OpenShift Container Platform 4.2

Operators

Working with Operators in OpenShift Container Platform 4.2

Last Updated: 2020-10-01

OpenShift Container Platform 4.2 Operators

Working with Operators in OpenShift Container Platform 4.2

Legal Notice

Abstract

This document provides information for working with Operators in OpenShift Container Platform 4.2. 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

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. 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 andkubectl 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 your cluster's current state. Off-cluster services continue to be a good match for a Service Broker, although Operators exist for these as well.

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
: The 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 4.2.

Operator Registry
: The Operator Registry stores ClusterServiceVersions (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 the OLM.

OperatorHub
: The 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.

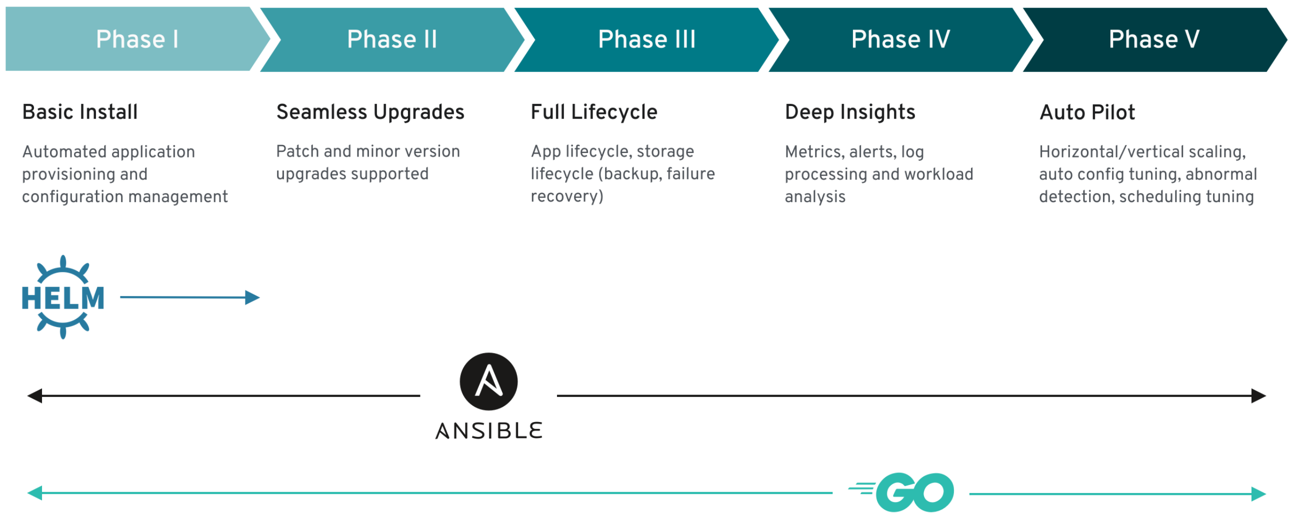
These tools are designed to be composable, so you can use any that are useful to you.

1.3. 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 an Operator's encapsulated operations 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


Phase I
Phase II
Phase III
Phase IV
Phase V
Basic Install
Automated application provisioning and configuration management
Seamless Upgrades
Patch and minor version upgrades supported
Full Lifecycle
App lifecycle, storage lifecycle (backup, failure recovery)
Deep Insights
Metrics, alerts, log processing and workload analysis
Auto Pilot
Horizontal/vertical scaling, auto config tuning, abnormal detection, scheduling tuning
HELM
ANSIBLE
GO

The above model also shows how these capabilities can best be developed through the Operator SDK's Helm, Go, and Ansible capabilities.

CHAPTER 2. UNDERSTANDING THE OPERATOR LIFECYCLE MANAGER (OLM)

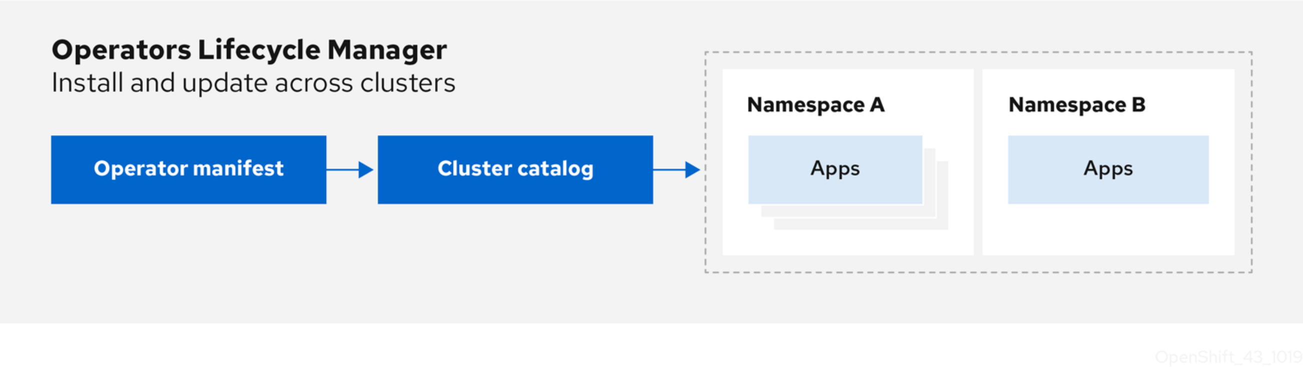
2.1. OPERATOR LIFECYCLE MANAGER WORKFLOW AND ARCHITECTURE

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

2.1.1. Overview of the Operator Lifecycle Manager

In OpenShift Container Platform 4.2, the *Operator Lifecycle Manager* (OLM) helps users install, update, and manage the lifecycle of all Operators and their associated services running across their clusters. It is part of the Operator Framework, an open source toolkit designed to manage Kubernetes native applications (Operators) in an effective, automated, and scalable way.

