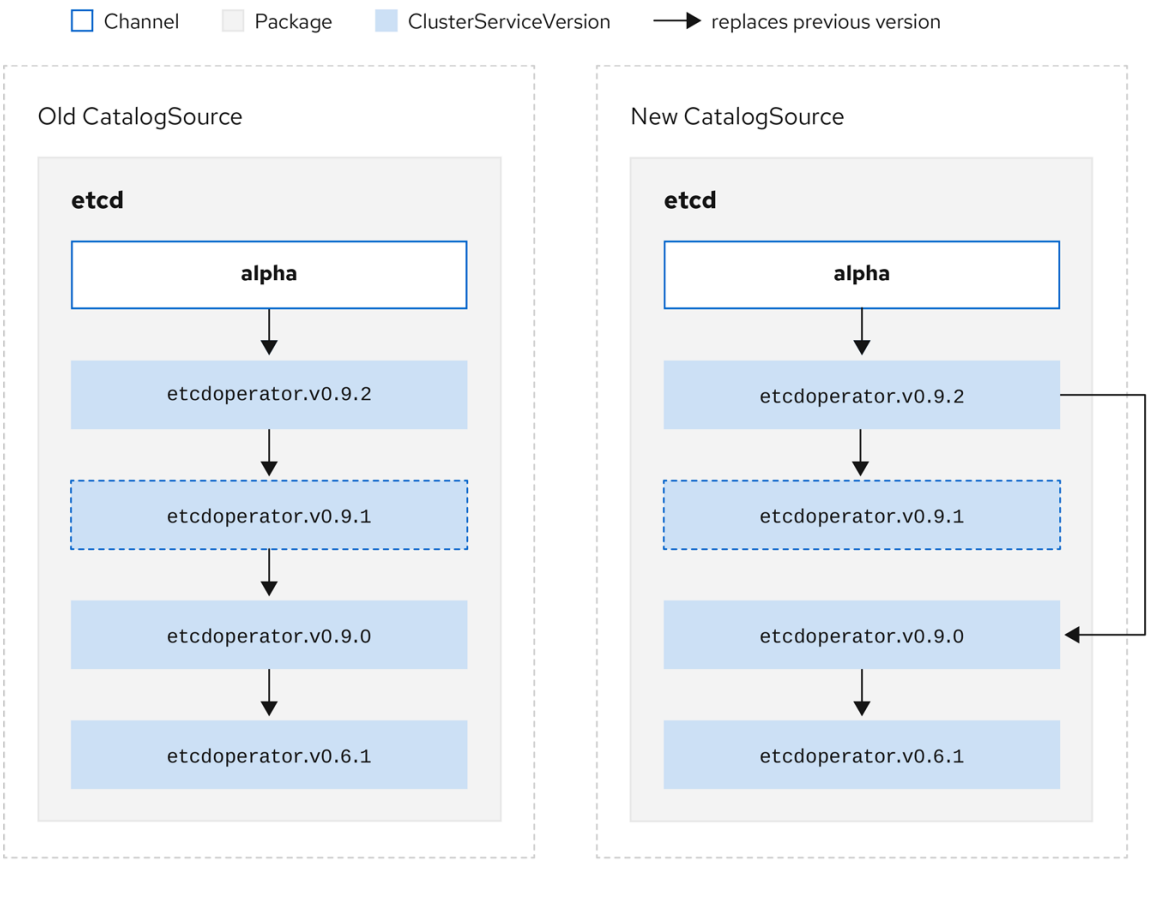
By shipping a new catalog and adding a *skipped* release, OLM is ensured that it can always get a single unique update regardless of the cluster state and whether it has seen the bad update yet.

Example CSV with skipped release

```
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**.

Figure 1.6. Skipping updates



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.

1.4.3.1.3. 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:

```
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

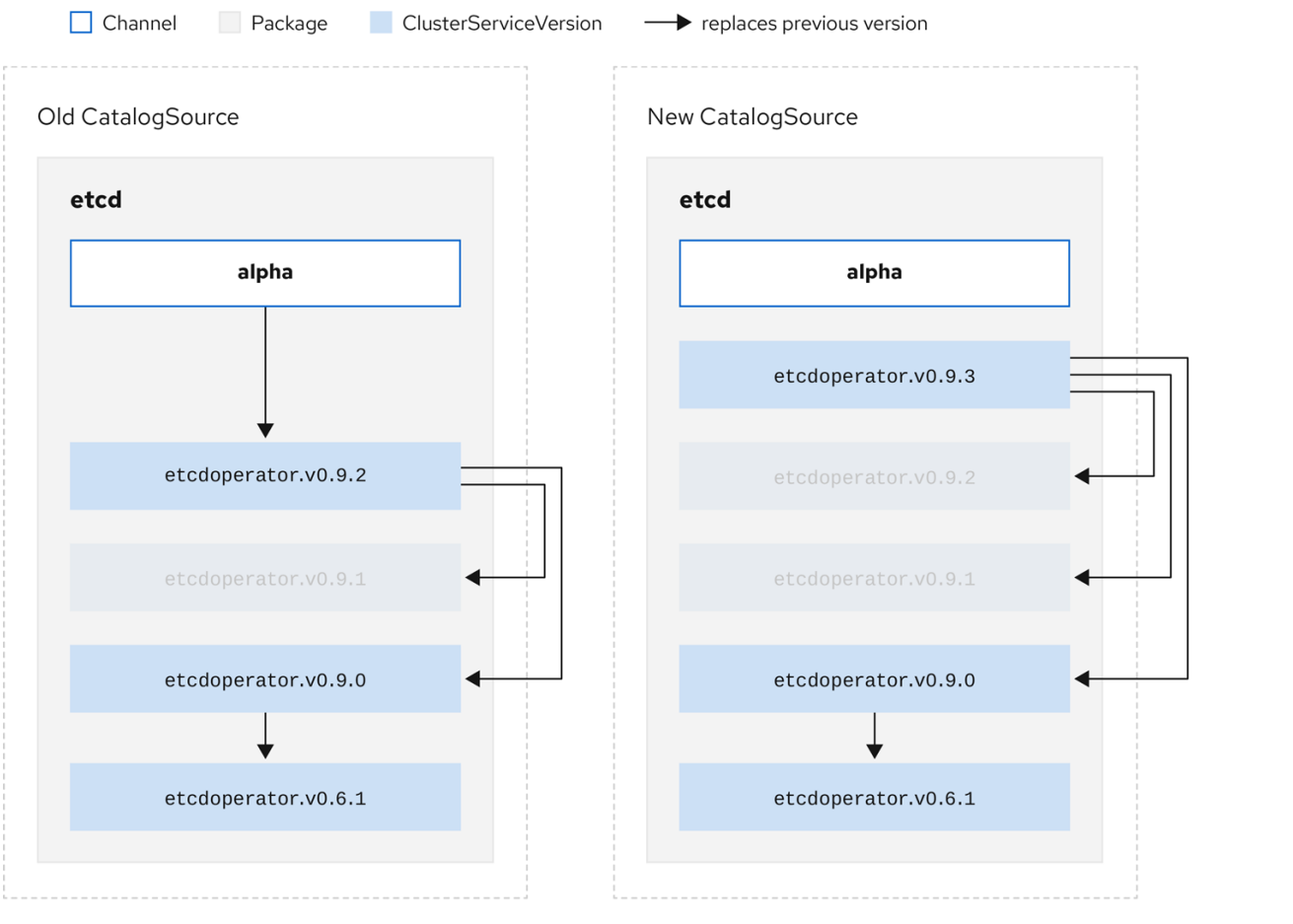
```
apiVersion: operators.coreos.com/v1alpha1
kind: ClusterServiceVersion
metadata:
  name: elasticsearch-operator.v4.1.2
  namespace: <namespace>
  annotations:
    olm.skipRange: '>=4.1.0 <4.1.2'
```

1.4.3.1.4. Z-stream 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 1.7. Replacing several Operators



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.

1.4.4. 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.

1.4.4.1. 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:

- install a set of Operators that require APIs that cannot be provided, or
- update an Operator in a way that breaks another that depends upon it.

1.4.4.2. 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.

1.4.4.2.1. Adding a new CRD version

Procedure

To add a new version of a CRD:

1. Add a new entry in the CRD resource under the **versions** section.
   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**:

```
versions:
  - name: v1alpha1
    served: true
    storage: false
```

```
  - name: v1beta1 (1)
    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:

```
customresourcedefinitions:
  owned:
  - name: cluster.example.com
    version: v1beta1 (1)
    kind: cluster
    displayName: Cluster
```

(1) Update the **version**.

3. Push the updated CRD and CSV to your bundle.

1.4.4.2.2. 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:

```
versions:
  - name: v1alpha1
    served: false (1)
    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:

```
versions:
  - name: v1alpha1
    served: false
    storage: false (1)
  - name: v1beta1
    served: true
    storage: true (2)
```

1 2 Update the **storage** fields accordingly.

NOTE

In order 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:

```
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.

1.4.4.3. 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.

1.4.5. Operator groups

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

1.4.5.1. 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.

1.4.5.2. Operator group membership

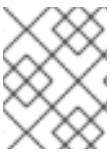
An Operator is considered a *member* of an Operator group if the following conditions are true:

- 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 1.4. Install modes and supported Operator groups

InstallModeType	Description
OwnNamespace	The Operator can be a member of an Operator group that selects its own namespace.
SingleNamespace	The Operator can be a member of an Operator group that selects one namespace.
MultiNamespace	The Operator can be a member of an Operator group that selects more than one namespace.
AllNamespaces	The Operator can be a member of an Operator group that selects all namespaces (target namespace set is the empty string **""**).



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.

1.4.5.3. Target namespace selection

You can explicitly name the target namespace for an Operator group using the **spec.targetNamespaces** parameter:

```
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:

```
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:

```
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 Opeator 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.

1.4.5.4. Operator group CSV annotations

Member CSVs of an Operator group have the following annotations:

Annotation	Description
olm.operatorGroup=<group_name>	Contains the name of the Operator group.

Annotation	Description
olm.operatorNamespace=<group_namespace>	Contains the namespace of the Operator group.
olm.targetNamespaces=<target_namespaces>	Contains a comma-delimited string that lists the target namespace selection of the Operator group.

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.

1.4.5.5. 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:

```
apiVersion: operators.coreos.com/v1
kind: OperatorGroup
metadata:
  annotations:
    olm.providedAPIs: PackageManifest.v1alpha1.packages.apps.redhat.com
  name: olm-operators
  namespace: local
  ...
spec:
  selector: {}
  serviceAccount:
    metadata:
      creationTimestamp: null
  targetNamespaces:
  - local
status:
  lastUpdated: 2019-02-19T16:18:28Z
  namespaces:
  - local
```

1.4.5.6. 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:

Cluster role	Label to match
<operatorgroup_name>-admin	**olm.opgroup.permissions/aggregate-to-admin: <operatorgroup_name>**
<operatorgroup_name>-edit	**olm.opgroup.permissions/aggregate-to-edit: <operatorgroup_name>**
<operatorgroup_name>-view	**olm.opgroup.permissions/aggregate-to-view: <operatorgroup_name>**

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 1.5. Cluster roles generated for each API resource from a CRD

Cluster role	Settings
<kind>.<group>-<version>-admin	Verbs on **<kind>**: • ***** Aggregation labels: • **rbac.authorization.k8s.io/aggregate-to-admin: true** • **olm.opgroup.permissions/aggregate-to-admin: <operatorgroup_name>**

Cluster role	Settings
<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 1.6. Cluster roles generated for each API resource from an API service

Cluster role	Settings

Cluster role	Settings
<kind>.<group>-<version>-admin	Verbs on **<kind>**: • ***** Aggregation labels: • **rbac.authorization.k8s.io/aggregate-to-admin: true** • **olm.opgroup.permissions/aggregate-to-admin: <operatorgroup_name>**
<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>**

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.

1.4.5.7. 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.

1.4.5.8. 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:

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

1.4.5.9. Operator 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.

1.4.5.10. 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.

1.4.6. Operator Lifecycle Manager metrics

1.4.6.1. Exposed metrics

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

Table 1.7. Metrics exposed by OLM

Name	Description
catalog_source_count	Number of catalog sources.
csv_abnormal	When reconciling a cluster service version (CSV), present whenever a CSV version is in any state other than **Succeeded**, for example when it is not installed. Includes the **name**, **namespace**, **phase**, **reason**, and **version** labels. A Prometheus alert is created when this metric is present.
csv_count	Number of CSVs successfully registered.
csv_succeeded	When reconciling a CSV, represents whether a CSV version is in a **Succeeded** state (value **1**) or not (value **0**). Includes the **name**, **namespace**, and **version** labels.
csv_upgrade_count	Monotonic count of CSV upgrades.
install_plan_count	Number of install plans.

Name	Description
subscription_count	Number of subscriptions.
subscription_sync_total	Monotonic count of subscription syncs. Includes the **channel**, **installed** CSV, and subscription **name** labels.

1.5. UNDERSTANDING OPERATORHUB

1.5.1. 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 Operator sources grouped into the following categories:

Category	Description
Red Hat Operators	Red Hat products packaged and shipped by Red Hat. Supported by Red Hat.
Certified Operators	Products from leading independent software vendors (ISVs). Red Hat partners with ISVs to package and ship. Supported by the ISV.
Community Operators	Optionally-visible software maintained by relevant representatives in the operator-framework/community-operators GitHub repository. No official support.
Custom Operators	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.

NOTE

OperatorHub content automatically refreshes every 60 minutes.

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 by Red Hat's ISV partner portal at connect.redhat.com.

1.5.2. OperatorHub architecture

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

The Marketplace Operator manages **OperatorHub** and **OperatorSource** custom resource definitions (CRDs).

NOTE

Although some **OperatorSource** object information is exposed through the OperatorHub UI, it is only used directly by those who are creating their own Operators.

1.5.2.1. OperatorHub CRD

You can use the **OperatorHub** CRD to change the state of the default **OperatorSource** objects provided with OperatorHub on the cluster between enabled and disabled. This capability is useful when configuring OpenShift Container Platform in restricted network environments.

Example OperatorHub custom resource

```
apiVersion: config.openshift.io/v1
kind: OperatorHub
metadata:
  name: cluster
spec:
  disableAllDefaultSources: true (1)
  sources: [ (2)
    {
      name: "community-operators",
      disabled: false
    }
  ]
```

1. **disableAllDefaultSources** is an override that controls availability of all default **OperatorSource** objects that are configured by default during an OpenShift Container Platform installation.

2. Disable default the default sources individually by changing the **disabled** parameter value per source.

1.5.2.2. OperatorSource CRD

For each Operator, the **OperatorSource** CRD is used to define the external data store used to store Operator bundles.

Example OperatorSource custom resource

```
apiVersion: operators.coreos.com/v1
kind: OperatorSource
metadata:
  name: community-operators
  namespace: marketplace
spec:
```

```
type: appregistry (1)
endpoint: https://quay.io/cnr (2)
registryNamespace: community-operators (3)
displayName: "Community Operators" (4)
publisher: "Red Hat" (5)
```

1. To identify the data store as an application registry, **type** is set to **appregistry**.
2. Currently, Quay is the external data store used by OperatorHub, so the endpoint is set to **https://quay.io/cnr** for the Quay.io **appregistry**.
3. For a Community Operator, **registryNamespace** is set to **community-operator**.
4. Optionally, set **displayName** to a name that appears for the Operator in the OperatorHub UI.
5. Optionally, set **publisher** to the person or organization publishing the Operator that appears in the OperatorHub UI.

1.5.3. Additional resources

- Catalog source
- Getting started with the Operator SDK
- Generating a ClusterServiceVersion (CSV)
- Operator installation and upgrade workflow in OLM
- Red Hat Partner Connect
- Red Hat Marketplace

1.6. CRDS

1.6.1. Extending the Kubernetes API with custom resource definitions

This guide describes how cluster administrators can extend their OpenShift Container Platform cluster by creating and managing custom resource definitions (CRDs).

1.6.1.1. 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 CRD 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.

1.6.1.2. 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

```
apiVersion: apiextensions.k8s.io/v1beta1 (1)
kind: CustomResourceDefinition
metadata:
  name: crontabs.stable.example.com (2)
spec:
  group: stable.example.com (3)
  version: v1 (4)
  scope: Namespaced (5)
  names:
    plural: crontabs (6)
    singular: crontab (7)
    kind: CronTab (8)
    shortNames:
    - ct (9)
```

(1) Use the **apiextensions.k8s.io/v1beta1** API.

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

(3) 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.

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

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

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

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

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

9. 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>/...
```

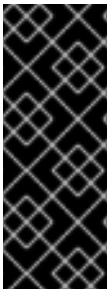
For example, using the example file, the following endpoint is created:

```
/apis/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.

1.6.1.3. 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

```
kind: ClusterRole
apiVersion: rbac.authorization.k8s.io/v1 (1)
metadata:
  name: aggregate-cron-tabs-admin-edit (2)
  labels:
    rbac.authorization.k8s.io/aggregate-to-admin: "true" (3)
    rbac.authorization.k8s.io/aggregate-to-edit: "true" (4)
rules:
- apiGroups: ["stable.example.com"] (5)
  resources: ["crontabs"] (6)
  verbs: ["get", "list", "watch", "create", "update", "patch", "delete", "deletecollection"] (7)
---
kind: ClusterRole
apiVersion: rbac.authorization.k8s.io/v1
metadata:
  name: aggregate-cron-tabs-view (8)
  labels:
    # Add these permissions to the "view" default role.
    rbac.authorization.k8s.io/aggregate-to-view: "true" (9)
    rbac.authorization.k8s.io/aggregate-to-cluster-reader: "true" (10)
rules:
- apiGroups: ["stable.example.com"] (11)
  resources: ["crontabs"] (12)
  verbs: ["get", "list", "watch"] (13)
```

(1) Use the **rbac.authorization.k8s.io/v1** API.