Figure 2.1. Operator Lifecycle Manager workflow



The OLM runs by default in OpenShift Container Platform 4.2, which aids cluster administrators in installing, upgrading, and granting access to Operators running on their cluster. The OpenShift Container Platform web console provides management screens for cluster administrators to install Operators, as well as grant specific projects access to use the catalog of Operators available on the cluster.

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

2.1.2. ClusterServiceVersions (CSVs)

A *ClusterServiceVersion* (CSV) is a YAML manifest created from Operator metadata that assists the Operator Lifecycle Manager (OLM) in running the Operator in a cluster.

A CSV is the metadata that accompanies an Operator container image, used to populate user interfaces with information like its logo, description, and version. It is also a source of technical information needed to run the Operator, like the RBAC rules it requires and which Custom Resources (CRs) it manages or depends on.

A CSV is composed of:

Metadata

- Application metadata:
  - Name, description, version (semver compliant), links, labels, icon, etc.

Install strategy

- Type: Deployment
  - Set of service accounts and required permissions
  - Set of Deployments.

CRDs

- Type
- Owned: Managed by this service
- Required: Must exist in the cluster for this service to run
- Resources: A list of resources that the Operator interacts with
- Descriptors: Annotate CRD spec and status fields to provide semantic information

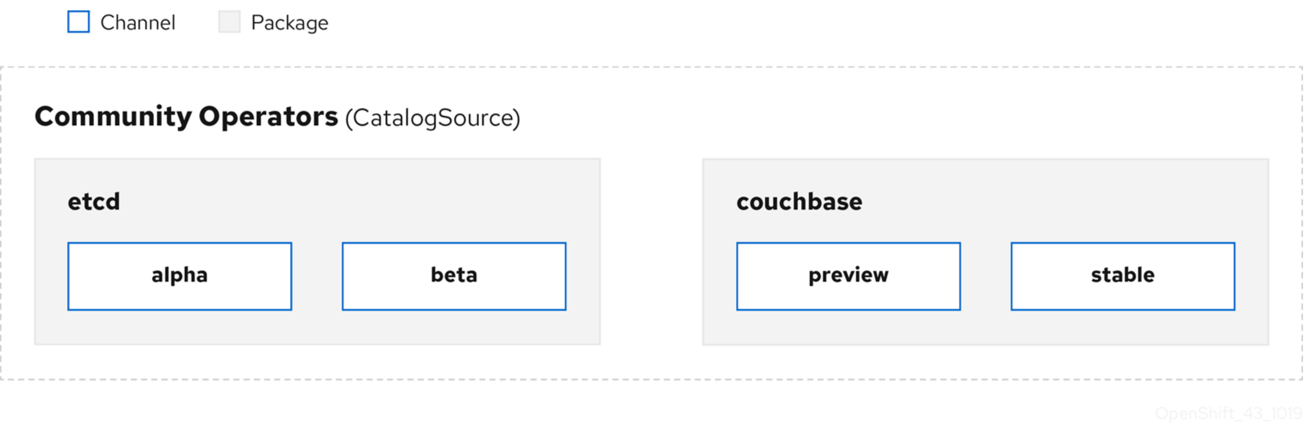
2.1.3. Operator installation and upgrade workflow in OLM

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

- ClusterServiceVersion (CSV)
- CatalogSource
- Subscription

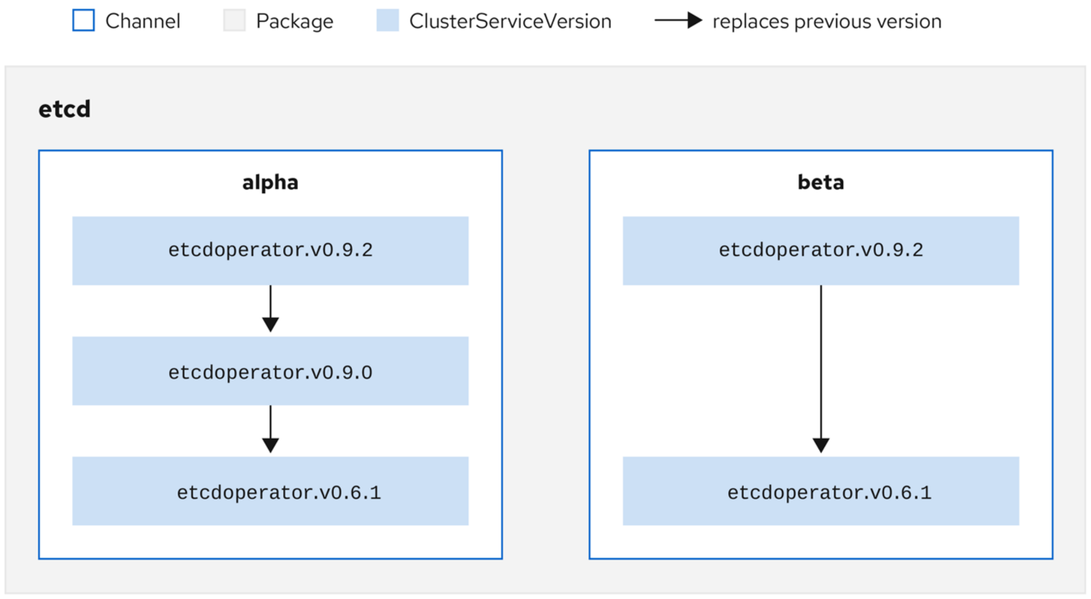
Operator metadata, defined in CSVs, can be stored in a collection called a CatalogSource. OLM uses CatalogSources, which use the Operator Registry API, to query for available Operators as well as upgrades for installed Operators.

Figure 2.2. CatalogSource overview



Within a CatalogSource, Operators are organized into *packages* and streams of updates called *channels*, which should be a familiar update pattern from OpenShift Container Platform or other software on a continuous release cycle like web browsers.

Figure 2.3. Packages and channels in a CatalogSource



OpenShift_43_1019

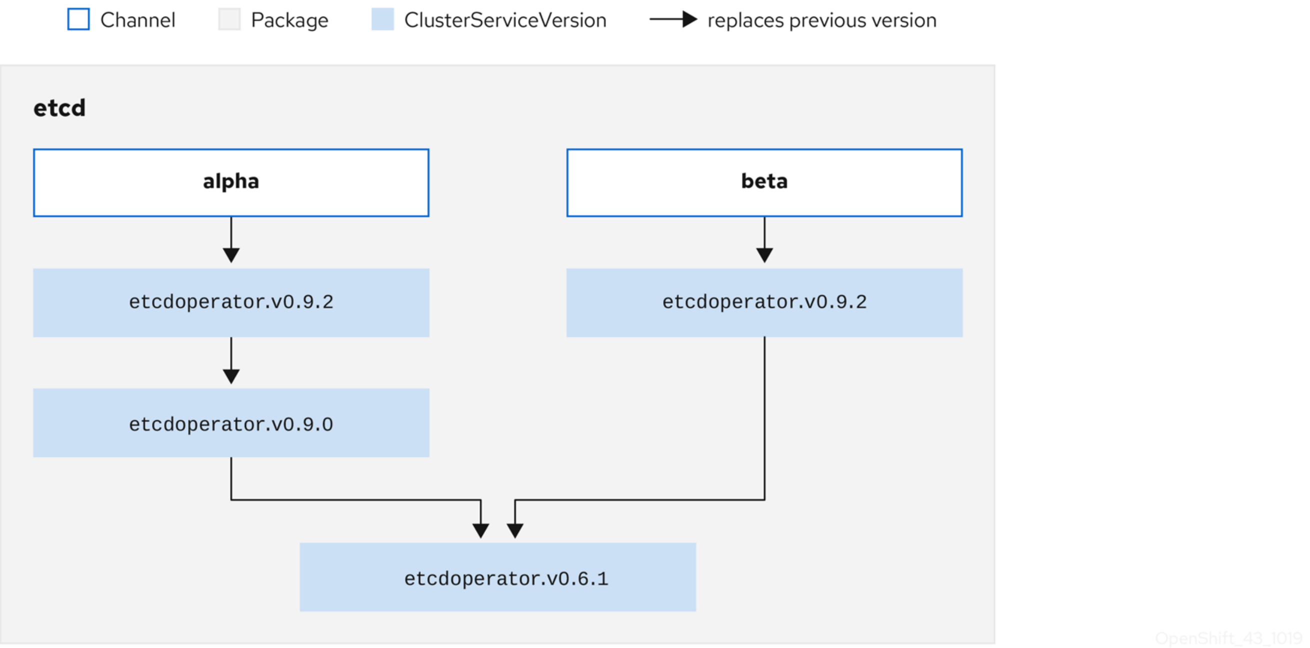
A user indicates a particular package and channel in a particular CatalogSource in a *Subscription*, for example an **etcd** package and its **alpha** channel. If a Subscription is made to a package that has not yet been installed in the namespace, the latest Operator for that package is installed.

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 2.4. OLM’s graph of available channel updates



For 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 CatalogSource, package, channel, and CSV, a catalog must be able to return, unambiguously and deterministically, a single CSV that **replaces** the input CSV.

2.1.3.1. Example upgrade path

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

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

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

2.1.3.2. Skipping upgrades

OLM’s basic path for upgrades is:

- A CatalogSource is updated with one or more updates to an Operator.
- OLM traverses every version of the Operator until reaching the latest version the CatalogSource 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.