(2) (8) Specify a name for the definition.

(3) Specify this label to grant permissions to the admin default role.

(4) Specify this label to grant permissions to the edit default role.

(5) (11) Specify the group name of the CRD.

(6) (12) Specify the plural name of the CRD that these rules apply to.

(7) (13) 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.

(9) Specify this label to grant permissions to the **view** default role.

(10) Specify this label to grant permissions to the **cluster-reader** default role.

2. Create the cluster role:

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

1.6.1.4. 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

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

1. 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:

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

1.6.1.5. 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

```
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

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

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

1 2 Custom data from the YAML that you used to create the object displays.

1.6.2. Managing resources from custom resource definitions

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

1.6.2.1. 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.

1.6.2.2. 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

```
apiVersion: "stable.example.com/v1" 1
kind: CronTab 2
metadata:
  name: my-new-cron-object 3
```

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

1. 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:

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

1.6.2.3. 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

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

1 2 Custom data from the YAML that you used to create the object displays.

CHAPTER 2. USER TASKS

2.1. 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.

2.1.1. 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 4.5 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 **Copied**, and then 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 New**.
   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 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.

2.2. 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.

2.2.1. 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.

2.2.2. 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

2.2.3. Installing from OperatorHub using the web console

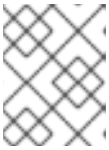
You can install and subscribe to an Operator from OperatorHub using the OpenShift Container Platform web console.

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.

2.2.4. 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

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

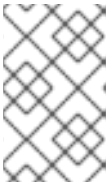
Note the catalog for your desired Operator.

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

```
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

```
apiVersion: operators.coreos.com/v1alpha1
kind: Subscription
metadata:
  name: <subscription_name>
  namespace: openshift-operators (1)
spec:
  channel: <channel_name> (2)
  name: <operator_name> (3)
  source: redhat-operators (4)
  sourceNamespace: openshift-marketplace (5)
```

   (1) 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.

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

2.2.5. 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

```
apiVersion: operators.coreos.com/v1alpha1
kind: Subscription
metadata:
  name: quay-operator
  namespace: quay
spec:
  channel: quay-v3.4
  installPlanApproval: Manual (1)
  name: quay-operator
```

```
source: redhat-operators
sourceNamespace: openshift-marketplace
startingCSV: quay-operator.v3.4.0 2
```

1 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:

```
$ 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

2.3. MANAGING ADMISSION WEBHOOKS IN OPERATOR LIFECYCLE MANAGER

Validating and mutating admission 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.

2.3.1. Defining webhooks in a CSV

The **ClusterServiceVersion** (CSV) resource includes a **webhookdefinitions** section to define validating and mutating admission webhooks that ship with an Operator. For example:

CSV containing a validating admission webhook

```
apiVersion: operators.coreos.com/v1alpha1
kind: ClusterServiceVersion
metadata:
  annotations:
    description: |-
      An example CSV that contains a webhook
  name: example-webhook.v1.0.0
  namespace: placeholder
spec:
  webhookdefinitions:
  - generateName: example.webhook.com
    type: ValidatingAdmissionWebhook
    deploymentName: "example-webhook-deployment"
    containerPort: 443
    sideEffects: "None"
    failurePolicy: "Ignore"
    admissionReviewVersions:
    - "v1"
```

```
    - "v1beta1"
    rules:
    - operations:
      - "CREATE"
      apiGroups:
      - ""
      apiVersions:
      - "v1"
      resources:
      - "configmaps"
    objectSelector:
      foo: bar
    webhookPath: "/validate"
...
```

Operator Lifecycle Manager (OLM) requires that you define the following:

- The **type** field must be set to either **ValidatingAdmissionWebhook** or **MutatingAdmissionWebhook**, 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.

2.3.2. Webhook considerations

When developing an admission webhook to be managed by Operator Lifecycle Manager (OLM), consider the following constraints:

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

2.3.3. Additional resources

- Types of webhook admission plug-ins

CHAPTER 3. ADMINISTRATOR TASKS

3.1. ADDING OPERATORS TO A CLUSTER

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

3.1.1. 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

3.1.2. 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

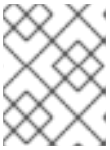
- Access to an OpenShift Container Platform cluster using an account with **cluster-admin** permissions.
- Access to an OpenShift Container Platform cluster using an account with Operator installation permissions.

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.

3.1.3. 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

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

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

```
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

```
apiVersion: operators.coreos.com/v1alpha1
kind: Subscription
metadata:
  name: <subscription_name>
  namespace: openshift-operators (1)
spec:
  channel: <channel_name> (2)
  name: <operator_name> (3)
  source: redhat-operators (4)
  sourceNamespace: openshift-marketplace (5)
```

(1) 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.

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

3.1.4. 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

   ```
   apiVersion: operators.coreos.com/v1alpha1
   kind: Subscription
   metadata:
     name: quay-operator
     namespace: quay
   spec:
     channel: quay-v3.4
     installPlanApproval: Manual (1)
     name: quay-operator
     source: redhat-operators
     sourceNamespace: openshift-marketplace
     startingCSV: quay-operator.v3.4.0 (2)
   ```

   (1) 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:

```
$ 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

3.2. 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.

3.2.1. 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 should follow a common convention within a given Operator. Some naming schemes include:

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

Alternatively, the channel names might follow the version numbers of the application provided by the Operator.

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.

3.2.2. 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**.

3.3. 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.

3.3.1. 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.

   The Operator, any Operator deployments, and pods are removed by this action. Any 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.

3.3.2. 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
```

3.4. 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)

3.4.1. 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**

   For example:

   Subscription object with proxy setting overrides

```
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-
operator@sha256:66a37fd61a06a43969854ee6d3e21088a98b93838e284a6086b13917f96b0
d9c
...
```

3.4.2. 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:

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

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 the **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:

```
kind: Subscription
metadata:
  name: my-operator
spec:
  package: etcd
  channel: alpha
  config: 1
    selector:
      matchLabels:
        <labels_for_pods> 2
    volumes: 3
    - name: trusted-ca
      configMap:
        name: trusted-ca
        items:
          - key: ca-bundle.crt 4
            path: tls-ca-bundle.pem 5
    volumeMounts: 6
```

```
- name: trusted-ca
  mountPath: /etc/pki/ca-trust/extracted/pem
  readOnly: true
```

1. 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.

3.5. 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.

3.5.1. Operator subscription condition types

Subscriptions can report the following condition types:

Table 3.1. Subscription condition types

Condition	Description
CatalogSourcesUnhealthy	Some or all of the catalog sources to be used in resolution are unhealthy.
InstallPlanMissing	An install plan for a subscription is missing.
InstallPlanPending	An install plan for a subscription is pending installation.
InstallPlanFailed	An install plan for a subscription has failed.

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.

3.5.2. Viewing Operator subscription status using the CLI

You can view Operator subscription status 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>
```

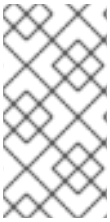
2. Use the **oc describe** command to inspect a **Subscription** resource:

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



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.

3.6. 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.

3.6.1. 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.

3.6.1.1. 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.

3.6.1.2. 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.

3.6.2. 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
  name: scoped
  namespace: scoped
EOF
```

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:

```
$ 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:

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

1 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.

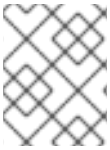
Any Operator tied to this Operator group is confined to the permissions granted to the specified service account. If the Operator requests permissions that are outside the scope of the service account, the installation fails with relevant errors.

3.6.2.1. 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**
- **Subscription**
- **Secret**
- **ServiceAccount**
- **Service**
- **ClusterRole** and **ClusterRoleBinding**
- **Role** and **RoleBinding**

In order 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.

```
kind: Role
rules:
- apiGroups: ["operators.coreos.com"]
  resources: ["subscriptions", "clusterserviceversions"]
  verbs: ["get", "create", "update", "patch"]
- apiGroups: [""]
  resources: ["services", "serviceaccounts"]
  verbs: ["get", "create", "update", "patch"]
- apiGroups: ["rbac.authorization.k8s.io"]
  resources: ["roles", "rolebindings"]
  verbs: ["get", "create", "update", "patch"]
- apiGroups: ["apps"] 1
  resources: ["deployments"]
  verbs: ["list", "watch", "get", "create", "update", "patch", "delete"]
- apiGroups: [""] 2
  resources: ["pods"]
  verbs: ["list", "watch", "get", "create", "update", "patch", "delete"]
```

1 2 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:

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

1 Required to get the secret from the OLM namespace.

3.6.3. 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:

```
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:

```
apiVersion: operators.coreos.com/v1
kind: InstallPlan
status:
  conditions:
  - lastTransitionTime: "2019-07-26T21:13:10Z"
    lastUpdateTime: "2019-07-26T21:13:10Z"
    message: 'error creating clusterrole etcdoperator.v0.9.4-clusterwide-dsfx4:
clusterroles.rbac.authorization.k8s.io
      is forbidden: User "system:serviceaccount:scoped:scoped" cannot create resource
      "clusterroles" in API group "rbac.authorization.k8s.io" at the cluster scope'
```

```
    reason: InstallComponentFailed
    status: "False"
    type: Installed
  phase: Failed
```

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.

3.7. MANAGING CUSTOM CATALOGS

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

3.7.1. Custom catalogs using Package Manifest Format

3.7.1.1. Understanding Operator catalog images

Operator Lifecycle Manager (OLM) always installs Operators from the latest version of an Operator catalog. For OpenShift Container Platform 4.5, Red Hat-provided Operators are distributed via Quay App Registry catalogs from quay.io.

Table 3.2. Red Hat-provided App Registry catalogs

Catalog	Description
redhat-operators	Public catalog for Red Hat products packaged and shipped by Red Hat. Supported by Red Hat.
certified-operators	Public catalog for products from leading independent software vendors (ISVs). Red Hat partners with ISVs to package and ship. Supported by the ISV.
community-operators	Public catalog for software maintained by relevant representatives in the operator-framework/community-operators GitHub repository. No official support.

As catalogs are updated, the latest versions of Operators change, and older versions may be removed or altered. This behavior can cause problems maintaining reproducible installs over time. In addition, when OLM runs on an OpenShift Container Platform cluster in a restricted network environment, it is unable to access the catalogs from quay.io directly.

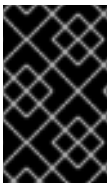
Using the **oc adm catalog build** command, cluster administrators can create an *Operator catalog image*. An Operator catalog image is:

- a point-in-time export of an App Registry type catalog's content.
- the result of converting an App Registry catalog to a container image type catalog.
- an immutable artifact.

Creating an Operator catalog image provides a simple way to use this content without incurring the aforementioned issues.

3.7.1.2. Building an Operator 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 1.4.4+
- 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:

```
$ 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:

```
$ 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>
```

   Also authenticate with **registry.redhat.io** so that the base image can be pulled during the build:

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

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

   1. Organization (namespace) to pull from an App Registry instance.
   2. Set **--from** to the **ose-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, **v1**.
   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: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/9cskxqs53ll3wdn434vw4cd80000gn/T/300666084/manifests-829192605
file=4.2 load=package
W1114 19:42:37.876180   34665 builder.go:141] error building database: error loading
package into db: fuse-camel-k-operator.v7.5.0 specifies replacement that couldn't be found
Uploading ... 244.9kB/s
```

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

3.7.1.3. Mirroring an Operator catalog image

Cluster administrators can mirror their catalog's content into a registry and use a CatalogSource to load the content onto an OpenShift Container Platform 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 pushed to a supported registry
- **oc** version 4.3.5+
- **podman** version 1.4.4+
- 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:

```
$ 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 \ 1
    <registry_host_name>:<port> \
    [-a ${REG_CREDS}] \ 2
    [--insecure] \ 3
    --filter-by-os='.*' \ 4
    [--manifests-only] 5
```

1. Specify your Operator catalog image.
2. Optional: If required, specify the location of your registry credentials file.
3. Optional: If you do not want to configure trust for the target registry, add the **--insecure** flag.
4. This flag is currently required due to a known issue with multiple architecture support.
5. Optional: Only generate the manifests required for mirroring and do not actually mirror the image content to a registry.

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. For more information, see BZ#1890951.

Example output

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

1 Temporary database generated by the command.

After running the command, a **<image_name>-manifests/** directory is created in the current directory and generates the following files:

- 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.
- 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**:

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

1 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/openshift4/ose-logging-
kibana5@sha256:aa4a8b2a00836d0e28aa6497ad90a3c116f135f382d8211e3c55f34f
b36dfe61
operatorbundle_name = clusterlogging.4.3.33-202008111029.p0

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

      ii. 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

```
registry.redhat.io/openshift4/ose-logging-
```

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

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:

```
$ oc image mirror \
  [-a ${REG_CREDS}] \
  --filter-by-os='.*' \
  -f ./redhat-operators-manifests/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. Apply the **ImageContentSourcePolicy** object:

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

4. Create a **CatalogSource** object that references your catalog image.

a. Modify the following to your specifications and save it as a **catalogsource.yaml** file:

```
apiVersion: operators.coreos.com/v1alpha1
kind: CatalogSource
metadata:
  name: my-operator-catalog
  namespace: openshift-marketplace
spec:
  sourceType: grpc
  image: <registry_host_name>:<port>/olm/redhat-operators:v1 (1)
  displayName: My Operator Catalog
  publisher: grpc
```

(1) Specify your custom Operator catalog image.

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

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

5. Verify the following resources are created successfully.

   a. Check the pods:

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

   Example output

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

   b. Check the catalog source:

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

   Example output

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

   c. Check the package manifest:

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

   Example output

   ```
   NAME    CATALOG             AGE
   etcd    My Operator Catalog  34s
   ```

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

Additional resources

- Architecture and operating system support for Operators

3.7.1.4. Updating an Operator 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 1.4.4+
- Access to mirror registry that supports Docker v2-2
- 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
```

- 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>
```

   Also authenticate with **registry.redhat.io** so that the base image can be pulled during the build:

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

2. 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:v4.5 \ 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 **ose-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:f73d42950021f9240389f99ddc5b0c7f1b533c054ba344654ff1edaf6bf827e3
to example_registry:5000/olm/redhat-operators:v2
```

3. 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> \
  [-a ${REG_CREDS}] \ 2
  [--insecure] \ 3
  --filter-by-os='.*' 4
```

1. Specify your new Operator catalog image.
2. Optional: If required, specify the location of your registry credentials file.
3. Optional: If you do not want to configure trust for the target registry, add the **--insecure** flag.
4. This flag is currently required due to a known issue with multiple architecture support. 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. For more information, see BZ#1890951.

4. Apply the newly generated manifests:

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

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.

5. 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 1
  displayName: My Operator Catalog
  publisher: grpc
```

1 Specify your new Operator catalog image.

      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

3.7.1.5. Testing an Operator 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 an OLM subscription by referencing the image in a

CatalogSource object. For this example, the procedure uses a custom **redhat-operators** catalog image previously built and pushed to a supported registry.

Prerequisites

- A custom Operator catalog image pushed to a supported registry
- **podman** version 1.4.4+
- **oc** version 4.3.5+
- Access to mirror registry that supports Docker v2-2
- **grpcurl**

Procedure

1. Pull the Operator catalog image:

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

2. Run the image:

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

3. Query the running image for available packages using **grpcurl**:

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

Example output

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

4. Get the latest Operator bundle in a channel:

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

Example output

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

```
  "channelName": "stable",
...
```

5. Get the digest of the image:

```
$ 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:

```
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:

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

3.7.2. Custom catalogs using Bundle Format

3.7.2.1. opm CLI

The new **opm** CLI tool is introduced alongside the new Bundle Format. This tool allows you to create and maintain catalogs of Operators from a list of bundles, called an *index*, that are equivalent to a "repository". 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 via an

included API that is served when the container image is run. On OpenShift Container Platform, OLM can use the index image as a catalog by referencing it in a CatalogSource, which polls the image at regular intervals to enable frequent updates to installed Operators on the cluster.

Additional resources

- To create a bundle image using the Operator SDK, see Working with bundle images.

3.7.2.2. Installing opm

You can install the **opm** CLI tool on your workstation.

Prerequisites

- **podman** version 1.4.4+

Procedure

1. 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
```

2. Authenticate with **registry.redhat.io**:

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

3. Extract the **opm** binary from the Operator Registry image and copy it to your local file system:

```
$ oc image extract registry.redhat.io/openshift4/ose-operator-registry:v4.5 \
    -a ${REG_CREDS} \ 1
    --path /usr/bin/opm:. \
    --confirm
```

1 Specify the location of your registry credentials file.

4. Make the binary executable:

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

5. Place the file anywhere in your **PATH**, such as **/usr/local/bin/**.

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

6. Verify that the client was installed correctly:

```
$ opm version
```

Example output

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

3.7.2.3. Creating an index image

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

Prerequisites

- **opm** version 1.12.3+
- **podman** version 1.4.4+
- A bundle image built and pushed to a registry.

Procedure

1. Start a new index:

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

1. 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:

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

3.7.2.4. Creating a catalog from an index image

You can create a catalog from an index image and apply it to an OpenShift Container Platform cluster.

Prerequisites

- An index image built and pushed to a registry.

Procedure

1. Apply a **CatalogSource** object to your cluster that references your index image:

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

apiVersion: operators.coreos.com/v1alpha1
kind: CatalogSource
metadata:
  name: test-catalog
```

```
  namespace: openshift-marketplace
spec:
  sourceType: grpc
  image: quay.io/<namespace>/test-catalog:latest (1)
  displayName: Test Catalog
  updateStrategy:
    registryPoll: (2)
      interval: 30m
EOF
```

(1) Specify your index image.