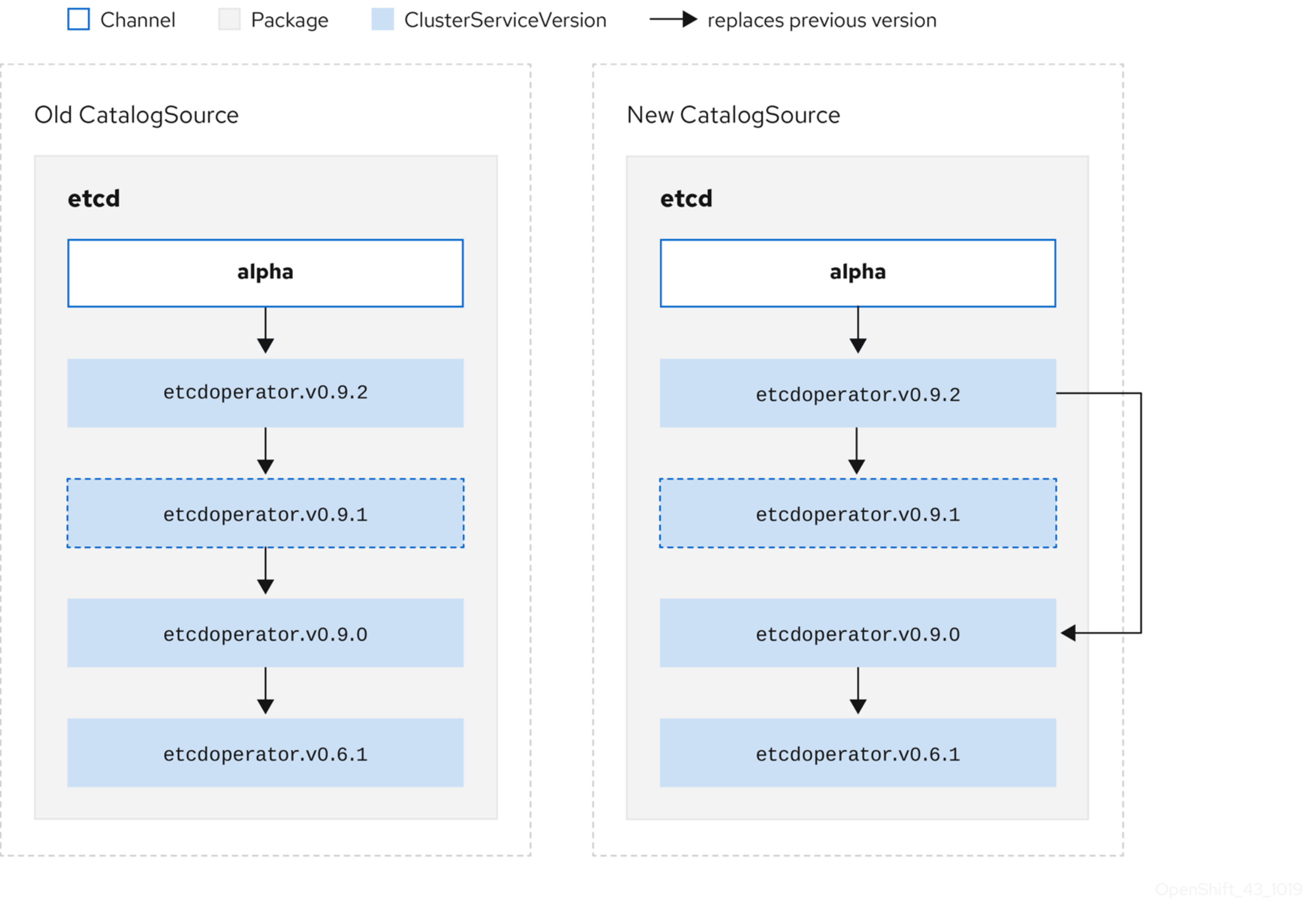
For example:

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 Old CatalogSource and New CatalogSource:

Figure 2.5. 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.

2.1.3.3. Replacing multiple Operators

Creating the 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.

For example:

CSV with skipRange

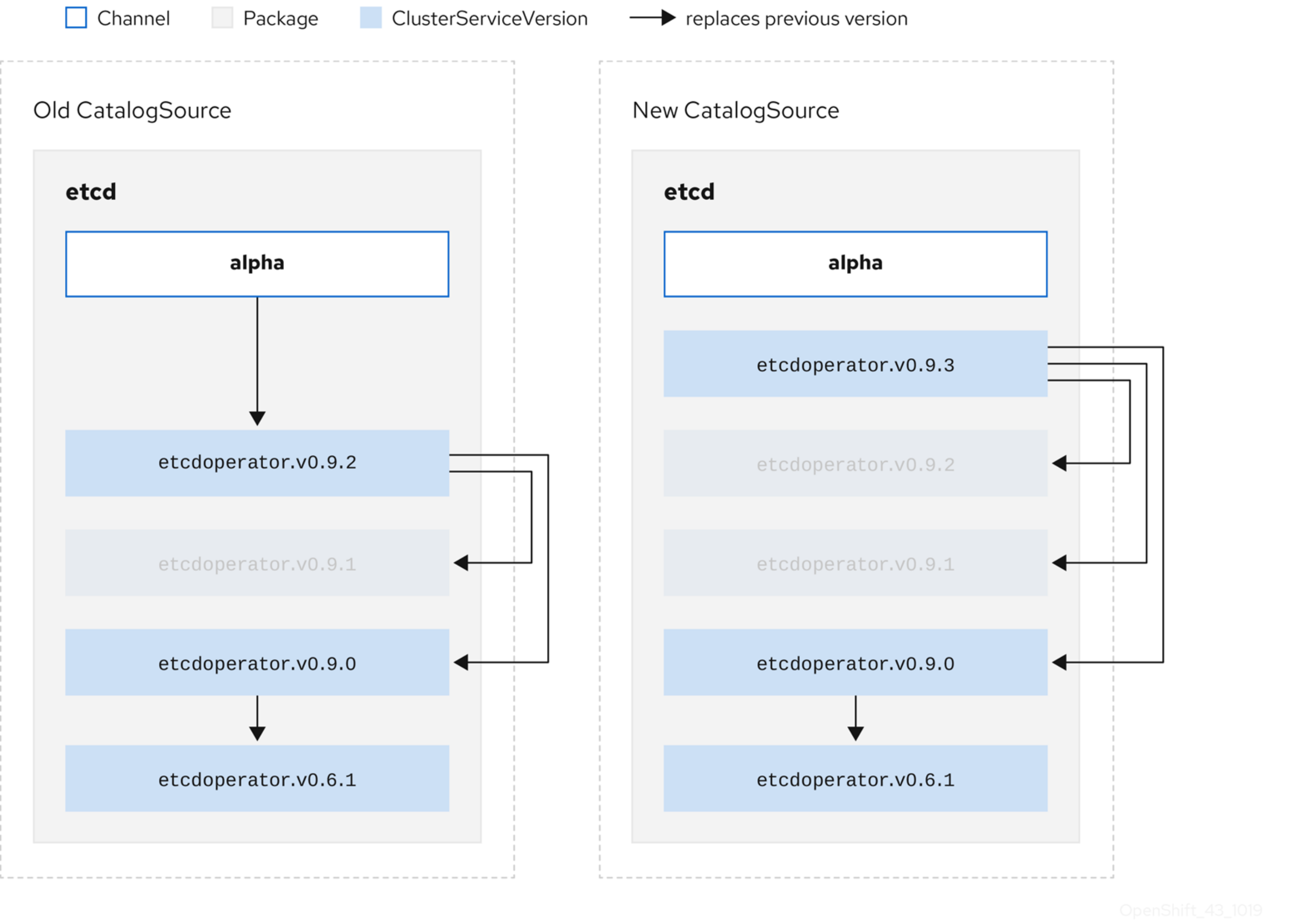
```
apiVersion: operators.coreos.com/v1alpha1
kind: ClusterServiceVersion
metadata:
  name: elasticsearch-operator.v4.1.2
  namespace: <namespace>
  annotations:
    olm.skipRange: '>=4.1.0 <4.1.2'
```

2.1.3.4. Z-stream support

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

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

Figure 2.6. 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.

2.1.4. Operator Lifecycle Manager architecture

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

Each of these Operators is responsible for managing the Custom Resource Definitions (CRDs) that are the basis for the OLM framework:

Table 2.1. CRDs managed by OLM and Catalog Operators

Resource	Short name	Owner	Description
ClusterServiceVersion	**csv**	OLM	Application metadata: name, version, icon, required resources, installation, etc.
InstallPlan	**ip**	Catalog	Calculated list of resources to be created in order 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	Used to group multiple namespaces and prepare them for use by an Operator.

Each of these Operators is also responsible for creating resources:

Table 2.2. Resources created by OLM and Catalog Operators

Resource	Owner
Deployments	OLM
ServiceAccounts	

Resource	Owner
(Cluster)Roles	
(Cluster)RoleBindings	
Custom Resource Definitions (CRDs)	Catalog
ClusterServiceVersions (CSVs)	

2.1.4.1. 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; users can choose to manually create these resources using the CLI, or users can choose to create these resources 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.

While the OLM Operator is often configured to watch all namespaces, it can also be operated alongside other OLM Operators so long as they all manage separate namespaces.

OLM Operator workflow

- Watches for ClusterServiceVersion (CSVs) in a namespace and checks that requirements are met. If so, runs the install strategy for the CSV.

  NOTE

  A CSV must be an active member of an OperatorGroup in order for the install strategy to be run.

2.1.4.2. Catalog Operator

The Catalog Operator is responsible for resolving and installing CSVs and the required resources they specify. It is also responsible for watching CatalogSources for updates to packages in channels and upgrading them (optionally automatically) to the latest available versions.

A user who wishes to track a package in a channel creates a Subscription resource configuring the desired package, channel, and the CatalogSource from which to pull updates. When updates are found, an appropriate InstallPlan is written into the namespace on behalf of the user.

Users can also create an InstallPlan resource directly, containing the names of the desired CSV and an approval strategy, and the Catalog Operator creates an execution plan for the creation of all of the required resources. After it is approved, the Catalog Operator creates all of the resources in an InstallPlan; this then independently satisfies the OLM Operator, which proceeds to install the CSVs.

Catalog Operator workflow

- Has a cache of CRDs and CSVs, indexed by name.

- Watches for unresolved InstallPlans created by a user:
  - Finds the CSV matching the name requested and adds it as a resolved resource.
  - For each managed or required CRD, adds it as a resolved resource.
  - For each required CRD, finds the CSV that manages it.
- Watches for resolved InstallPlans and creates all of the discovered resources for it (if approved by a user or automatically).
- Watches for CatalogSources and Subscriptions and creates InstallPlans based on them.

2.1.4.3. 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 all of the information that is required to update a CSV to the latest version in a channel, stepping through each intermediate version.

2.1.5. Exposed metrics

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

Table 2.3. Metrics exposed by OLM

Name	Description
csv_count	Number of CSVs successfully registered.
install_plan_count	Number of InstallPlans.
subscription_count	Number of Subscriptions.
csv_upgrade_count	Monotonic count of CatalogSources.

2.2. OPERATOR LIFECYCLE MANAGER DEPENDENCY RESOLUTION

This guide outlines dependency resolution and Custom Resource Definition (CRD) upgrade lifecycles within the Operator Lifecycle Manager (OLM) in OpenShift Container Platform.

2.2.1. 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.

2.2.2. Custom Resource Definition (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 (CRs), that are associated with the serving versions of the CRD are valid when validated against the new CRD's validation schema.

2.2.2.1. 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 one version **v1alpha1** and you want to add a new version **v1beta1** and mark it as the new storage version:

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

   1 Add a new entry for **v1beta1**.

2. Ensure the referencing version of the CRD in your CSV's **owned** section 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.

2.2.2.2. Deprecating or removing a CRD version

OLM does not allow a serving version of a 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 CRD's status. 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 version of the CRD in your CSV’s **owned** section is updated accordingly if that version is removed from the CRD.

2.2.3. Example dependency resolution scenarios

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

Example: Deprecating dependent APIs
A and B are APIs (e.g., CRDs):

- A’s provider depends on B.
- B’s provider has a Subscription.
- B’s provider 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:

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

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

This is another case OLM prevents with its upgrade strategy.

2.3. OPERATORGROUPS

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

2.3.1. About OperatorGroups

An *OperatorGroup* is an OLM resource that provides multitenant configuration to OLM-installed Operators. An OperatorGroup selects a set of 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 ClusterServiceVersion’s (CSV) **olm.targetNamespaces** annotation. This annotation is applied to member Operator’s CSV instances and is projected into their deployments.

2.3.2. OperatorGroup membership

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

- The Operator's CSV exists in the same namespace as the OperatorGroup.
- The Operator's CSV's InstallModes support the set of namespaces targeted by the OperatorGroup.