(2) Catalog sources can automatically check for new versions to keep up to date.

2. Verify using the OpenShift Container Platform web console or CLI that the catalog loaded successfully and that packages are available. For example, using the CLI:

   a. Check the pods:

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

   b. Check the catalog source:

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

   c. Check the package manifest:

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

3.7.2.5. Updating an index image

You can update an existing index image using the **opm** CLI.

Prerequisites

- **opm** version 1.12.3+
- **podman** version 1.4.4+
- An index image built and pushed to a registry.
- A **CatalogSource** object created and applied to a cluster.

Procedure

1. Update the existing index:

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

1. The additional bundle image to add to the index.
2. The existing index that was previously pushed.
3. The image tag that you want the updated index image to have.

2. Push the updated index image:

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

3. After Operator Lifecycle Manager (OLM) polls the index image at its regular interval, verify that the new packages are successfully added:

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

3.8. USING OPERATOR LIFECYCLE MANAGER ON RESTRICTED NETWORKS

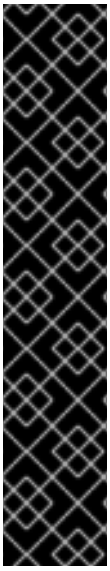
For OpenShift Container Platform clusters that are installed on restricted networks, also known as disconnected clusters, Operator Lifecycle Manager (OLM) by default cannot access the Red Hat-provided OperatorHub sources hosted remotely on Quay.io because those remote sources require full Internet connectivity.

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 is used to prepare local mirrors of the remote OperatorHub sources, and requires full Internet access to pull the remote content.

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 local mirrors of the OperatorHub content.
- Configure OLM to install and manage Operators from the local sources 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

- Enabling your Operator for restricted network environments

3.8.1. Understanding Operator catalog images

Operator Lifecycle Manager (OLM) always installs Operators from the latest version of an Operator catalog. For OpenShift Container Platform 4.5, Red Hat-provided Operators are distributed via Quay App Registry catalogs from quay.io.

Table 3.3. Red Hat-provided App Registry catalogs

Catalog	Description
redhat-operators	Public catalog for Red Hat products packaged and shipped by Red Hat. Supported by Red Hat.
certified-operators	Public catalog for products from leading independent software vendors (ISVs). Red Hat partners with ISVs to package and ship. Supported by the ISV.
community-operators	Public catalog for software maintained by relevant representatives in the operator-framework/community-operators GitHub repository. No official support.

As catalogs are updated, the latest versions of Operators change, and older versions may be removed or altered. This behavior can cause problems maintaining reproducible installs over time. In addition, when OLM runs on an OpenShift Container Platform cluster in a restricted network environment, it is unable to access the catalogs from quay.io directly.

Using the **oc adm catalog build** command, cluster administrators can create an *Operator catalog image*. An Operator catalog image is:

- a point-in-time export of an App Registry type catalog's content.
- the result of converting an App Registry catalog to a container image type catalog.

- an immutable artifact.

Creating an Operator catalog image provides a simple way to use this content without incurring the aforementioned issues.

3.8.2. Building an Operator 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 1.4.4+
- 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:

```
$ 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:

```
$ 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>
```

   Also authenticate with **registry.redhat.io** so that the base image can be pulled during the build:

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

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

   1. Organization (namespace) to pull from an App Registry instance.
   2. Set **--from** to the **ose-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, **v1**.
   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: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/9cskxqs53ll3wdn434vw4cd80000gn/T/300666084/manifests-829192605
file=4.2 load=package
W1114 19:42:37.876180   34665 builder.go:141] error building database: error loading
package into db: fuse-camel-k-operator.v7.5.0 specifies replacement that couldn't be found
Uploading ... 244.9kB/s
```

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

3.8.3. Configuring OperatorHub for restricted networks

Cluster administrators can configure OLM and OperatorHub to use local content in a restricted network environment using a custom Operator catalog image. 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 pushed to a supported registry
- **oc** version 4.3.5+
- **podman** version 1.4.4+
- 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:

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

Procedure

1. Disable the default **OperatorSource** objects by adding **disableAllDefaultSources: true** to the spec:

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

This disables the default sources that are configured by default during an OpenShift Container Platform installation.

2. 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 \ 1
    <registry_host_name>:<port> \
    [-a ${REG_CREDS}] \ 2
    [--insecure] \ 3
    --filter-by-os='.*' \ 4
    [--manifests-only] 5
```

1. Specify your Operator catalog image.
2. Optional: If required, specify the location of your registry credentials file.
3. Optional: If you do not want to configure trust for the target registry, add the **--insecure** flag.
4. This flag is currently required due to a known issue with multiple architecture support.
5. Optional: Only generate the manifests required for mirroring and do not actually mirror the image content to a registry.

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. For more information, see BZ#1890951.

Example output

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

1. Temporary database generated by the command.

After running the command, a **<image_name>-manifests/** directory is created in the current directory and generates the following files:

- 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.
- 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.

3. 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**:

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

      1 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/openshift4/ose-logging-
kibana5@sha256:aa4a8b2a00836d0e28aa6497ad90a3c116f135f382d8211e3c55f34f
b36dfe61
operatorbundle_name = clusterlogging.4.3.33-202008111029.p0

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

      ii. 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

```
registry.redhat.io/openshift4/ose-logging-
kibana5@sha256:aa4a8b2a00836d0e28aa6497ad90a3c116f135f382d8211e3c55f34f
b36dfe61=<registry_host_name>:<port>/openshift4-ose-logging-kibana5:a767c8f0
```

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

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:

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

4. Apply the **ImageContentSourcePolicy** object:

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

5. Create a **CatalogSource** object that references your catalog image.

a. Modify the following to your specifications and save it as a **catalogsource.yaml** file:

```
apiVersion: operators.coreos.com/v1alpha1
kind: CatalogSource
metadata:
  name: my-operator-catalog
  namespace: openshift-marketplace
spec:
  sourceType: grpc
  image: <registry_host_name>:<port>/olm/redhat-operators:v1 (1)
  displayName: My Operator Catalog
  publisher: grpc
```

(1) Specify your custom Operator catalog image.

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

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

6. Verify the following resources are created successfully.

   a. Check the pods:

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

   Example output

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

   b. Check the catalog source:

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

   Example output

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

   c. Check the package manifest:

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

   Example output

   ```
   NAME    CATALOG             AGE
   etcd    My Operator Catalog  34s
   ```

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

Additional resources

- Mirroring images for a disconnected installation
- Architecture and operating system support for Operators

3.8.4. Updating an Operator 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 1.4.4+
- Access to mirror registry that supports Docker v2-2
- 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
```

- 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>
```

   Also authenticate with **registry.redhat.io** so that the base image can be pulled during the build:

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

2. 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:v4.5 \ 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 **ose-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:f73d42950021f9240389f99ddc5b0c7f1b533c054ba344654ff1edaf6bf827e3
to example_registry:5000/olm/redhat-operators:v2
```

3. 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> \
  [-a ${REG_CREDS}] \ 2
  [--insecure] \ 3
  --filter-by-os='.*' 4
```

1. Specify your new Operator catalog image.
2. Optional: If required, specify the location of your registry credentials file.
3. Optional: If you do not want to configure trust for the target registry, add the **--insecure** flag.
4. This flag is currently required due to a known issue with multiple architecture support. 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. For more information, see BZ#1890951.

4. Apply the newly generated manifests:

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

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.

5. 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 1
  displayName: My Operator Catalog
  publisher: grpc
```

1 Specify your new Operator catalog image.

      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

3.8.5. Testing an Operator 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 an OLM subscription by referencing the image in a

CatalogSource object. For this example, the procedure uses a custom **redhat-operators** catalog image previously built and pushed to a supported registry.

Prerequisites

- A custom Operator catalog image pushed to a supported registry
- **podman** version 1.4.4+
- **oc** version 4.3.5+
- Access to mirror registry that supports Docker v2-2
- **grpcurl**

Procedure

1. Pull the Operator catalog image:

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

2. Run the image:

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

3. Query the running image for available packages using **grpcurl**:

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

Example output

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

4. Get the latest Operator bundle in a channel:

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

Example output

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

```
  "channelName": "stable",
...
```

5. Get the digest of the image:

```
$ 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:

```
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:

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

CHAPTER 4. DEVELOPING OPERATORS

4.1. GETTING STARTED WITH THE OPERATOR SDK

This guide outlines the basics of the Operator SDK and walks Operator authors with cluster administrator access to a Kubernetes-based cluster (such as OpenShift Container Platform) through an example of building a simple Go-based Memcached Operator and managing its lifecycle from installation to upgrade.

This is accomplished using two centerpieces of the Operator Framework: Operator SDK (the **operator-sdk** CLI tool and **controller-runtime** library API) and Operator Lifecycle Manager (OLM).

NOTE

OpenShift Container Platform 4.5 supports Operator SDK v0.17.2.

4.1.1. Architecture of 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 is a framework designed to make writing Operators easier by providing:

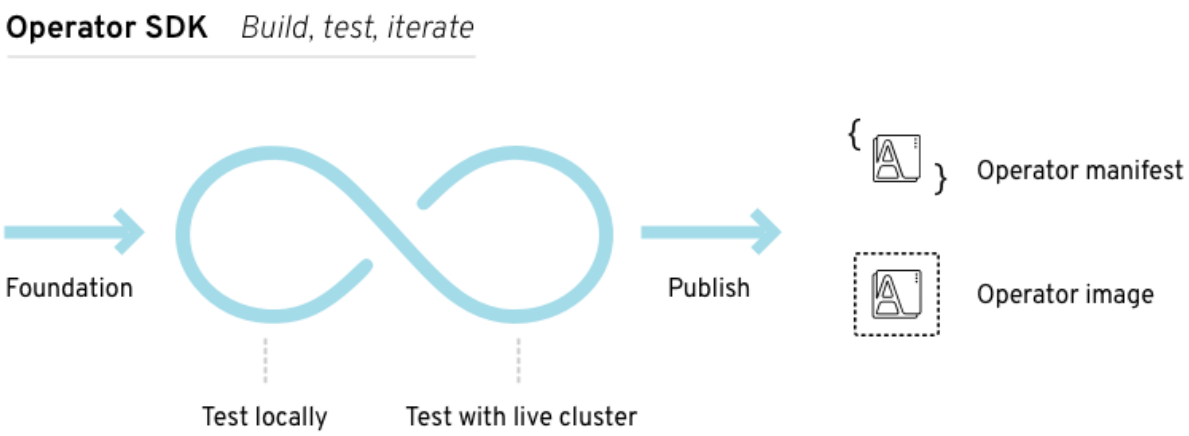
- High-level APIs and abstractions to write the operational logic more intuitively
- Tools for scaffolding and code generation to quickly bootstrap a new project
- Extensions to cover common Operator use cases

4.1.1.1. Workflow

The Operator SDK provides the following workflow to develop a new Operator:

1. Create a new Operator project using the Operator SDK command line interface (CLI).
2. Define new resource APIs by adding custom resource definitions (CRDs).
3. Specify resources to watch 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 4.1. Operator SDK workflow



At a high level, an Operator using the Operator SDK processes events for watched resources in an Operator author-defined handler and takes actions to reconcile the state of the application.

4.1.1.2. Manager file

The main program for the Operator is the manager file at **cmd/manager/main.go**. The manager automatically registers the scheme for all custom resources (CRs) defined under **pkg/apis/** and runs all controllers under **pkg/controller/**.

The manager can restrict the namespace that all controllers watch for resources:

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

By default, this is the namespace that the Operator is running in. To watch all namespaces, you can leave the namespace option empty:

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

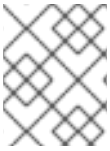
4.1.1.3. Prometheus Operator support

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.

4.1.2. Installing the Operator SDK CLI

The Operator SDK has a CLI tool that assists developers in creating, building, and deploying a new Operator project. You can install the SDK CLI on your workstation so you are prepared to start authoring your own Operators.



NOTE

This guide uses minikube v0.25.0+ as the local Kubernetes cluster and Quay.io for the public registry.

4.1.2.1. Installing from GitHub release

You can download and install a pre-built release binary of the Operator SDK CLI from the project on GitHub.

Prerequisites

- Go v1.13+
- **docker** v17.03+, **podman** v1.2.0+, or **buildah** v1.7+
- OpenShift CLI (**oc**) v4.5+ installed
- Access to a cluster based on Kubernetes v1.12.0+
- Access to a container registry

Procedure

1. Set the release version variable:

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

2. Download the release binary.
   - For Linux:

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

   - For macOS:

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

3. Verify the downloaded release binary.
   a. Download the provided **.asc** file.
      - For Linux:

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

      - For macOS:

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

b. Place the binary and corresponding **.asc** file into the same directory and run the following command to verify the binary:

- For Linux:

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

- For macOS:

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

If you do not have the public key of the maintainer on your workstation, you will get the following error:

Example output with error

```
$ gpg: assuming signed data in 'operator-sdk-${RELEASE_VERSION}-x86_64-apple-
darwin'
$ gpg: Signature made Fri Apr  5 20:03:22 2019 CEST
$ gpg:                using RSA key <key_id> 1
$ gpg: Can't check signature: No public key
```

1 RSA key string.

To download the key, run the following command, replacing **<key_id>** with the RSA key string provided in the output of the previous command:

```
$ gpg [--keyserver keys.gnupg.net] --recv-key "<key_id>" 1
```

1 If you do not have a key server configured, specify one with the **--keyserver** option.

4. Install the release binary in your **PATH**:

- For Linux:

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

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

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

- For macOS:

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

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

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

5. Verify that the CLI tool was installed correctly:

```
$ operator-sdk version
```

4.1.2.2. Installing from Homebrew

You can install the SDK CLI using Homebrew.

Prerequisites

- Homebrew
- **docker** v17.03+, **podman** v1.2.0+, or **buildah** v1.7+
- OpenShift CLI (**oc**) v4.5+ installed
- Access to a cluster based on Kubernetes v1.12.0+
- Access to a container registry

Procedure

1. Install the SDK CLI using the **brew** command:

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

2. Verify that the CLI tool was installed correctly:

```
$ operator-sdk version
```

4.1.2.3. Compiling and installing from source

You can obtain the Operator SDK source code to compile and install the SDK CLI.

Prerequisites

- Git
- Go v1.13+
- **docker** v17.03+, **podman** v1.2.0+, or **buildah** v1.7+
- OpenShift CLI (**oc**) v4.5+ installed
- Access to a cluster based on Kubernetes v1.12.0+
- Access to a container registry

Procedure

1. Clone the **operator-sdk** repository:

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

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

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

```
$ cd operator-sdk
```

2. Check out the desired release branch:

```
$ git checkout master
```

3. Compile and install the SDK CLI:

```
$ make dep
```

```
$ make install
```

   This installs the CLI binary **operator-sdk** at *$GOPATH/bin*.

4. Verify that the CLI tool was installed correctly:

```
$ operator-sdk version
```

4.1.3. Building a Go-based Operator using the Operator SDK

The Operator SDK makes it easier to build Kubernetes native applications, a process that can require deep, application-specific operational knowledge. The SDK not only lowers that barrier, but it also helps reduce the amount of boilerplate code needed for many common management capabilities, such as metering or monitoring.

This procedure walks through an example of building a simple Memcached Operator using tools and libraries provided by the SDK.

Prerequisites

- Operator SDK CLI installed on the development workstation
- Operator Lifecycle Manager (OLM) installed on a Kubernetes-based cluster (v1.8 or above to support the **apps/v1beta2** API group), for example OpenShift Container Platform 4.5
- Access to the cluster using an account with **cluster-admin** permissions
- OpenShift CLI (**oc**) v4.5+ installed

Procedure

1. **Create a new project.**

Use the CLI to create a new **memcached-operator** project:

```
$ mkdir -p $GOPATH/src/github.com/example-inc/
```

```
$ cd $GOPATH/src/github.com/example-inc/
```

```
$ operator-sdk new memcached-operator
```

```
$ cd memcached-operator
```

2. **Add a new custom resource definition (CRD).**

   a. Use the CLI to add a new CRD API called **Memcached**, with **APIVersion** set to **cache.example.com/v1apha1** and **Kind** set to **Memcached**:

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

   This scaffolds the Memcached resource API under **pkg/apis/cache/v1alpha1/**.

   b. Modify the spec and status of the **Memcached** custom resource (CR) at the **pkg/apis/cache/v1alpha1/memcached_types.go** file:

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

   c. After modifying the ***_types.go** file, always run the following command to update the generated code for that resource type:

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

3. **Optional: Add custom validation to your CRD.**
   OpenAPI v3.0 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 CR when it is created or updated.

   Additionally, a **pkg/apis/<group>/<version>/zz_generated.openapi.go** file is generated. This file contains the Go representation of this validation block if the **+k8s:openapi-gen=true annotation** is present above the **Kind** type declaration, which is present by default. This auto-generated code is the OpenAPI model of your Go **Kind** type, from which you can create a full OpenAPI Specification and generate a client.

   As an Operator author, you can use Kubebuilder markers (annotations) to configure custom validations for your API. These markers must always have a **+kubebuilder:validation** prefix. For example, adding an enum-type specification can be done by adding the following marker:

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

Usage of markers in API code is discussed in the Kubebuilder Generating CRDs and Markers for Config/Code Generation documentation. A full list of OpenAPIv3 validation markers is also available in the Kubebuilder CRD Validation documentation.

If you add any custom validations, run the following command to update the OpenAPI validation section in the **deploy/crds/cache.example.com_memcacheds_crd.yaml** file for the CRD:

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

Example generated YAML

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

4. **Add a new controller.**

   a. Add a new controller to the project to watch and reconcile the **Memcached** resource:

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

   This scaffolds a new controller implementation under **pkg/controller/memcached/**.

   b. For this example, replace the generated controller file **pkg/controller/memcached/memcached_controller.go** with the example implementation. The example controller executes the following reconciliation logic for each **Memcached** resource:

   - 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** resource status with the names of the Memcached pods.

   The next two sub-steps inspect how the controller watches resources and how the reconcile loop is triggered. You can skip these steps to go directly to building and running the Operator.

   c. Inspect the controller implementation at the **pkg/controller/memcached/memcached_controller.go** file to see how the controller watches resources.
   The first watch is for the **Memcached** type as the primary resource. For each add, update, or delete event, the reconcile loop is sent a reconcile **Request** (a **<namespace>:<name>** key) for that **Memcached** object:

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

The next watch is for **Deployment** objects, but the event handler maps each event to a reconcile **Request** for the owner of the deployment. In this case, this is the **Memcached** object for which the deployment was created. This allows the controller to watch deployments as a secondary resource:

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

d. 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>:<name>** key used to lookup the primary resource object, **Memcached**, from the cache:

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

Based on the return value of the **Reconcile()** function, the reconcile **Request** might be requeued, and the loop might be triggered again:

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

5. **Build and run the Operator.**

a. Before running the Operator, the CRD must be registered with the Kubernetes API server:

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

b. After registering the CRD, there are two options for running the Operator:

- As a Deployment inside a Kubernetes cluster
- As Go program outside a cluster

Choose one of the following methods.

i. *Option A:* Running as a deployment inside the cluster.

A. Build the **memcached-operator** image and push it to a registry:

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

B. The deployment manifest is generated at **deploy/operator.yaml**. Update the deployment image as follows since the default is just a placeholder:

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

C. Ensure you have an account on Quay.io for the next step, or substitute your preferred container registry. On the registry, create a new public image repository named **memcached-operator**.

D. Push the image to the registry:

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

E. Set up RBAC and create the **memcached-operator** manifests:

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

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

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

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

F. Verify that the **memcached-operator** deploy is up and running:

```
$ oc get deployment
```

Example output

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

ii. *Option B:* Running locally outside the cluster.
This method is preferred during development cycle to deploy and test faster.

Run the Operator locally with the default Kubernetes configuration file present at **$HOME/.kube/config**:

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

You can use a specific **kubeconfig** using the flag **--kubeconfig=<path/to/kubeconfig>**.

6. **Verify that the Operator can deploy a Memcached application** by creating a **Memcached** CR.

a. Create the example **Memcached** CR that was generated at **deploy/crds/cache_v1alpha1_memcached_cr.yaml**.

b. View the file:

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

Example output

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

c. Create the object:

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

d. Ensure that **memcached-operator** creates the deployment for the CR:

```
$ oc get deployment
```

Example output

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

e. Check the pods and CR to confirm the CR status is updated with the pod names:

```
$ oc get pods
```

Example output

```
NAME                                  READY     STATUS    RESTARTS   AGE
example-memcached-6fd7c98d8-7dqdr     1/1       Running   0          1m
example-memcached-6fd7c98d8-g5k7v     1/1       Running   0          1m
example-memcached-6fd7c98d8-m7vn7     1/1       Running   0          1m
memcached-operator-7cc7cfdf86-vvjqk   1/1       Running   0          2m
```

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

Example output

```
apiVersion: cache.example.com/v1alpha1
kind: Memcached
metadata:
  clusterName: ""
  creationTimestamp: 2018-03-31T22:51:08Z
  generation: 0
  name: example-memcached
  namespace: default
```

```
  resourceVersion: "245453"
  selfLink:
/apis/cache.example.com/v1alpha1/namespaces/default/memcacheds/example-
memcached
  uid: 0026cc97-3536-11e8-bd83-0800274106a1
spec:
  size: 3
status:
  nodes:
  - example-memcached-6fd7c98d8-7dqdr
  - example-memcached-6fd7c98d8-g5k7v
  - example-memcached-6fd7c98d8-m7vn7
```

7. **Verify that the Operator can manage a deployed Memcached application** by updating the size of the deployment.

   a. Change the **spec.size** field in the **memcached** CR from **3** to **4**:

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

   Example output

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

   b. Apply the change:

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

   c. Confirm that the Operator changes the deployment size:

   ```
   $ oc get deployment
   ```

   Example output

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

8. **Clean up the resources:**

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

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

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

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

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

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

Additional resources

- For more information about OpenAPI v3.0 validation schemas in CRDs, refer to the Kubernetes documentation.

4.1.4. Managing a Go-based Operator using Operator Lifecycle Manager

The previous section has covered manually running an Operator. The next sections explore using Operator Lifecycle Manager (OLM), which is what enables a more robust deployment model for Operators being run in production environments.

OLM helps you to install, update, and generally manage the lifecycle of all of the Operators and their associated services on a Kubernetes cluster. It runs as an Kubernetes extension and lets you use **oc** for all the lifecycle management functions without any additional tools.

Prerequisites

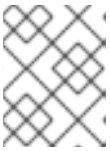
- OLM installed on a Kubernetes-based cluster (v1.8 or above to support the **apps/v1beta2** API group), for example OpenShift Container Platform 4.5
- Memcached Operator built

Procedure

1. **Generate an Operator manifest.**
   An Operator manifest describes how to display, create, and manage the application, in this case Memcached, as a whole. It is defined by a **ClusterServiceVersion** (CSV) object and is required for OLM to function.

   From the **memcached-operator**/ directory that was created when you built the Memcached Operator, generate the CSV manifest:

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

   NOTE

   See Building a CSV for the Operator Framework for more information on manually defining a manifest file.

2. **Create an Operator group** that specifies the namespaces that the Operator will target. Create the following Operator group in the namespace where you will create the CSV. In this example, the **default** namespace is used:

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

```
spec:
 targetNamespaces:
 - default
```

3. **Deploy the Operator.** Use the files that were generated into the **deploy**/ directory by the Operator SDK when you built the Memcached Operator.

   a. Edit the generated CSV manifest file by adding **displayName** fields for each custom resource definition (CRD) **kind** in the **spec.customresourcedefinitions.owned** section:

   deploy/olm-catalog/memcached-operator/0.0.1/memcached-operator.v0.0.1.clusterserviceversion.yaml file

   ```
   ...
   spec:
     customresourcedefinitions:
       owned:
       - kind: Memcached
         name: memcacheds.cache.example.com
         version: v1alpha1
         description: Memcached is the Schema for the memcacheds API
         displayName: Memcached (1)
   ...
   ```

   (1) Specify a display name for the CRD.

   b. Apply the CSV manifest to the specified namespace in the cluster:

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

   When you apply this manifest, the cluster does not immediately update because it does not yet meet the requirements specified in the manifest.

   c. Create the role, role binding, and service account to grant resource permissions to the Operator, and the custom resource definition (CRD) to create the **Memcached** custom resource that the Operator manages:

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

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

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

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

   Because OLM creates Operators in a particular namespace when a manifest is applied, administrators can leverage the native Kubernetes RBAC permission model to restrict which users are allowed to install Operators.

4. **Create an application instance.**
   The Memcached Operator is now running in the **default** namespace. Users interact with

Operators via instances of custom resources; in this case, the resource has the kind **Memcached**. Native Kubernetes RBAC also applies to custom resources, providing administrators control over who can interact with each Operator.

Creating instances of **Memcached** objects in this namespace will now trigger the Memcached Operator to instantiate pods running the **memcached** server that are managed by the Operator. The more custom resources you create, the more unique Memcached application instances are managed by the Memcached Operator running in this namespace.

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

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

```
$ oc get Memcached
```

Example output

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

```
$ oc get pods
```

Example output

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

4.1.5. Additional resources

- See Appendices to learn about the project directory structures created by the Operator SDK.
- Operator Development Guide for Red Hat Partners

4.2. CREATING ANSIBLE-BASED OPERATORS

This guide outlines Ansible support in the Operator SDK and walks Operator authors through examples building and running Ansible-based Operators with the **operator-sdk** CLI tool that use Ansible playbooks and modules.

4.2.1. Ansible support in 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. This framework includes the Operator SDK, which assists developers in bootstrapping and building an Operator based on their expertise without requiring knowledge of Kubernetes API complexities.

One of the Operator SDK options for generating an Operator project includes leveraging existing Ansible playbooks and modules to deploy Kubernetes resources as a unified application, without having to write any Go code.

4.2.1.1. 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 4.1. Custom resource fields

Field	Description
apiVersion	Version of the CR to be created.
kind	Kind of the CR to be created.
metadata	Kubernetes-specific metadata to be created.
spec (optional)	Key-value list of variables which are passed to Ansible. This field is empty by default.
status	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**.
annotations	Kubernetes-specific annotations to be appended to the CR.

The following list of CR annotations modify the behavior of the Operator:

Table 4.2. Ansible-based Operator annotations

Annotation	Description
ansible.operator-sdk/reconcile-period	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.

Example Ansible-based Operator annotation

```
apiVersion: "test1.example.com/v1alpha1"
kind: "Test1"
metadata:
  name: "example"
annotations:
  ansible.operator-sdk/reconcile-period: "30s"
```

4.2.1.2. 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 4.3. watches.yaml file mappings

Field	Description
group	Group of CR to watch.
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 (1)
  group: test1.example.com
  kind: Test1
  role: /opt/ansible/roles/Test1

- version: v1alpha1 (2)
  group: test2.example.com
  kind: Test2
```

```
    playbook: /opt/ansible/playbook.yml

  - version: v1alpha1 (3)
    group: test3.example.com
    kind: Test3
    playbook: /opt/ansible/test3.yml
    reconcilePeriod: 0
    manageStatus: false
```

1. 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.

4.2.1.2.1. 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 4.4. Advanced watches.yaml file options

Feature	YAML key	Description	Annotation for override	Default value
Reconcile period	**reconcilePeriod**	Time between reconcile runs for a particular CR.	**ansbile.operator-sdk/reconcile-period**	**1m**
Manage status	**manageStatus**	Allows the Operator to manage the **conditions** section of each CR **status** section.		**true**
Watch dependent resources	**watchDependentResources**	Allows the Operator to dynamically watch resources that are created by Ansible.		**true**
Watch cluster-scoped resources	**watchClusterScopedResources**	Allows the Operator to watch cluster-scoped resources that are created by Ansible.		**false**

Feature	YAML key	Description	Annotation for override	Default value
Max runner artifacts	**maxRunnerArtifacts**	Manages the number of artifact directories that Ansible Runner keeps in the Operator container for each individual resource.	**ansible.operator-sdk/max-runner-artifacts**	**20**

Example watches.yml file with advanced options

```
- version: v1alpha1
  group: app.example.com
  kind: AppService
  playbook: /opt/ansible/playbook.yml
  maxRunnerArtifacts: 30
  reconcilePeriod: 5s
  manageStatus: False
  watchDependentResources: False
```

4.2.1.3. 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:

```
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:

```
{ "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:

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

4.2.1.4. Ansible Runner directory

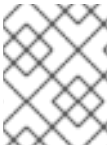
Ansible Runner keeps information about Ansible runs in the container. This is located at **/tmp/ansible-operator/runner/<group>/<version>/<kind>/<namespace>/<name>**.

Additional resources

- To learn more about the **runner** directory, see the Ansible Runner documentation.

4.2.2. Installing the Operator SDK CLI

The Operator SDK has a CLI tool that assists developers in creating, building, and deploying a new Operator project. You can install the SDK CLI on your workstation so you are prepared to start authoring your own Operators.



NOTE

This guide uses minikube v0.25.0+ as the local Kubernetes cluster and Quay.io for the public registry.

4.2.2.1. Installing from GitHub release

You can download and install a pre-built release binary of the Operator SDK CLI from the project on GitHub.

Prerequisites

- Go v1.13+
- **docker** v17.03+, **podman** v1.2.0+, or **buildah** v1.7+
- OpenShift CLI (**oc**) v4.5+ installed
- Access to a cluster based on Kubernetes v1.12.0+
- Access to a container registry

Procedure

1. Set the release version variable:

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

2. Download the release binary.
   - For Linux:

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

- For macOS:

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

3. Verify the downloaded release binary.

   a. Download the provided **.asc** file.

   - For Linux:

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

   - For macOS:

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

   b. Place the binary and corresponding **.asc** file into the same directory and run the following command to verify the binary:

   - For Linux:

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

   - For macOS:

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

   If you do not have the public key of the maintainer on your workstation, you will get the following error:

   Example output with error

```
$ gpg: assuming signed data in 'operator-sdk-${RELEASE_VERSION}-x86_64-apple-darwin'
$ gpg: Signature made Fri Apr  5 20:03:22 2019 CEST
$ gpg:                using RSA key <key_id> 1
$ gpg: Can't check signature: No public key
```

   1 RSA key string.

   To download the key, run the following command, replacing **<key_id>** with the RSA key string provided in the output of the previous command:

```
$ gpg [--keyserver keys.gnupg.net] --recv-key "<key_id>" 1
```

1 If you do not have a key server configured, specify one with the **--keyserver** option.

4. Install the release binary in your **PATH**:

   - For Linux:

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

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

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

   - For macOS:

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

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

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

5. Verify that the CLI tool was installed correctly:

```
$ operator-sdk version
```

4.2.2.2. Installing from Homebrew

You can install the SDK CLI using Homebrew.

Prerequisites

- Homebrew
- **docker** v17.03+, **podman** v1.2.0+, or **buildah** v1.7+
- OpenShift CLI (**oc**) v4.5+ installed
- Access to a cluster based on Kubernetes v1.12.0+
- Access to a container registry

Procedure

1. Install the SDK CLI using the **brew** command:

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

2. Verify that the CLI tool was installed correctly:

```
$ operator-sdk version
```

4.2.2.3. Compiling and installing from source

You can obtain the Operator SDK source code to compile and install the SDK CLI.

Prerequisites

- Git
- Go v1.13+
- **docker** v17.03+, **podman** v1.2.0+, or **buildah** v1.7+
- OpenShift CLI (**oc**) v4.5+ installed
- Access to a cluster based on Kubernetes v1.12.0+
- Access to a container registry

Procedure

1. Clone the **operator-sdk** repository:

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

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

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

```
$ cd operator-sdk
```

2. Check out the desired release branch:

```
$ git checkout master
```

3. Compile and install the SDK CLI:

```
$ make dep
```

```
$ make install
```

   This installs the CLI binary **operator-sdk** at *$GOPATH/bin*.

4. Verify that the CLI tool was installed correctly:

```
$ operator-sdk version
```

4.2.3. Building an Ansible-based Operator using the Operator SDK

This procedure walks through an example of building a simple Memcached Operator powered by Ansible playbooks and modules using tools and libraries provided by the Operator SDK.

Prerequisites

- Operator SDK CLI installed on the development workstation
- Access to a Kubernetes-based cluster v1.11.3+ (for example OpenShift Container Platform 4.5) using an account with **cluster-admin** permissions
- OpenShift CLI (**oc**) v4.5+ installed
- **ansible** v2.9.0+
- **ansible-runner** v1.1.0+
- **ansible-runner-http** v1.0.0+

Procedure

1. **Create a new Operator project.** A namespace-scoped Operator watches and manages resources in a single namespace. Namespace-scoped Operators are preferred because of their flexibility. They enable decoupled upgrades, namespace isolation for failures and monitoring, and differing API definitions.
   To create a new Ansible-based, namespace-scoped **memcached-operator** project and change to the new directory, use the following commands:

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

   ```
   $ cd memcached-operator
   ```

   This creates the **memcached-operator** project specifically for watching the **Memcached** resource with API version **example.com/v1apha1** and kind **Memcached**.

2. **Customize the Operator logic.**
   For this example, the **memcached-operator** executes 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.

   By default, the **memcached-operator** watches **Memcached** resource events as shown in the **watches.yaml** file and executes the Ansible role **Memcached**:

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

   You can optionally customize the following logic in the **watches.yaml** file:

a. Specifying a **role** option configures the Operator to use this specified path when launching **ansible-runner** with an Ansible role. By default, the **operator-sdk new** command fills in an absolute path to where your role should go:

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

b. Specifying a **playbook** option in the **watches.yaml** file configures the Operator to use this specified path when launching **ansible-runner** with an Ansible playbook:

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

3. **Build the Memcached Ansible role.**
Modify the generated Ansible role under the **roles/memcached/** directory. This Ansible role controls the logic that is executed when a resource is modified.

a. **Define the Memcached spec.**
Defining the spec for an Ansible-based Operator can be done entirely in Ansible. The Ansible Operator passes all key-value pairs listed in the CR spec field along to Ansible as variables. The names of all variables in the spec field are converted to snake case (lowercase with an underscore) by the Operator before running Ansible. For example, **serviceAccount** in the spec becomes **service_account** in Ansible.

TIP

You should perform some type validation in Ansible on the variables to ensure that your application is receiving expected input.

In case the user does not set the **spec** field, set a default by modifying the **roles/memcached/defaults/main.yml** file:

```
size: 1
```

b. **Define the Memcached deployment.**
With the **Memcached** spec now defined, you can define what Ansible is actually executed on resource changes. Because this is an Ansible role, the default behavior executes the tasks in the **roles/memcached/tasks/main.yml** file.

The goal is for Ansible to create a deployment if it does not exist, which runs the **memcached:1.4.36-alpine** image. Ansible 2.7+ supports the k8s Ansible module, which this example leverages to control the deployment definition.

Modify the **roles/memcached/tasks/main.yml** to match the following:

```
- name: start memcached
  k8s:
    definition:
      kind: Deployment
      apiVersion: apps/v1
```

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

NOTE

This example used the **size** variable to control the number of replicas of the **Memcached** deployment. This example sets the default to **1**, but any user can create a CR that overwrites the default.

4. **Deploy the CRD.**
   Before running the Operator, Kubernetes needs to know about the new custom resource definition (CRD) that the Operator will be watching. Deploy the **Memcached** CRD:

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

5. **Build and run the Operator.**
   There are two ways to build and run the Operator:

   - As a pod inside a Kubernetes cluster.
   - As a Go program outside the cluster using the **operator-sdk up** command.

   Choose one of the following methods:

   a. **Run as a pod**inside a Kubernetes cluster. This is the preferred method for production use.

      i. Build the **memcached-operator** image and push it to a registry:

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

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

ii. Deployment manifests are generated in the **deploy/operator.yaml** file. The deployment image in this file needs to be modified from the placeholder **REPLACE_IMAGE** to the previous built image. To do this, run:

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

iii. Deploy the **memcached-operator** manifests:

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

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

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

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

iv. Verify that the **memcached-operator** deployment is up and running:

```
$ oc get deployment
```

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

b. **Run outside the cluster.** This method is preferred during the development cycle to speed up deployment and testing.
Ensure that Ansible Runner and Ansible Runner HTTP Plug-in are installed or else you will see unexpected errors from Ansible Runner when a CR is created.

It is also important that the role path referenced in the **watches.yaml** file exists on your machine. Because normally a container is used where the role is put on disk, the role must be manually copied to the configured Ansible roles path (for example **/etc/ansible/roles**).

i. To run the Operator locally with the default Kubernetes configuration file present at **$HOME/.kube/config**:

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

To run the Operator locally with a provided Kubernetes configuration file:

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

6. **Create a Memcached CR.**

a. Modify the **deploy/crds/cache_v1alpha1_memcached_cr.yaml** file as shown and create a **Memcached** CR:

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

Example output

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

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

b. Ensure that the **memcached-operator** creates the deployment for the CR:

```
$ oc get deployment
```

Example output

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

c. Check the pods to confirm three replicas were created:

```
$ oc get pods
```

```
NAME                                  READY     STATUS    RESTARTS   AGE
example-memcached-6fd7c98d8-7dqdr     1/1       Running   0          1m
example-memcached-6fd7c98d8-g5k7v     1/1       Running   0          1m
example-memcached-6fd7c98d8-m7vn7     1/1       Running   0          1m
memcached-operator-7cc7cfdf86-vvjqk   1/1       Running   0          2m
```

7. **Update the size.**

a. Change the **spec.size** field in the **memcached** CR from **3** to **4** and apply the change:

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

Example output

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

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

b. Confirm that the Operator changes the deployment size:

```
$ oc get deployment
```

Example output

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

8. **Clean up the resources:**

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

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

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

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

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

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

4.2.4. Managing application lifecycle using the k8s Ansible module

To manage the lifecycle of your application on Kubernetes using Ansible, you can use the **k8s** Ansible module. This Ansible module 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 **k8s** Ansible module. To get started, install the module on your local workstation and test it using a playbook before moving on to using it within an Operator.

4.2.4.1. Installing the k8s Ansible module

To install the **k8s** Ansible module on your local workstation:

Procedure

1. Install Ansible 2.9+:

```
$ sudo yum install ansible
```

2. Install the OpenShift python client package using **pip**:

```
$ sudo pip install openshift
```

```
$ sudo pip install kubernetes
```

4.2.4.2. Testing the k8s Ansible module locally

Sometimes, it is beneficial for a developer to run the Ansible code from their local machine as opposed to running and rebuilding the Operator each time.

Procedure

1. Install the **community.kubernetes** collection:

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

2. Initialize a new Ansible-based Operator project:

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

Example output

```
Create test1-operator/tmp/init/galaxy-init.sh
Create test1-operator/tmp/build/Dockerfile
Create test1-operator/tmp/build/test-framework/Dockerfile
Create test1-operator/tmp/build/go-test.sh
Rendering Ansible Galaxy role [test1-operator/roles/test1]...
Cleaning up test1-operator/tmp/init
Create test1-operator/watches.yaml
Create test1-operator/deploy/rbac.yaml
Create test1-operator/deploy/crd.yaml
Create test1-operator/deploy/cr.yaml
Create test1-operator/deploy/operator.yaml
Run git init ...
Initialized empty Git repository in /home/user/go/src/github.com/user/opsdk/test1-
operator/.git/
Run git init done
```

```
$ cd test1-operator
```

3. Modify the **roles/test1/tasks/main.yml** file with the Ansible logic that you want. This example creates and deletes a namespace with the switch of a variable.

```
- name: set test namespace to "{{ state }}"
  community.kubernetes.k8s:
    api_version: v1
    kind: Namespace
    state: "{{ state }}"
    name: test
  ignore_errors: true (1)
```

(1) Setting **ignore_errors: true** ensures that deleting a nonexistent project does not fail.

4. Modify the **roles/test1/defaults/main.yml** file to set **state** to **present** by default:

```
state: present
```

5. Create an Ansible playbook **playbook.yml** in the top-level directory, which includes the **test1** role:

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

6. Run the playbook:

```
$ 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] ***************************************************************************

PROCEDURE [Gathering Facts]
*********************************************************************
ok: [localhost]

Task [test1 : set test namespace to present]
changed: [localhost]

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

7. Check that the namespace was created:

```
$ oc get namespace
```

Example output

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

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

PROCEDURE [Gathering Facts]
*********************************************************************
ok: [localhost]

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

```
changed: [localhost]

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

9. Check that the namespace was deleted:

```
$ oc get namespace
```

Example output

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

4.2.4.3. Testing the k8s Ansible module inside an Operator

After you are familiar with using the **k8s** Ansible module 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.

4.2.4.3.1. Testing an Ansible-based Operator locally

After getting comfortable testing Ansible workflows locally, you can test the logic inside of an Ansible-based Operator running locally.

To do so, use the **operator-sdk run --local** command from the top-level directory of your Operator project. This command reads from the **watches.yaml** file and uses the **~/.kube/config** file to communicate with a Kubernetes cluster just as the **k8s** Ansible module does.

Procedure

1. Because the **run --local** command reads from the **watches.yaml** file, there are options available to the Operator author. If **role** is left alone (by default, **/opt/ansible/roles/<name>**) you must copy the role over to the **/opt/ansible/roles/** directory from the Operator directly.
This is cumbersome because changes are not reflected from the current directory. Instead, change the **role** field to point to the current directory and comment out the existing line:

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

2. Create a custom resource definition (CRD) and proper role-based access control (RBAC) definitions for the custom resource (CR) **Test1**. The **operator-sdk** command autogenerates these files inside of the **deploy/** directory:

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

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

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

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

3. Run the **run --local** command:

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

Example output

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

4. Now that the Operator is watching the resource **Test1** for events, the creation of a CR triggers your Ansible role to execute. View the **deploy/cr.yaml** file:

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

Because the **spec** field is not set, Ansible is invoked with no extra variables. The next section covers how extra variables are passed from a CR to Ansible. This is why it is important to set reasonable defaults for the Operator.

5. Create a CR instance of **Test1** with the default variable **state** set to **present**:

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

6. Check that the namespace **test** was created:

```
$ oc get namespace
```

Example output

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

7. Modify the **deploy/cr.yaml** file to set the **state** field to **absent**:

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

8. Apply the changes and confirm that the namespace is deleted:

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

```
$ oc get namespace
```

Example output

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

4.2.4.3.2. Testing an Ansible-based Operator on a cluster

After getting familiar running Ansible logic inside of an Ansible-based Operator locally, you can test the Operator inside of a pod on a Kubernetes cluster, such as OpenShift Container Platform. Running as a pod on a cluster is preferred for production use.

Procedure

1. Build the **test1-operator** image and push it to a registry:

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

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

2. Deployment manifests are generated in the **deploy/operator.yaml** file. The deployment image in this file must be modified from the placeholder **REPLACE_IMAGE** to the previously-built image. To do so, run the following command:

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

If you are performing these steps on macOS, use the following command instead:

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

3. Deploy the **test1-operator**:

```
$ oc create -f deploy/crds/test1_v1alpha1_test1_crd.yaml 1
```

1 Only required if the CRD does not exist already.

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

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

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

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

4. Verify that the **test1-operator** is up and running:

```
$ oc get deployment
```

Example output

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

5. You can now view the Ansible logs for the **test1-operator**:

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

4.2.5. Managing custom resource status using the operator_sdk.util Ansible collection

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

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.

Procedure

1. To track CR status manually from your application, update the **watches.yaml** file with a **manageStatus** field set to **false**:

```
- version: v1
```

```
    group: api.example.com
    kind: Test1
    role: Test1
    manageStatus: false
```

2. Use the **operator_sdk.util.k8s_status** Ansible module to update the subresource. For example, to update with key **test1** and value **test2**, **operator_sdk.util** can be used as shown:

```
- operator_sdk.util.k8s_status:
    api_version: app.example.com/v1
    kind: Test1
    name: "{{ meta.name }}"
    namespace: "{{ meta.namespace }}"
    status:
      test1: test2
```

Collections can also be declared in the **meta/main.yml** for the role, which is included for new scaffolded Ansible Operators:

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

Declaring collections in the role meta allows you to invoke the **k8s_status** module directly:

```
k8s_status:
  <snip>
  status:
    test1: test2
```

Additional resources

- For more details about user-driven status management from Ansible-based Operators, see the Ansible-based Operator Status Proposal for Operator SDK .

4.2.6. Additional resources

- See Appendices to learn about the project directory structures created by the Operator SDK.
- Reaching for the Stars with Ansible Operator - Red Hat OpenShift Blog
- Operator Development Guide for Red Hat Partners

4.3. CREATING HELM-BASED OPERATORS

This guide outlines Helm chart support in the Operator SDK and walks Operator authors through an example of building and running an Nginx Operator with the **operator-sdk** CLI tool that uses an existing Helm chart.

4.3.1. Helm chart support in 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. This framework includes the Operator SDK, which assists developers in bootstrapping and building an Operator based on their expertise without

requiring knowledge of Kubernetes API complexities.

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**:

```
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:

```
{{ .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:

```
$ 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.

4.3.2. Installing the Operator SDK CLI

The Operator SDK has a CLI tool that assists developers in creating, building, and deploying a new Operator project. You can install the SDK CLI on your workstation so you are prepared to start authoring your own Operators.

NOTE

This guide uses minikube v0.25.0+ as the local Kubernetes cluster and Quay.io for the public registry.

4.3.2.1. Installing from GitHub release

You can download and install a pre-built release binary of the Operator SDK CLI from the project on GitHub.

Prerequisites

- Go v1.13+
- **docker** v17.03+, **podman** v1.2.0+, or **buildah** v1.7+
- OpenShift CLI (**oc**) v4.5+ installed
- Access to a cluster based on Kubernetes v1.12.0+
- Access to a container registry

Procedure

1. Set the release version variable:

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

2. Download the release binary.
   - For Linux:

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

   - For macOS:

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

3. Verify the downloaded release binary.
   a. Download the provided **.asc** file.
      - For Linux:

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

      - For macOS:

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

   b. Place the binary and corresponding **.asc** file into the same directory and run the following command to verify the binary:
      - For Linux:

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

- For macOS:

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

If you do not have the public key of the maintainer on your workstation, you will get the following error:

Example output with error

```
$ gpg: assuming signed data in 'operator-sdk-${RELEASE_VERSION}-x86_64-apple-darwin'
$ gpg: Signature made Fri Apr  5 20:03:22 2019 CEST
$ gpg:                using RSA key <key_id> 1
$ gpg: Can't check signature: No public key
```

1 RSA key string.

To download the key, run the following command, replacing **<key_id>** with the RSA key string provided in the output of the previous command:

```
$ gpg [--keyserver keys.gnupg.net] --recv-key "<key_id>" 1
```

1 If you do not have a key server configured, specify one with the **--keyserver** option.

4. Install the release binary in your **PATH**:

- For Linux:

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

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

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

- For macOS:

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

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

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

5. Verify that the CLI tool was installed correctly:

```
$ operator-sdk version
```

4.3.2.2. Installing from Homebrew

You can install the SDK CLI using Homebrew.

Prerequisites

- Homebrew
- **docker** v17.03+, **podman** v1.2.0+, or **buildah** v1.7+
- OpenShift CLI (**oc**) v4.5+ installed
- Access to a cluster based on Kubernetes v1.12.0+
- Access to a container registry

Procedure

1. Install the SDK CLI using the **brew** command:

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

2. Verify that the CLI tool was installed correctly:

```
$ operator-sdk version
```

4.3.2.3. Compiling and installing from source

You can obtain the Operator SDK source code to compile and install the SDK CLI.

Prerequisites

- Git
- Go v1.13+
- **docker** v17.03+, **podman** v1.2.0+, or **buildah** v1.7+
- OpenShift CLI (**oc**) v4.5+ installed
- Access to a cluster based on Kubernetes v1.12.0+
- Access to a container registry

Procedure

1. Clone the **operator-sdk** repository:

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

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

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

```
$ cd operator-sdk
```

2. Check out the desired release branch:

```
$ git checkout master
```

3. Compile and install the SDK CLI:

```
$ make dep
```

```
$ make install
```

This installs the CLI binary **operator-sdk** at *$GOPATH/bin*.

4. Verify that the CLI tool was installed correctly:

```
$ operator-sdk version
```

4.3.3. Building a Helm-based Operator using the Operator SDK

This procedure walks through an example of building a simple Nginx Operator powered by a Helm chart using tools and libraries provided by the Operator SDK.

TIP

It is best practice to build a new Operator for each chart. This can allow for more native-behaving Kubernetes APIs (for example, **oc get Nginx**) and flexibility if you ever want to write a fully-fledged Operator in Go, migrating away from a Helm-based Operator.

Prerequisites

- Operator SDK CLI installed on the development workstation
- Access to a Kubernetes-based cluster v1.11.3+ (for example OpenShift Container Platform 4.5) using an account with **cluster-admin** permissions
- OpenShift CLI (**oc**) v4.5+ installed

Procedure

1. **Create a new Operator project.** A namespace-scoped Operator watches and manages resources in a single namespace. Namespace-scoped Operators are preferred because of their flexibility. They enable decoupled upgrades, namespace isolation for failures and monitoring, and differing API definitions.
   To create a new Helm-based, namespace-scoped **nginx-operator** project, use the following command:

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

```
$ cd nginx-operator
```

This creates the **nginx-operator** project specifically for watching the Nginx resource with API version **example.com/v1apha1** and kind **Nginx**.

2. **Customize the Operator logic.**
   For this example, the **nginx-operator** 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 (for example, replica count, image, service type) as specified by the Nginx CR.

   By default, the **nginx-operator** watches **Nginx** resource events as shown in the **watches.yaml** file and executes Helm releases using the specified chart:

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

   a. **Review the Nginx Helm chart.**
   When a Helm Operator project is created, the Operator SDK creates an example 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 Chart developer documentation.

   b. **Understand the Nginx CR 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.