An InstallMode consists of an **InstallModeType** field and a boolean **Supported** field. A CSV's spec can contain a set of InstallModes of four distinct **InstallModeTypes**:

Table 2.4. InstallModes and supported OperatorGroups

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

NOTE

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

2.3.3. Target namespace selection

The set of namespaces for the OperatorGroup is specified using a label selector with the **spec.selector** field:

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

You can also explicitly name the target namespaces using the **spec.targetNamespaces** field:

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

NOTE

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* OperatorGroup, 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 an OperatorGroup's **status.namespaces** field. A global OperatorGroup's **status.namespace** contains the empty string (**""**), which signals to a consuming Operator that it should watch all namespaces.

2.3.4. OperatorGroup CSV annotations

Member CSVs of an OperatorGroup have the following annotations:

Annotation	Description
olm.operatorGroup=<group_name>	Contains the name of the OperatorGroup.
olm.operatorGroupNamespace=<group_namespace>	Contains the namespace of the OperatorGroup.
olm.targetNamespaces=<target_namespaces>	Contains a comma-delimited string that lists the OperatorGroup's target namespace selection.

NOTE

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

2.3.5. Provided APIs annotation

Information about what **GroupVersionKinds** (GVKs) are provided by an OperatorGroup are shown in an **olm.providedAPIs** annotation. The annotation's value is a string consisting of **<kind>.<version>.<group>** delimited with commas. The GVKs of CRDs and APIServices provided by all active member

CSVs of an OperatorGroup are included.

Review the following example of an OperatorGroup 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
```

2.3.6. Role-based access control

When an OperatorGroup is created, three ClusterRoles are generated. Each contains a single AggregationRule with a ClusterRoleSelector set to match a label, as shown below:

ClusterRole	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 OperatorGroup, as long as the CSV is watching all namespaces with the **AllNamespaces** InstallMode and is not in a failed state with reason **InterOperatorGroupOwnerConflict**.

- ClusterRoles for each API resource from a CRD
- ClusterRoles for each API resource from an APIService
- Additional Roles and RoleBindings

Table 2.5. ClusterRoles generated for each API resource from a CRD

ClusterRole	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>**

ClusterRole	Settings
<kind>.<group>-<version>-view-crdview	Verbs on **apiextensions.k8s.io customresourcedefinitions <crd-name>**: ● **get** Aggregation labels: ● **rbac.authorization.k8s.io/aggregate-to-view: true** ● **olm.opgroup.permissions/aggregate-to-view: <operatorgroup_name>**

Table 2.6. ClusterRoles generated for each API resource from an APIService

ClusterRole	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>**

ClusterRole	Settings
<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 RoleBindings

- If the CSV defines exactly one target namespace that contains *, then a ClusterRole and corresponding ClusterRoleBinding are generated for each permission defined in the CSV's permissions field. 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 RoleBindings in the Operator namespace with the **olm.owner: <csv_name>** and **olm.owner.namespace: <csv_namespace>** labels are copied into the target namespace.

2.3.7. Copied CSVs

OLM creates copies of all active member CSVs of an OperatorGroup in each of that OperatorGroup's target namespaces. 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 OperatorGroup that their source CSV belongs to no longer targets the copied CSV's namespace.

2.3.8. Static OperatorGroups

An OperatorGroup is *static* if its **spec.staticProvidedAPIs** field is set to **true**. As a result, OLM does not modify the OperatorGroup's **olm.providedAPIs** annotation, which means that it can be set in advance. This is useful when a user wants to use an OperatorGroup 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 OperatorGroup 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.monitoring.coreos.com,ServiceMonitor.v1.monitoring.coreos.com
spec:
  staticProvidedAPIs: true
  selector:
    matchLabels:
      something.cool.io/cluster-monitoring: "true"
```

2.3.9. OperatorGroup intersection

Two OperatorGroups 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 OperatorGroups with intersecting provided APIs can compete for the same resources in the set of intersecting namespaces.

NOTE

When checking intersection rules, an OperatorGroup's 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 CSV's OperatorGroup 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 OperatorGroup's:
  - Continue transitioning.
- If **true** and the CSV's provided APIs are *not* a subset of the OperatorGroup's:
  - If the OperatorGroup is static:
    - Clean up any deployments that belong to the CSV.
    - Transition the CSV to a failed state with status reason **CannotModifyStaticOperatorGroupProvidedAPIs**.
  - If the OperatorGroup is *not* static:
    - Replace the OperatorGroup'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 OperatorGroup'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 OperatorGroup's:
  - If the OperatorGroup is static:

        - Clean up any deployments that belong to the CSV.

        - Transition the CSV to a failed state with status reason **CannotModifyStaticOperatorGroupProvidedAPIs**.

    - If the OperatorGroup is *not* static:

        - Replace the OperatorGroup’s **olm.providedAPIs** annotation with the difference between itself and the CSV’s provided APIs.

> NOTE
>
> Failure states caused by OperatorGroups are non-terminal.

The following actions are performed each time an OperatorGroup synchronizes:

- The set of provided APIs from active member CSVs is calculated from the cluster. Note that copied CSVs are ignored.

- The cluster set is compared to **olm.providedAPIs**, and if **olm.providedAPIs** contains any extra APIs, then those APIs are pruned.

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

2.3.10. Troubleshooting OperatorGroups

Membership

- If more than one OperatorGroup exists in a single namespace, any CSV created in that namespace will transition to a failure state with the reason **TooManyOperatorGroups**. CSVs in a failed state for this reason will transition to pending once the number of OperatorGroups in their namespaces reaches one.

- If a CSV’s InstallModes do not support the target namespace selection of the OperatorGroup in its namespace, the CSV will transition to a failure state with the reason **UnsupportedOperatorGroup**. CSVs in a failed state for this reason will transition to pending once either the OperatorGroup’s target namespace selection changes to a supported configuration, or the CSV’s InstallModes are modified to support the OperatorGroup’s target namespace selection.