   Override these defaults by setting the desired values in the CR spec. You can use the number of replicas as an example:

      i. First, inspect the **helm-charts/nginx/values.yaml** file to find that the chart has a value called **replicaCount** and it is set to **1** by default. To have 2 Nginx instances in your deployment, your CR spec must contain **replicaCount: 2**.
      Update the **deploy/crds/example.com_v1alpha1_nginx_cr.yaml** file to look like the following:

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

```
  name: example-nginx
spec:
  replicaCount: 2
```

ii. Similarly, the default service port is set to **80**. To instead use **8080**, update the **deploy/crds/example.com_v1alpha1_nginx_cr.yaml** file again by adding the service port override:

```
apiVersion: example.com/v1alpha1
kind: Nginx
metadata:
  name: example-nginx
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 works.

3. **Deploy the CRD.**
Before running the Operator, Kubernetes must know about the new custom resource definition (CRD) that the Operator will be watching. Deploy the following CRD:

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

4. **Build and run the Operator.**
There are two ways to build and run the Operator:

- As a pod inside a Kubernetes cluster.
- As a Go program outside the cluster using the **operator-sdk up** command.

Choose one of the following methods:

a. **Run as a pod**inside a Kubernetes cluster. This is the preferred method for production use.

i. Build the **nginx-operator** image and push it to a registry:

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

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

ii. Deployment manifests are generated in the **deploy/operator.yaml** file. The deployment image in this file needs to be modified from the placeholder **REPLACE_IMAGE** to the previous built image. To do this, run:

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

iii. Deploy the **nginx-operator** manifests:

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

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

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

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

iv. Verify that the **nginx-operator** deployment is up and running:

```
$ oc get deployment
```

Example output

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

b. **Run outside the cluster.** This method is preferred during the development cycle to speed up deployment and testing.
It is important that the chart path referenced in the **watches.yaml** file exists on your machine. By default, the **watches.yaml** file is scaffolded to work with an Operator image built with the **operator-sdk build** command. When developing and testing your Operator with the **operator-sdk run --local** command, the SDK looks in your local file system for this path.

i. Create a symlink at this location to point to the path of your Helm chart:

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

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

ii. To run the Operator locally with the default Kubernetes configuration file present at **$HOME/.kube/config**:

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

To run the Operator locally with a provided Kubernetes configuration file:

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

5. **Deploy the Nginx CR.**
Apply the **Nginx** CR that you modified earlier:

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

Ensure that the **nginx-operator** creates the deployment for the CR:

```
$ oc get deployment
```

Example output

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

Check the pods to confirm two replicas were created:

```
$ oc get pods
```

Example output

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

Check that the service port is set to **8080**:

```
$ oc get service
```

Example output

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

6. **Update the replicaCount and remove the port.**
Change the **spec.replicaCount** field from **2** to **3**, remove the **spec.service** field, and apply the change:

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

Example output

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

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

Confirm that the Operator changes the deployment size:

```
$ oc get deployment
```

Example output

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

Check that the service port is set to the default **80**:

```
$ oc get service
```

Example output

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

7. **Clean up the resources:**

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

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

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

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

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

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

4.3.4. Additional resources

- See Appendices to learn about the project directory structures created by the Operator SDK.
- Operator Development Guide for Red Hat Partners

4.4. GENERATING A CLUSTER SERVICE VERSION (CSV)

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 **generate csv** subcommand to generate a CSV for the current Operator project customized using information contained in manually-defined 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.

The CSV version is the same as the Operator version, and a new CSV is generated when upgrading Operator versions. Operator authors can use the **--csv-version** flag to have their Operator state encapsulated in a CSV with the supplied semantic version:

```
$ operator-sdk generate csv --csv-version <version>
```

This action is idempotent and only updates the CSV file when a new version is supplied, or a YAML manifest or source file is changed. Operator authors should not have to directly modify most fields in a CSV manifest. Those that require modification are defined in this guide. For example, the CSV version must be included in **metadata.name**.

4.4.1. How CSV generation works

The **deploy**/ directory of an Operator project is the standard location for all manifests required to deploy an Operator. The Operator SDK can use data from manifests in **deploy**/ to write a cluster service version (CSV).

The following command:

```
$ operator-sdk generate csv --csv-version <version>
```

writes a CSV YAML file to the **deploy**/**olm-catalog**/ directory by default.

Exactly three types of manifests are required to generate a CSV:

- **operator.yaml**
- ***_{crd,cr}.yaml**
- RBAC role files, for example **role.yaml**

Operator authors may have different versioning requirements for these files and can configure which specific files are included in the **deploy**/**olm-catalog**/**csv-config.yaml** file.

Workflow
Depending on whether an existing CSV is detected, and assuming all configuration defaults are used, the **generate csv** subcommand either:

- Creates a new CSV, with the same location and naming convention as exists currently, using available data in YAML manifests and source files.
    a. The update mechanism checks for an existing CSV in **deploy**/. When one is not found, it creates a **ClusterServiceVersion** object, referred to here as a *cache*, and populates fields easily derived from Operator metadata, such as Kubernetes API **ObjectMeta**.
    b. The update mechanism searches **deploy**/ for manifests that contain data a CSV uses, such as a **Deployment** resource, and sets the appropriate CSV fields in the cache with this data.
    c. After the search completes, every cache field populated is written back to a CSV YAML file.

or:

- Updates an existing CSV at the currently pre-defined location, using available data in YAML manifests and source files.
    a. The update mechanism checks for an existing CSV in **deploy**/. When one is found, the CSV YAML file contents are marshaled into a CSV cache.
    b. The update mechanism searches **deploy**/ for manifests that contain data a CSV uses, such as a **Deployment** resource, and sets the appropriate CSV fields in the cache with this data.
    c. After the search completes, every cache field populated is written back to a CSV YAML file.

NOTE

Individual YAML fields are overwritten and not the entire file, as descriptions and other non-generated parts of a CSV should be preserved.

4.4.2. CSV composition configuration

Operator authors can configure CSV composition by populating several fields in the **deploy/olm-catalog/csv-config.yaml** file:

Field	Description
operator-path (string)	The Operator resource manifest file path. Default: **deploy/operator.yaml**.
crd-cr-path-list (string(, string)*)	A list of CRD and CR manifest file paths. Default: **[deploy/crds/*_{crd,cr}.yaml]**.
rbac-path-list (string(, string)*)	A list of RBAC role manifest file paths. Default: **[deploy/role.yaml]**.

4.4.3. 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, English 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.

Table 4.5. Required

Field	Description
metadata.name	A unique name for this CSV. Operator version should be included in the name to ensure uniqueness, for example **app-operator.v0.1.1**.
metadata.capabilities	The capability level according to the Operator maturity model. Options include **Basic Install**, **Seamless Upgrades**, **Full Lifecycle**, **Deep Insights**, and **Auto Pilot**.
spec.displayName	A public name to identify the Operator.
spec.description	A short description of the functionality of the Operator.
spec.keywords	Keywords describing the Operator.

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 4.6. 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

- Operator maturity model

4.4.4. Generating a CSV

Prerequisites

- An Operator project generated using the Operator SDK

Procedure

1. In your Operator project, configure your CSV composition by modifying the **deploy/olm-catalog/csv-config.yaml** file, if desired.
2. Generate the CSV:

```
$ operator-sdk generate csv --csv-version <version>
```

3. In the new CSV generated in the **deploy/olm-catalog/** directory, ensure all required, manually-defined fields are set appropriately.

4.4.5. Enabling your Operator for restricted network environments

As an Operator author, your CSV must meet the following additional requirements for your Operator to run properly in a restricted network environment:

- 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.

You must use SHA references to related images in two places in the Operator's CSV:

- in **spec.relatedImages**:

```
...
spec:
  relatedImages: 1
    - name: etcd-operator 2
      image: quay.io/etcd-
operator/operator@sha256:d134a9865524c29fcf75bbc4469013bc38d8a15cb5f41acfddb6b9e4
92f556e4 3
    - name: etcd-image
      image: quay.io/etcd-
operator/etcd@sha256:13348c15263bd8838ec1d5fc4550ede9860fcbb0f843e48cbccec07810e
ebb68
...
```

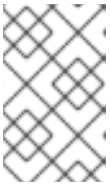
1. Create a **relatedImages** section and set the list of related images.
2. Specify a unique identifier for the image.
3. Specify each image by a digest (SHA), not by an image tag.

- in the **env** section of the Operators Deployments when declaring environment variables that inject the image that the Operator should use:

```
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@sha256:13348c15263bd8838ec1d5fc4550ede9860fcbb0f843e48cbccec07810e
ebb68 (2)
                - name: ETCD_LOG_LEVEL
                  value: INFO
                image: quay.io/etcd-
operator/operator@sha256:d134a9865524c29fcf75bbc4469013bc38d8a15cb5f41acfddb6b9e4
92f556e4 (3)
                imagePullPolicy: IfNotPresent
                livenessProbe:
                  httpGet:
                    path: /healthy
                    port: 8080
                  initialDelaySeconds: 10
                  periodSeconds: 30
                name: etcd-operator
                readinessProbe:
                  httpGet:
                    path: /ready
                    port: 8080
                  initialDelaySeconds: 10
                  periodSeconds: 30
                resources: {}
              serviceAccountName: etcd-operator
    strategy: deployment
```

1. 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**.

- Look for the **Disconnected** annotation, which indicates that the Operator works in a disconnected environment:

```
metadata:
  annotations:
    operators.openshift.io/infrastructure-features: '["Disconnected"]'
```

Operators can be filtered in OperatorHub by this infrastructure feature.

4.4.6. 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:

```
labels:
    operatorframework.io/arch.<arch>: supported 1
    operatorframework.io/os.<os>: supported 2
```

1 Set **<arch>** to a supported string.

2 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:

```
labels:
    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:

```
labels:
    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:

```
labels:
  operatorframework.io/arch.s390x: supported
  operatorframework.io/os.zos: supported
  operatorframework.io/os.linux: supported (1)
  operatorframework.io/arch.amd64: supported (2)
```

(1) (2) 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.

4.4.6.1. 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 4.7. Architectures supported on OpenShift Container Platform

Architecture	String
AMD64	**amd64**
64-bit PowerPC little-endian	**ppc64le**
IBM Z	**s390x**

Table 4.8. Operating systems supported on OpenShift Container Platform

Operating system	String
Linux	**linux**
z/OS	**zos**

NOTE

Different versions of OpenShift Container Platform and other Kubernetes-based distributions might support a different set of architectures and operating systems.

4.4.7. Setting a suggested namespace

Some Operators must be deployed in a specific namespace, or with ancillary resources in specific namespaces, in order 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:

```
metadata:
  annotations:
    operatorframework.io/suggested-namespace: <namespace> 1
```

1 Set your suggested namespace.

4.4.8. 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*.

4.4.8.1. 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 4.9. Owned CRD fields

Field	Description	Required/optional
Name	The full name of your CRD.	Required
Version	The version of that object API.	Required
Kind	The machine readable name of your CRD.	Required
DisplayName	A human readable version of your CRD name, for example **MongoDB Standalone**.	Required
Description	A short description of how this CRD is used by the Operator or a description of the functionality provided by the CRD.	Required
Group	The API group that this CRD belongs to, for example **database.example.com**.	Optional
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.	Optional

Field	Description	Required/optional
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.	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
  group: mongodb.com
  kind: MongoDbStandalone
  name: mongodbstandalones.mongodb.com
  resources:
    - kind: Service
      name: ''
      version: v1
    - kind: StatefulSet
      name: ''
      version: v1beta2
```

```
    - kind: Pod
      name: ''
      version: v1
    - kind: ConfigMap
      name: ''
      version: v1
  specDescriptors:
    - description: Credentials for Ops Manager or Cloud Manager.
      displayName: Credentials
      path: credentials
      x-descriptors:
        - 'urn:alm:descriptor:com.tectonic.ui:selector:core:v1:Secret'
    - description: Project this deployment belongs to.
      displayName: Project
      path: project
      x-descriptors:
        - 'urn:alm:descriptor:com.tectonic.ui:selector:core:v1:ConfigMap'
    - description: MongoDB version to be installed.
      displayName: Version
      path: version
      x-descriptors:
        - 'urn:alm:descriptor:com.tectonic.ui:label'
  statusDescriptors:
    - description: The status of each of the pods for the MongoDB cluster.
      displayName: Pod Status
      path: pods
      x-descriptors:
        - 'urn:alm:descriptor:com.tectonic.ui:podStatuses'
  version: v1
  description: >-
    MongoDB Deployment consisting of only one host. No replication of
    data.
```

4.4.8.2. 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 4.10. Required CRD fields

Field	Description	Required/optional
Name	The full name of the CRD you require.	Required
Version	The version of that object API.	Required

Field	Description	Required/optional
Kind	The Kubernetes object kind.	Required
DisplayName	A human readable version of the CRD.	Required
Description	A summary of how the component fits in your larger architecture.	Required

Example required CRD

```
required:
- name: etcdclusters.etcd.database.coreos.com
  version: v1beta2
  kind: EtcdCluster
  displayName: etcd Cluster
  description: Represents a cluster of etcd nodes.
```

4.4.8.3. CRD 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**:

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

4.4.8.4. 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

```
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.

4.4.9. Understanding your API services

As with CRDs, there are two types of API services that your Operator may use: *owned* and *required*.

4.4.9.1. 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 4.11. Owned API service fields

Field	Description	Required/optional
Group	Group that the API service provides, for example **database.example.com**.	Required
Version	Version of the API service, for example **v1alpha1**.	Required
Kind	A kind that the API service is expected to provide.	Required
Name	The plural name for the API service provided.	Required

Field	Description	Required/optional
DeploymentName	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 "Install Ready" phase.	Required
DisplayName	A human readable version of your API service name, for example **MongoDB Standalone**.	Required
Description	A short description of how this API service is used by the Operator or a description of the functionality provided by the API service.	Required
Resources	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.	Optional
SpecDescriptors, **StatusDescriptors**, and **ActionDescriptors**	Essentially the same as for owned CRDs.	Optional

4.4.9.1.1. 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.
- A new CA key/certificate pair is generated for each installation and the base64-encoded CA bundle is embedded in the respective API service resource.

4.4.9.1.2. 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 host name 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.

4.4.9.2. 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 4.12. Required API service fields

Field	Description	Required/optional
Group	Group that the API service provides, for example **database.example.com**.	Required
Version	Version of the API service, for example **v1alpha1**.	Required
Kind	A kind that the API service is expected to provide.	Required
DisplayName	A human readable version of your API service name, for example **MongoDB Standalone**.	Required
Description	A short description of how this API service is used by the Operator or a description of the functionality provided by the API service.	Required

4.5. WORKING WITH BUNDLE IMAGES

You can use the Operator SDK to package Operators using the Bundle Format.

4.5.1. Building a bundle image

You can build, push, and validate an Operator bundle image using the Operator SDK.

Prerequisites

- Operator SDK version 0.17.2
- **podman** version 1.4.4+
- An Operator project generated using the Operator SDK

Procedure

1. From your Operator project directory, build the bundle image using the Operator SDK:

```
$ operator-sdk bundle create \
    quay.io/<namespace>/test-operator:v0.1.0 \ 1
    -b podman 2
```

1. The image tag that you want the bundle image to have.
2. The CLI tool to use for building the container image, either **docker** (default), **podman**, or **buildah**. This example uses **podman**.

NOTE

If your local manifests are not located in the default **<project_root>/deploy/olm-catalog/test-operator/manifests**, specify the location with the **--directory** flag.

2. Log in to the registry where you want to push the bundle image. For example:

```
$ podman login quay.io
```

3. Push the bundle image to the registry:

```
$ podman push quay.io/<namespace>/test-operator:v0.1.0
```

4. Validate the bundle image in the remote registry:

```
$ operator-sdk bundle validate \
    quay.io/<namespace>/test-operator:v0.1.0 \
    -b podman
```

Example output

```
INFO[0000] Unpacked image layers                         bundle-dir=/tmp/bundle-041168359
container-tool=podman
INFO[0000] running podman pull                           bundle-dir=/tmp/bundle-041168359
container-tool=podman
INFO[0002] running podman save                           bundle-dir=/tmp/bundle-041168359
container-tool=podman
INFO[0002] All validation tests have completed successfully      bundle-dir=/tmp/bundle-
041168359 container-tool=podman
```

4.5.2. Additional resources

- See Operator Framework packaging formats for more details on the Bundle Format.

4.6. VALIDATING OPERATORS USING THE SCORECARD

Operator authors should validate that their Operator is packaged correctly and free of syntax errors. As an Operator author, you can use the Operator SDK scorecard tool to validate your Operator packaging and run tests.

NOTE

OpenShift Container Platform 4.5 supports Operator SDK v0.17.2.

4.6.1. About the scorecard tool

To validate an Operator, the scorecard tool provided by the Operator SDK begins by creating all resources required by any related custom resources (CRs) and the Operator. The scorecard then creates a proxy container in the deployment of the Operator which is used to record calls to the API server and run some of the tests. The tests performed also examine some of the parameters in the CRs.

4.6.2. Scorecard configuration

The scorecard tool uses a configuration file that allows you to configure internal plug-ins, as well as several global configuration options.

4.6.2.1. Configuration file

The default location for the scorecard tool configuration is the **<project_dir>/.osdk-scorecard.***. The following is an example of a YAML-formatted configuration file:

Scorecard configuration file

```
scorecard:
  output: json
  plugins:
    - basic: 1
        cr-manifest:
          - "deploy/crds/cache.example.com_v1alpha1_memcached_cr.yaml"
          - "deploy/crds/cache.example.com_v1alpha1_memcachedrs_cr.yaml"
    - olm: 2
        cr-manifest:
          - "deploy/crds/cache.example.com_v1alpha1_memcached_cr.yaml"
          - "deploy/crds/cache.example.com_v1alpha1_memcachedrs_cr.yaml"
        csv-path: "deploy/olm-catalog/memcached-operator/0.0.3/memcached-
operator.v0.0.3.clusterserviceversion.yaml"
```

1 **basic** tests configured to test two custom resources (CRs).

2 **olm** tests configured to test two CRs.

Configuration methods for global options take the following priority, highest to lowest:

Command arguments (if available) → configuration file → default

The configuration file must be in YAML format. As the configuration file might be extended to allow configuration of all **operator-sdk** subcommands in the future, the scorecard configuration must be under a **scorecard** subsection.

NOTE

Configuration file support is provided by the **viper** package. For more info on how **viper** configuration works, see the README.

4.6.2.2. Command arguments

While most of the scorecard tool configuration is done using a configuration file, you can also use the following arguments:

Table 4.13. Scorecard tool arguments

Flag	Type	Description
--bundle, **-b**	string	The path to a bundle directory used for the bundle validation test.
--config	string	The path to the scorecard configuration file. The default is **<project_dir>/.osdk-scorecard**. The file type and extension must be **.yaml**. If a configuration file is not provided or found at the default location, the scorecard exits with an error.
--output, **-o**	string	Output format. Valid options are **text** and **json**. The default format is **text**, which is designed to be a human readable format. The **json** format uses the JSON schema output format used for plug-ins defined later.
--kubeconfig, **-o**	string	The path to the **kubeconfig** file. It sets the**kubeconfig** for internal plug-ins.
--version	string	The version of scorecard to run. The default and only valid option is **v1alpha2**.
--selector, **-l**	string	The label selector to filter tests on.
--list, **-L**	bool	If **true**, only print the test names that would be run based on selector filtering.

4.6.2.3. Configuration file options

The scorecard configuration file provides the following options:

Table 4.14. Scorecard configuration file options

Option	Type	Description
bundle	string	Equivalent of the **--bundle** flag. Operator Lifecycle Manager (OLM) bundle directory path, when specified, runs bundle validation.
output	string	Equivalent of the **--output** flag. If this option is defined by both the configuration file and the flag, the flag value takes priority.

Option	Type	Description
kubeconfig	string	Equivalent of the **--kubeconfig** flag. If this option is defined by both the configuration file and the flag, the flag value takes priority.
plugins	array	An array of plug-in names.

4.6.2.3.1. Basic and OLM plug-ins

The scorecard supports the internal **basic** and **olm** plug-ins, which are configured by a **plugins** section in the configuration file.

Table 4.15. Plug-in options

Option	Type	Description
cr-manifest	[]string	The path(s) for CRs being tested. Required if **olm-deployed** is unset or **false**.
csv-path	string	The path to the cluster service version (CSV) for the Operator. Required for OLM tests or if **olm-deployed** is set to **true**.
olm-deployed	bool	Indicates that the CSV and relevant CRDs have been deployed onto the cluster by OLM.
kubeconfig	string	The path to the **kubeconfig** file. If both the global **kubeconfig** and this field are set, this field is used for the plug-in.
namespace	string	The namespace to run the plug-ins in. If unset, the default specified by the **kubeconfig** file is used.
init-timeout	int	Time in seconds until a timeout during initialization of the Operator.
crds-dir	string	The path to the directory containing CRDs that must be deployed to the cluster.
namespaced-manifest	string	The manifest file with all resources that run within a namespace. By default, the scorecard combines the **service_account.yaml**, **role.yaml**, **role_binding.yaml**, and **operator.yaml** files from the **deploy** directory into a temporary manifest to use as the namespaced manifest.

Option	Type	Description
global-manifest	string	The manifest containing required resources that run globally (not namespaced). By default, the scorecard combines all CRDs in the **crds-dir** directory into a temporary manifest to use as the global manifest.

NOTE

Currently, using the scorecard with a CSV does not permit multiple CR manifests to be set through the CLI, configuration file, or CSV annotations. You must tear down your Operator in the cluster, re-deploy, and re-run the scorecard for each CR that is tested.

Additional resources

- You can either set **cr-manifest** or your CSV **metadata.annotations['alm-examples']** to provide CRs to the scorecard, but not both. See CRD templates for details.

4.6.3. Tests performed

By default, the scorecard tool has a set of internal tests it can run available across two internal plug-ins. If multiple CRs are specified for a plug-in, the test environment is fully cleaned up after each CR so that each CR gets a clean testing environment.

Each test has a short name that uniquely identifies the test. This is useful when selecting a specific test or tests to run. For example:

```
$ operator-sdk scorecard -o text --selector=test=checkspectest
```

```
$ operator-sdk scorecard -o text --selector='test in (checkspectest,checkstatustest)'
```

4.6.3.1. Basic plug-in

The following basic Operator tests are available from the **basic** plug-in:

Table 4.16. basic plug-in tests

Test	Description	Short name
Spec Block Exists	This test checks the custom resources (CRs) created in the cluster to make sure that all CRs have a **spec** block. This test has a maximum score of **1**.	**checkspectest**
Status Block Exists	This test checks the CRs created in the cluster to make sure that all CRs have a **status** block. This test has a maximum score of **1**.	**checkstatustest**

Test	Description	Short name
Writing Into CRs Has An Effect	This test reads the scorecard proxy logs to verify that the Operator is making **PUT** or **POST**, or both, requests to the API server, indicating that it is modifying resources. This test has a maximum score of **1**.	**writingintocrshaseffecttest**

4.6.3.2. OLM plug-in

The following Operator Lifecycle Manager (OLM) integration tests are available from the **olm** plug-in:

Table 4.17. olm plug-in tests

Test	Description	Short name
OLM Bundle Validation	This test validates the OLM bundle manifests found in the bundle directory as specified by the bundle flag. If the bundle contents contain errors, then the test result output includes the validator log as well as error messages from the validation library.	**bundlevalidationtest**
Provided APIs Have Validation	This test verifies that the CRDs for the provided CRs contain a validation section and that there is validation for each **spec** and **status** field detected in the CR. This test has a maximum score equal to the number of CRs provided by the **cr-manifest** option.	**crdshavevalidationtest**
Owned CRDs Have Resources Listed	This test makes sure that the CRDs for each CR provided by the **cr-manifest** option have a**resources** subsection in the **owned** CRDs section of the 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. This test has a maximum score equal to the number of CRs provided by the **cr-manifest** option.	**crdshaveresourcestest**
Spec Fields With Descriptors	This test verifies that every field in the **spec** sections of custom resources have a corresponding descriptor listed in the CSV. This test has a maximum score equal to the total number of fields in the **spec** sections of each custom resource passed in by the **cr-manifest** option.	**specdescriptorstest**
Status Fields With Descriptors	This test verifies that every field in the **status** sections of custom resources have a corresponding descriptor listed in the CSV. This test has a maximum score equal to the total number of fields in the **status** sections of each custom resource passed in by the **cr-manifest** option.	**statusdescriptorstest**

Additional resources

- Owned CRDs

4.6.4. Running the scorecard

Prerequisites

The following prerequisites for the Operator project are checked by the scorecard tool:

- Access to a cluster running Kubernetes 1.11.3 or later.
- If you want to use the scorecard to check the integration of your Operator project with Operator Lifecycle Manager (OLM), then a cluster service version (CSV) file is also required. This is a requirement when the **olm-deployed** option is used.
- For Operators that were not generated using the Operator SDK (non-SDK Operators):
  - Resource manifests for installing and configuring the Operator and custom resources (CRs).
  - Configuration getter that supports reading from the **KUBECONFIG** environment variable, such as the **clientcmd** or **controller-runtime** configuration getters. This is required for the scorecard proxy to work correctly.

Procedure

1. Define a **.osdk-scorecard.yaml** configuration file in your Operator project.
2. Create the namespace defined in the RBAC files (**role_binding**).
3. Run the scorecard from the root directory of your Operator project:

   ```
   $ operator-sdk scorecard
   ```

   The scorecard return code is **1** if any of the executed texts did not pass and **0** if all selected tests passed.

4.6.5. Running the scorecard with an OLM-managed Operator

The scorecard can be run using a cluster service version (CSV), providing a way to test cluster-ready and non-Operator SDK Operators.

Procedure

1. The scorecard requires a proxy container in the deployment pod of the Operator to read Operator logs. A few modifications to your CSV and creation of one extra object are required to run the proxy *before* deploying your Operator with Operator Lifecycle Manager (OLM).
   This step can be performed manually or automated using bash functions. Choose one of the following methods.
   - **Manual method:**
     a. Create a proxy server secret containing a local **kubeconfig** file`.
        i. Generate a user name using the namespaced owner reference of the scorecard proxy.

```
$ echo
'{"apiVersion":"","kind":"","name":"scorecard","uid":"","Namespace":"'<namespace>
'"}' | base64 -w 0 (1)
```

(1) Replace **<namespace>** with the namespace your Operator will deploy in.

ii. Write a **Config** manifest **scorecard-config.yaml** using the following template, replacing **<username>** with the base64 user name generated in the previous step:

```
apiVersion: v1
kind: Config
clusters:
- cluster:
    insecure-skip-tls-verify: true
    server: http://<username>@localhost:8889
  name: proxy-server
contexts:
- context:
    cluster: proxy-server
    user: admin/proxy-server
  name: <namespace>/proxy-server
current-context: <namespace>/proxy-server
preferences: {}
users:
- name: admin/proxy-server
  user:
    username: <username>
    password: unused
```

iii. Encode the **Config** as base64:

```
$ cat scorecard-config.yaml | base64 -w 0
```

iv. Create a **Secret** manifest **scorecard-secret.yaml**:

```
apiVersion: v1
kind: Secret
metadata:
  name: scorecard-kubeconfig
  namespace: <namespace> (1)
data:
  kubeconfig: <kubeconfig_base64> (2)
```

(1) Replace **<namespace>** with the namespace your Operator will deploy in.

(2) Replace **<kubeconfig_base64>** with the **Config** encoded as base64.

v. Apply the secret:

```
$ oc apply -f scorecard-secret.yaml
```

vi. Insert a volume referring to the secret into the deployment for the Operator:

```
spec:
  install:
    spec:
      deployments:
      - name: memcached-operator
        spec:
          ...
          template:
            ...
            spec:
              containers:
              ...
              volumes:
              - name: scorecard-kubeconfig (1)
                secret:
                  secretName: scorecard-kubeconfig
                  items:
                  - key: kubeconfig
                    path: config
```

(1) Scorecard **kubeconfig** volume.

b. Insert a volume mount and **KUBECONFIG** environment variable into each container in the deployment of your Operator:

```
spec:
  install:
    spec:
      deployments:
      - name: memcached-operator
        spec:
          ...
          template:
            ...
            spec:
              containers:
              - name: container1
                ...
                volumeMounts:
                - name: scorecard-kubeconfig (1)
                  mountPath: /scorecard-secret
                env:
                - name: KUBECONFIG (2)
                  value: /scorecard-secret/config
              - name: container2 (3)
                ...
```

(1) Scorecard **kubeconfig** volume mount.

(2) Scorecard **kubeconfig** environment variable.

(3) Repeat the same for this and all other containers.

c. Insert the scorecard proxy container into the deployment of your Operator:

```
spec:
  install:
    spec:
      deployments:
      - name: memcached-operator
        spec:
          ...
          template:
            ...
            spec:
              containers:
              ...
              - name: scorecard-proxy (1)
                command:
                - scorecard-proxy
                env:
                - name: WATCH_NAMESPACE
                  valueFrom:
                    fieldRef:
                      apiVersion: v1
                      fieldPath: metadata.namespace
                image: quay.io/operator-framework/scorecard-proxy:master
                imagePullPolicy: Always
                ports:
                - name: proxy
                  containerPort: 8889
```

(1) Scorecard proxy container.

- **Automated method:**
  The **community-operators** repository has several bash functions that can perform the previous steps in the procedure for you.

  a. Run the following **curl** command:

  ```
  $ curl -Lo csv-manifest-modifiers.sh \
      https://raw.githubusercontent.com/operator-framework/community-operators/master/scripts/lib/file
  ```

  b. Source the **csv-manifest-modifiers.sh** file:

  ```
  $ . ./csv-manifest-modifiers.sh
  ```

  c. Create the **kubeconfig** secret file:

  ```
  $ create_kubeconfig_secret_file scorecard-secret.yaml "<namespace>" (1)
  ```

  (1) Replace **<namespace>** with the namespace your Operator will deploy in.

  d. Apply the secret:

```
$ oc apply -f scorecard-secret.yaml
```

e. Insert the **kubeconfig** volume:

```
$ insert_kubeconfig_volume "<csv_file>" 1
```

1 Replace **<csv_file>** with the path to your CSV manifest.

f. Insert the **kubeconfig** secret mount:

```
$ insert_kubeconfig_secret_mount "<csv_file>"
```

g. Insert the proxy container:

```
$ insert_proxy_container "<csv_file>" "quay.io/operator-framework/scorecard-proxy:master"
```

2. After inserting the proxy container, follow the steps in the *Getting started with the Operator SDK* guide to bundle your CSV and custom resource definitions (CRDs) and deploy your Operator on OLM.

3. After your Operator has been deployed on OLM, define a **.osdk-scorecard.yaml** configuration file in your Operator project and ensure both the **csv-path: <csv_manifest_path>** and **olm-deployed** options are set.

4. Run the scorecard with both the **csv-path: <csv_manifest_path>** and **olm-deployed** options set in your scorecard configuration file:

```
$ operator-sdk scorecard
```

Additional resources

- Managing a Go-based Operator using Operator Lifecycle Manager

4.7. 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.

4.7.1. 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.

4.7.2. Metrics helper

In Go-based Operators generated using the Operator SDK, the following function exposes general metrics about the running program:

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

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:

```
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.

4.7.2.1. 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
```

4.7.3. 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).

4.7.3.1. 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.
    services := []*v1.Service{}
    // Create one ServiceMonitor per application per namespace.
    // Change the below value to name of the Namespace you want the ServiceMonitor to be
created in.
    ns := "default"
    // restConfig is used for talking to the Kubernetes apiserver
    restConfig := config.GetConfig()

    // Pass the Service(s) to the helper function, which in turn returns the array of
ServiceMonitor objects.
    serviceMonitors, err := metrics.CreateServiceMonitors(restConfig, ns, services)
```

```
    if err != nil {
        // Handle errors here.
    }
    ...
}
```

4.8. 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.

The following examples illustrate how to use the two options.

4.8.1. Using 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**:

```
import (
  ...
  "github.com/operator-framework/operator-sdk/pkg/leader"
)

func main() {
  ...
  err = leader.Become(context.TODO(), "memcached-operator-lock")
  if err != nil {
    log.Error(err, "Failed to retry for leader lock")
    os.Exit(1)
  }
  ...
}
```

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.

4.8.2. Using Leader-with-lease election

The Leader-with-lease implementation can be enabled using the Manager Options for leader election:

```
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 in order to create the config map for leader election. You can override this namespace by setting the **LeaderElectionNamespace** option for the Manager.

4.9. OPERATOR SDK CLI REFERENCE

This guide documents the Operator SDK CLI commands and their syntax:

```
$ operator-sdk <command> [<subcommand>] [<argument>] [<flags>]
```

4.9.1. build

The **operator-sdk build** command compiles the code and builds the executables. After **build** completes, the image is built using a local container engine. It must then be pushed to a remote registry.

Table 4.18. build arguments

Argument	Description
<image>	The container image to be built, for example **quay.io/example/operator:v0.0.1**.

Table 4.19. build flags

Flag	Description
--enable-tests (bool)	Enable in-cluster testing by adding test binary to the image.

Flag	Description
--namespaced-manifest (string)	Path of namespaced resources manifest for tests. Default: **deploy/operator.yaml**.
--test-location (string)	Location of tests. Default: **./test/e2e**.
-h, --help	Usage help output.

If **--enable-tests** is set, the **build** command also builds the testing binary, adds it to the container image, and generates a **deploy/test-pod.yaml** file that allows a user to run the tests as a pod on a cluster.

For example:

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

Example output

```
building example-operator...

building container quay.io/example/operator:v0.0.1...
Sending build context to Docker daemon  163.9MB
Step 1/4 : FROM alpine:3.6
 ---> 77144d8c6bdc
Step 2/4 : ADD tmp/_output/bin/example-operator /usr/local/bin/example-operator
 ---> 2ada0d6ca93c
Step 3/4 : RUN adduser -D example-operator
 ---> Running in 34b4bb507c14
Removing intermediate container 34b4bb507c14
 ---> c671ec1cff03
Step 4/4 : USER example-operator
 ---> Running in bd336926317c
Removing intermediate container bd336926317c
 ---> d6b58a0fcb8c
Successfully built d6b58a0fcb8c
Successfully tagged quay.io/example/operator:v0.0.1
```

4.9.2. completion

The **operator-sdk completion** command generates shell completions to make issuing CLI commands quicker and easier.

Table 4.20. completion subcommands

Subcommand	Description
bash	Generate bash completions.
zsh	Generate zsh completions.

Table 4.21. **completion** flags

Flag	Description
-h, --help	Usage help output.

For example:

```
$ operator-sdk completion bash
```

Example output

```
# bash completion for operator-sdk                -*- shell-script -*-
...
# ex: ts=4 sw=4 et filetype=sh
```

4.9.3. print-deps

The **operator-sdk print-deps** command prints the most recent Golang packages and versions required by Operators. It prints in columnar format by default.

Table 4.22. **print-deps** flags

Flag	Description
--as-file	Print packages and versions in **Gopkg.toml** format.

For example:

```
$ operator-sdk print-deps --as-file
```

Example output

```
required = [
  "k8s.io/code-generator/cmd/defaulter-gen",
  "k8s.io/code-generator/cmd/deepcopy-gen",
  "k8s.io/code-generator/cmd/conversion-gen",
  "k8s.io/code-generator/cmd/client-gen",
  "k8s.io/code-generator/cmd/lister-gen",
  "k8s.io/code-generator/cmd/informer-gen",
  "k8s.io/code-generator/cmd/openapi-gen",
  "k8s.io/gengo/args",
]

[[override]]
  name = "k8s.io/code-generator"
  revision = "6702109cc68eb6fe6350b83e14407c8d7309fd1a"
  ...
```

4.9.4. generate

The **operator-sdk generate** command invokes a specific generator to generate code as needed.

4.9.4.1. crds

The **generate crds** subcommand generates custom resource definitions (CRDs) or updates them if they exist, under **deploy/crds/__crd.yaml**. OpenAPI V3 validation YAML is generated as a **validation** object.

Table 4.23. generate crds flags

Flag	Description
--crd-version (string)	CRD version to generate. Default: **v1beta1**
-h, **--help**	Help for **generate crds**

For example:

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

```
$ tree deploy/crds
```

Example output

```
├── deploy/crds/app.example.com_v1alpha1_appservice_cr.yaml
└── deploy/crds/app.example.com_appservices_crd.yaml
```

4.9.4.2. csv

The **csv** subcommand writes a cluster service version (CSV) manifest for use with Operator Lifecycle Manager (OLM). It also optionally writes CRD files to **deploy/olm-catalog/<operator_name>/<csv_version>**.

Table 4.24. generate csv flags

Flag	Description
--csv-channel (string)	The channel the CSV should be registered under in the package manifest.
--csv-config (string)	The path to the CSV configuration file. Default: **deploy/olm-catalog/csv-config.yaml**.
--csv-version (string)	The semantic version of the CSV manifest. Required.
--default-channel	Use the channel passed to **--csv-channel** as the default channel of the package manifests. Only valid when **--csv-channel** is set.
--from-version (string)	The semantic version of CSV manifest to use as a base for a new version.

Flag	Description
--operator-name	The Operator name to use while generating the CSV.
--update-crds	Updates CRD manifests in **deploy/<operator_name>/<csv_version>** using the latest CRD manifests.

For example:

```
$ operator-sdk generate csv \
    --csv-version 0.1.0 \
  --update-crds
```

Example output

```
INFO[0000] Generating CSV manifest version 0.1.0
INFO[0000] Fill in the following required fields in file deploy/olm-catalog/operator-
name/0.1.0/operator-name.v0.1.0.clusterserviceversion.yaml:
 spec.keywords
 spec.maintainers
 spec.provider
 spec.labels
INFO[0000] Created deploy/olm-catalog/operator-name/0.1.0/operator-
name.v0.1.0.clusterserviceversion.yaml
```

4.9.4.3. k8s

The **k8s** subcommand runs the Kubernetes code-generators for all CRD APIs under **pkg/apis/**. Currently, **k8s** only runs **deepcopy-gen** to generate the required **DeepCopy()** functions for all custom resource (CR) types.

NOTE

This command must be run every time the API (**spec** and **status**) for a custom resource type is updated.

For example:

```
$ tree pkg/apis/app/v1alpha1/
```

Example output

```
pkg/apis/app/v1alpha1/
├── appservice_types.go
├── doc.go
└── register.go
```

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

Example output

```
Running code-generation for Custom Resource (CR) group versions: [app:v1alpha1]
Generating deepcopy funcs
```

```
$ tree pkg/apis/app/v1alpha1/
```

Example output

```
pkg/apis/app/v1alpha1/
├── appservice_types.go
├── doc.go
├── register.go
└── zz_generated.deepcopy.go
```

4.9.5. new

The **operator-sdk new** command creates a new Operator application and generates (or *scaffolds*) a default project directory layout based on the input **<project_name>**.

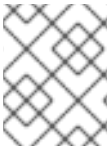
Table 4.25. new arguments

Argument	Description
<project_name>	Name of the new project.

Table 4.26. new flags

Flag	Description
--api-version	CRD API version in the format **<group_name>/<version>**, for example **app.example.com/v1alpha1**. Used with **ansible** or **helm** types.
--generate-playbook	Generate an Ansible playbook skeleton. Used with **ansible** type.
--header-file <string>	Path to file containing headers for generated Go files. Copied to **hack/boilerplate.go.txt**.
--helm-chart <string>	Initialize Helm Operator with existing Helm chart: **<url>**, **<repo>/<name>**, or local path.
--helm-chart-repo <string>	Chart repository URL for the requested Helm chart.
--helm-chart-version <string>	Specific version of the Helm chart. Default: latest version.
--help, -h	Usage and help output.
--kind <string>	CRD kind, for example **AppService**. Used with **ansible** or **helm** types.

Flag	Description
--skip-git-init	Do not initialize the directory as a Git repository.
--type	Type of Operator to initialize: **go**, **ansible** or **helm**. Default: **go**.



NOTE

Starting with Operator SDK v0.12.0, the **--dep-manager** flag and support for **dep**-based projects have been removed. Go projects are now scaffolded to use Go modules.

Example usage for Go project

```
$ mkdir $GOPATH/src/github.com/example.com/
```

```
$ cd $GOPATH/src/github.com/example.com/
```

```
$ operator-sdk new app-operator
```

Example usage for Ansible project

```
$ operator-sdk new app-operator \
    --type=ansible \
    --api-version=app.example.com/v1alpha1 \
    --kind=AppService
```

4.9.6. add

The **operator-sdk add** command adds a controller or resource to the project. The command must be run from the Operator project root directory.

Table 4.27. add subcommands

Subcommand	Description
api	Adds a new API definition for a new custom resource (CR) under **pkg/apis** and generates the custom resource definition (CRD) and CR files under **deploy/crds/**. If the API already exists at **pkg/apis/<group>/<version>**, then the command does not overwrite and returns an error.
controller	Adds a new controller under **pkg/controller/<kind>/**. The controller expects to use the CR type that should already be defined under **pkg/apis/<group>/<version>** via the **operator-sdk add api --kind=<kind> --api-version=<group/version>** command. If the controller package for that kind already exists at **pkg/controller/<kind>**, then the command does not overwrite and returns an error.

Subcommand	Description
crd	Adds a CRD and the CR files. The **<project_name>/deploy** path must already exist. The **--api-version** and **--kind** flags are required to generate the new Operator application. • Generated CRD filename: **<project_name>/deploy/crds/<group>_<version>_<kind>_crd.yaml** • Generated CR filename: **<project_name>/deploy/crds/<group>_<version>_<kind>_cr.yaml**

Table 4.28. **add api** flags

Flag	Description
--api-version (string)	CRD API version in the format **<group_name>/<version>**, for example **app.example.com/v1alpha1**.
--kind (string)	CRD **Kind** (e.g., **AppService**).

For example:

```
$ operator-sdk add api \
    --api-version app.example.com/v1alpha1 \
    --kind AppService
```

Example output

```
Create pkg/apis/app/v1alpha1/appservice_types.go
Create pkg/apis/addtoscheme_app_v1alpha1.go
Create pkg/apis/app/v1alpha1/register.go
Create pkg/apis/app/v1alpha1/doc.go
Create deploy/crds/app_v1alpha1_appservice_cr.yaml
Create deploy/crds/app_v1alpha1_appservice_crd.yaml
Running code-generation for Custom Resource (CR) group versions: [app:v1alpha1]
Generating deepcopy funcs
```

```
$ tree pkg/apis
```

Example output

```
pkg/apis/
├── addtoscheme_app_appservice.go
├── apis.go
└── app
```

```
    └── v1alpha1
        ├── doc.go
        ├── register.go
        └── types.go
```

```
$ operator-sdk add controller \
    --api-version app.example.com/v1alpha1 \
    --kind AppService
```

Example output

```
Create pkg/controller/appservice/appservice_controller.go
Create pkg/controller/add_appservice.go
```

```
$ tree pkg/controller
```

Example output

```
pkg/controller/
├── add_appservice.go
├── appservice
│   └── appservice_controller.go
└── controller.go
```

```
$ operator-sdk add crd \
    --api-version app.example.com/v1alpha1 \
    --kind AppService
```

Example output

```
Generating Custom Resource Definition (CRD) files
Create deploy/crds/app_v1alpha1_appservice_crd.yaml
Create deploy/crds/app_v1alpha1_appservice_cr.yaml
```

4.9.7. test

The **operator-sdk test** command can test the Operator locally.

4.9.7.1. local

The **local** subcommand runs Go tests built using the test framework of the Operator SDK locally.

Table 4.29. test local arguments

Arguments	Description
<test_location> (string)	Location of end-to-end (e2e) test files, for example **./test/e2e/**.

Table 4.30. test local flags

Flags	Description
--kubeconfig (string)	Location of **kubeconfig** for a cluster. Default:**~/.kube/config**.
--global-manifest (string)	Path to manifest for global resources. Default: **deploy/crd.yaml**.
--namespaced-manifest (string)	Path to manifest for per-test, namespaced resources. Default: combines **deploy/service_account.yaml**, **deploy/rbac.yaml**, and **deploy/operator.yaml**.
--namespace (string)	If non-empty, a single namespace to run tests in, for example **operator-test**. Default: **""**.
--go-test-flags (string)	Extra arguments to pass to **go test**, for example **-f "-v -parallel=2"**.
--up-local	Enable running the Operator locally with **go run** instead of as an image in the cluster.
--no-setup	Disable test resource creation.
--image (string)	Use a different Operator image from the one specified in the namespaced manifest.
-h, --help	Usage help output.

For example:

```
$ operator-sdk test local ./test/e2e/
```

Example output

```
ok  github.com/operator-framework/operator-sdk-samples/memcached-operator/test/e2e 20.410s
```

4.9.8. run

The **operator-sdk run** command provides options that can launch the Operator in various environments.

Table 4.31. run arguments

Arguments	Description
--kubeconfig (string)	The file path to a Kubernetes configuration file. Default: **$HOME/.kube/config**
--local	The Operator is run locally by building the Operator binary with the ability to access a Kubernetes cluster using a **kubeconfig** file.

Arguments	Description
--namespace (string)	The namespace where the Operator watches for changes. Default: **default**
--operator-flags	Flags that the local Operator might require. Example: **--flag1 value1 --flag2=value2**. For use with the**--local** flag only.
-h, --help	Usage help output.

4.9.8.1. --local

The **--local** flag launches the Operator on the local machine by building the Operator binary with the ability to access a Kubernetes cluster using a **kubeconfig** file.

For example:

```
$ operator-sdk run --local \
  --kubeconfig "mycluster.kubecfg" \
  --namespace "default" \
  --operator-flags "--flag1 value1 --flag2=value2"
```

The following example uses the default **kubeconfig**, the default namespace environment variable, and passes in flags for the Operator. To use the Operator flags, your Operator must know how to handle the option. For example, for an Operator that understands the **resync-interval** flag:

```
$ operator-sdk run --local --operator-flags "--resync-interval 10"
```

If you are planning on using a different namespace than the default, use the **--namespace** flag to change where the Operator is watching for custom resources (CRs) to be created:

```
$ operator-sdk run --local --namespace "testing"
```

For this to work, your Operator must handle the **WATCH_NAMESPACE** environment variable. This can be accomplished using the utility function **k8sutil.GetWatchNamespace** in your Operator.

4.10. APPENDICES

4.10.1. Operator project scaffolding layout

The **operator-sdk** CLI generates a number of packages for each Operator project. The following sections describes a basic rundown of each generated file and directory.

4.10.1.1. Go-based projects

Go-based Operator projects (the default type) generated using the **operator-sdk new** command contain the following directories and files:

File/folders	Purpose
cmd/	Contains **manager/main.go** file, which is the main program of the Operator. This instantiates a new manager which registers all custom resource defintitions (CRDs) under **pkg/apis/** and starts all controllers under **pkg/controllers/**.
pkg/apis/	Contains the directory tree that defines the APIs of the CRDs. Users are expected to edit the **pkg/apis/<group>/<version>/<kind>_types.go** files to define the API for each resource type and import these packages in their controllers to watch for these resource types.
pkg/controller	This **pkg** contains the controller implementations. Users are expected to edit the **pkg/controller/<kind>/<kind>_controller.go** files to define the reconcile logic of the controller for handling a resource type of the specified kind.
build/	Contains the Dockerfile and build scripts used to build the Operator.
deploy/	Contains various YAML manifests for registering CRDs, setting up RBAC, and deploying the Operator as a deployment.
Gopkg.toml **Gopkg.lock**	The Go Dep manifests that describe the external dependencies of this Operator.
vendor/	The Golang vendor folder that contains the local copies of the external dependencies that satisfy the imports of this project. Go Dep manages the vendor directly.

4.10.1.2. Helm-based projects

Helm-based Operator projects generated using the **operator-sdk new --type helm** command contain the following directories and files:

File/folders	Purpose
deploy/	Contains various YAML manifests for registering CRDs, setting up RBAC, and deploying the Operator as a Deployment.
helm-charts/<kind>	Contains a Helm chart initialized using the equivalent of the **helm create** command.
build/	Contains the Dockerfile and build scripts used to build the Operator.
watches.yaml	Contains group/version/kind (GVK) and Helm chart location.

CHAPTER 5. RED HAT OPERATORS

5.1. CLOUD CREDENTIAL OPERATOR

Purpose
The Cloud Credential Operator manages cloud provider credentials as Kubernetes custom resource definitions (CRDs).

Project
openshift-cloud-credential-operator

CRDs

- **credentialsrequests.cloudcredential.openshift.io**
  - Scope: Namespaced
  - CR: **credentialsrequest**
  - Validation: Yes

Configuration objects
No configuration required.

Notes

- The Cloud Credential Operator uses credentials from **kube-system/aws-creds**.
- The Cloud Credential Operator creates secrets based on **credentialsrequest**.

5.2. CLUSTER AUTHENTICATION OPERATOR

Purpose
The Cluster Authentication Operator installs and maintains the **Authentication** custom resource in a cluster and can be viewed with:

```
$ oc get clusteroperator authentication -o yaml
```

Project
cluster-authentication-operator

5.3. 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.

5.4. 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.

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

5.5. 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
- **prometheusrules.monitoring.coreos.com**

    - Scope: Namespaced
    - CR: **prometheusrule**
    - Validation: Yes
- **servicemonitors.monitoring.coreos.com**
    - Scope: Namespaced
    - CR: **servicemonitor**
    - Validation: Yes

Configuration objects

```
$ oc -n openshift-monitoring edit cm cluster-monitoring-config
```

5.6. CLUSTER NETWORK OPERATOR

Purpose

The Cluster Network Operator installs and upgrades the networking components on an OpenShift Container Platform cluster.

5.7. 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
```

The custom resource definition (CRD) **openshiftcontrollermanagers.operator.openshift.io** can be viewed in a cluster with:

```
$ oc get crd openshiftcontrollermanagers.operator.openshift.io -o yaml
```

Project

cluster-openshift-controller-manager-operator

5.8. 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

5.9. CLUSTER STORAGE OPERATOR

Purpose
The Cluster Storage Operator sets OpenShift Container Platform cluster-wide storage defaults. It ensures a default storage class exists for OpenShift Container Platform clusters.

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.

5.10. CLUSTER VERSION OPERATOR

Purpose
Project
cluster-version-operator

5.11. CONSOLE OPERATOR

Purpose
The Console Operator installs and maintains the OpenShift Container Platform web console on a cluster.

Project
console-operator

5.12. 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

5.13. 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

Configuration objects

```
$ oc edit etcd cluster
```

5.14. 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:

```
$ 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:

```
$ 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**:

```
$ oc get network/cluster -o jsonpath='{.status.clusterNetwork[*]}'
```

Example output

```
map[cidr:10.128.0.0/14 hostPrefix:23]
```

5.15. 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: Cluser
  - CR: **kubeapiserver**
  - Validation: Yes

Configuration objects

```
$ oc edit kubeapiserver
```

5.16. 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

5.17. 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 in order to form a valid configuration at the end.

5.18. 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**

5.19. 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

5.20. MARKETPLACE OPERATOR

Purpose
The Marketplace Operator is a conduit to bring off-cluster Operators to your cluster.

Project
operator-marketplace

5.21. 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.

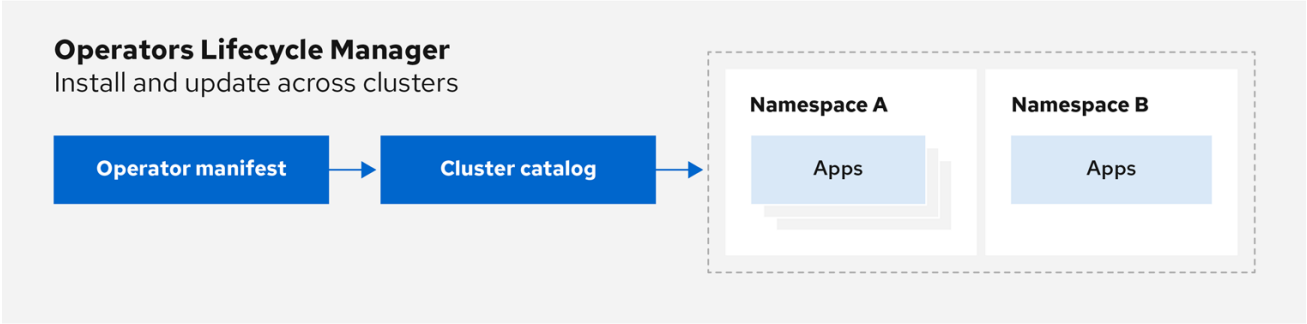
The Node Tuning Operator is part of a standard OpenShift Container Platform installation in version 4.1 and later.

Project
cluster-node-tuning-operator

5.22. 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 5.1. Operator Lifecycle Manager workflow



OLM runs by default in OpenShift Container Platform 4.5, 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 5.1. CRDs managed by OLM and Catalog Operators

Resource	Short name	Owner	Description
ClusterServiceVersion (CSV)	**csv**	OLM	Application metadata: name, version, icon, required resources, installation, and so on.
InstallPlan	**ip**	Catalog	Calculated list of resources to be created to automatically install or upgrade a CSV.
CatalogSource	**catsrc**	Catalog	A repository of CSVs, CRDs, and packages that define an application.
Subscription	**sub**	Catalog	Used to keep CSVs up to date by tracking a channel in a package.
OperatorGroup	**og**	OLM	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.

Each of these Operators is also responsible for creating the following resources:

Table 5.2. Resources created by OLM and Catalog Operators

Resource	Owner
Deployments	OLM
ServiceAccounts	
(Cluster)Roles	
(Cluster)RoleBindings	
CustomResourceDefinitions (CRDs)	Catalog
ClusterServiceVersions	

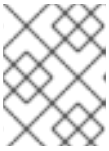
OLM Operator

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).

5.23. OPENSHIFT API SERVER OPERATOR

Purpose

The OpenShift API Server Operator installs and maintains the **openshift-apiserver** on a cluster.

Project

openshift-apiserver-operator

CRDs

- **openshiftapiservers.operator.openshift.io**
  - Scope: Cluster
  - CR: **openshiftapiserver**
  - Validation: Yes

5.24. PROMETHEUS OPERATOR

Purpose

The Prometheus Operator for Kubernetes provides easy monitoring definitions for Kubernetes services and deployment and management of Prometheus instances.

Once installed, the Prometheus Operator provides the following features:

- Create and Destroy: Easily launch a Prometheus instance for your Kubernetes namespace, a specific application or team easily using the Operator.

- Simple Configuration: Configure the fundamentals of Prometheus like versions, persistence, retention policies, and replicas from a native Kubernetes resource.

- Target Services via Labels: Automatically generate monitoring target configurations based on familiar Kubernetes label queries; no need to learn a Prometheus specific configuration language.

Project
prometheus-operator