CHAPTER 3. UNDERSTANDING THE OPERATORHUB

This guide outlines the architecture of the OperatorHub.

3.1. OVERVIEW OF THE OPERATORHUB

The *OperatorHub* is available via the OpenShift Container Platform web console and is the interface that cluster administrators use to discover and install Operators. With one click, an Operator can be pulled from their 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 the Operator Lifecycle Manager (OLM).

Cluster administrators can choose from OperatorSources 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.

Additional resources

- Red Hat Partner Connect
- Red Hat Marketplace

3.2. 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 information is exposed through the OperatorHub user interface, it is only used directly by those who are creating their own Operators.

NOTE

While OperatorHub no longer uses CatalogSourceConfig resources, they are still supported in OpenShift Container Platform.

3.2.1. OperatorHub CRD

You can use the OperatorHub CRD to change the state of the default OperatorSources 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 OperatorSources that are configured by default during a OpenShift Container Platform installation.
2. Disable default OperatorSources individually by changing the **disabled** parameter value per source.

3.2.2. 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 the 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.

CHAPTER 4. ADDING OPERATORS TO A CLUSTER

This guide walks cluster administrators through installing Operators to an OpenShift Container Platform cluster and subscribing Operators to namespaces.

4.1. INSTALLING OPERATORS FROM THE OPERATORHUB

As a cluster administrator, you can install an Operator from the OperatorHub using the OpenShift Container Platform web console or the CLI. You can then subscribe the Operator to one or more namespaces to make it available for developers on your cluster.

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

Installation Mode

Choose **All namespaces on the cluster (default)**to have the Operator installed on all namespaces or choose individual namespaces, if available, to only install the Operator on selected namespaces. This example chooses **All namespaces…** to make the Operator available to all users and projects.

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, the 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, the 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.

4.1.1. Installing from the OperatorHub using the web console

This procedure uses the Couchbase Operator as an example to install and subscribe to an Operator from the OperatorHub using the OpenShift Container Platform web console.

Prerequisites

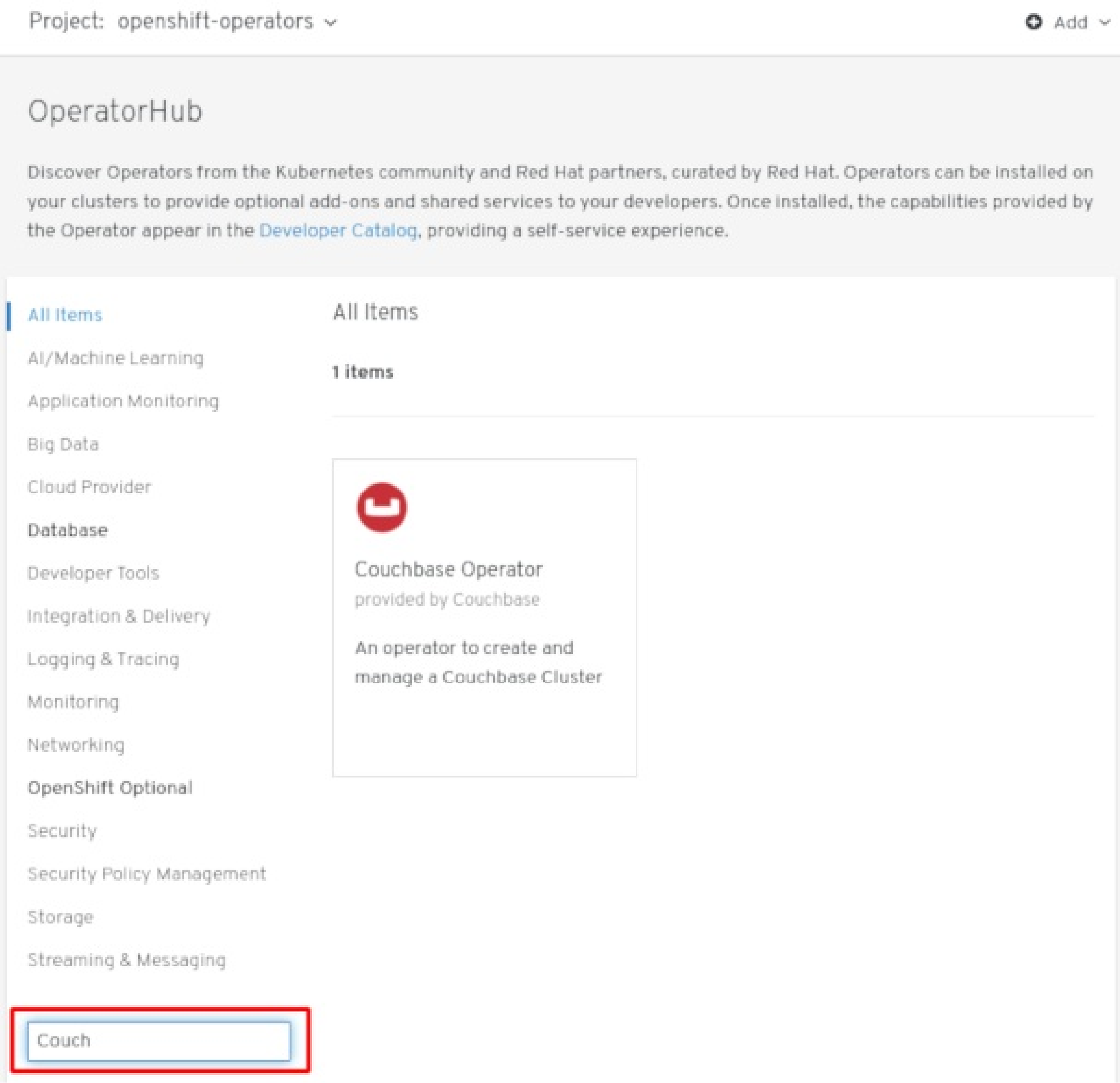
- Access to an OpenShift Container Platform cluster using an account with **cluster-admin** permissions.

Procedure

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

2. Scroll or type a keyword into the **Filter by keyword** box (in this case, **Couchbase**) to find the Operator you want.

Figure 4.1. Filter Operators by keyword



Project: openshift-operators

OperatorHub

Discover Operators from the Kubernetes community and Red Hat partners, curated by Red Hat. Operators can be installed on your clusters to provide optional add-ons and shared services to your developers. Once installed, the capabilities provided by the Operator appear in the Developer Catalog, providing a self-service experience.

All Items

1 items

Couchbase Operator

provided by Couchbase

An operator to create and manage a Couchbase Cluster

Couch

3. Select the Operator. For a Community Operator, you are warned that Red Hat does not certify those Operators. You must acknowledge that warning before continuing. Information about the Operator is displayed.

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

5. On the **Create Operator Subscription** 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.

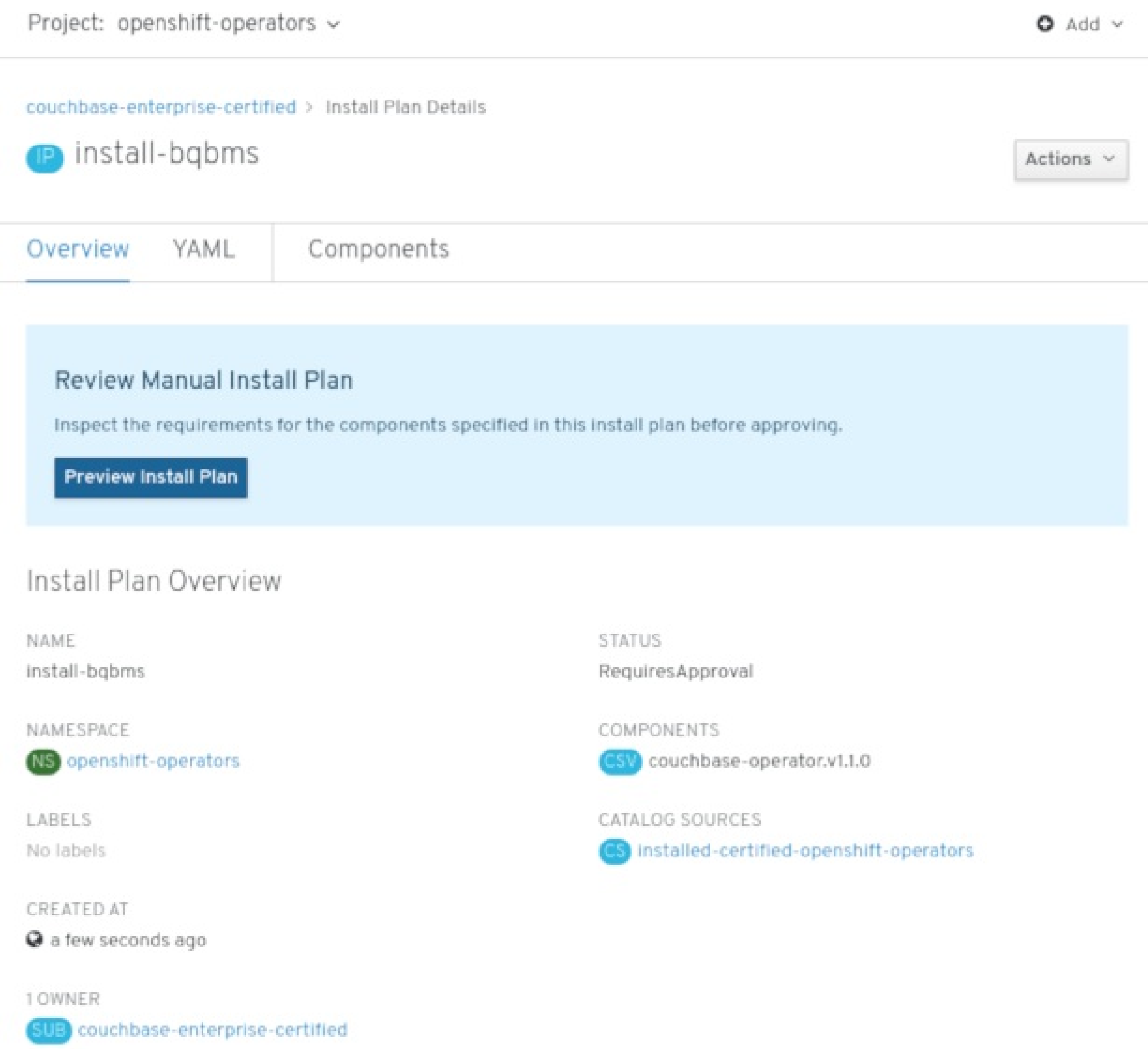
    b. Select an **Update Channel** (if more than one is available).

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

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

   a. If you selected a Manual approval strategy, the Subscription’s upgrade status will remain **Upgrading** until you review and approve its Install Plan.

   Figure 4.2. Manually approving from theInstall Plan page


Project: openshift-operators
Add
couchbase-enterprise-certified > Install Plan Details
IP
Actions
Overview
YAML
Components
Preview Install Plan

install-bqbms

Review Manual Install Plan

Inspect the requirements for the components specified in this install plan before approving.

Install Plan Overview

NAME
install-bqbms

STATUS
RequiresApproval

NAMESPACE
NS openshift-operators

COMPONENTS
CSV couchbase-operator.v1.1.0

LABELS
No labels

CATALOG SOURCES
CS installed-certified-openshift-operators

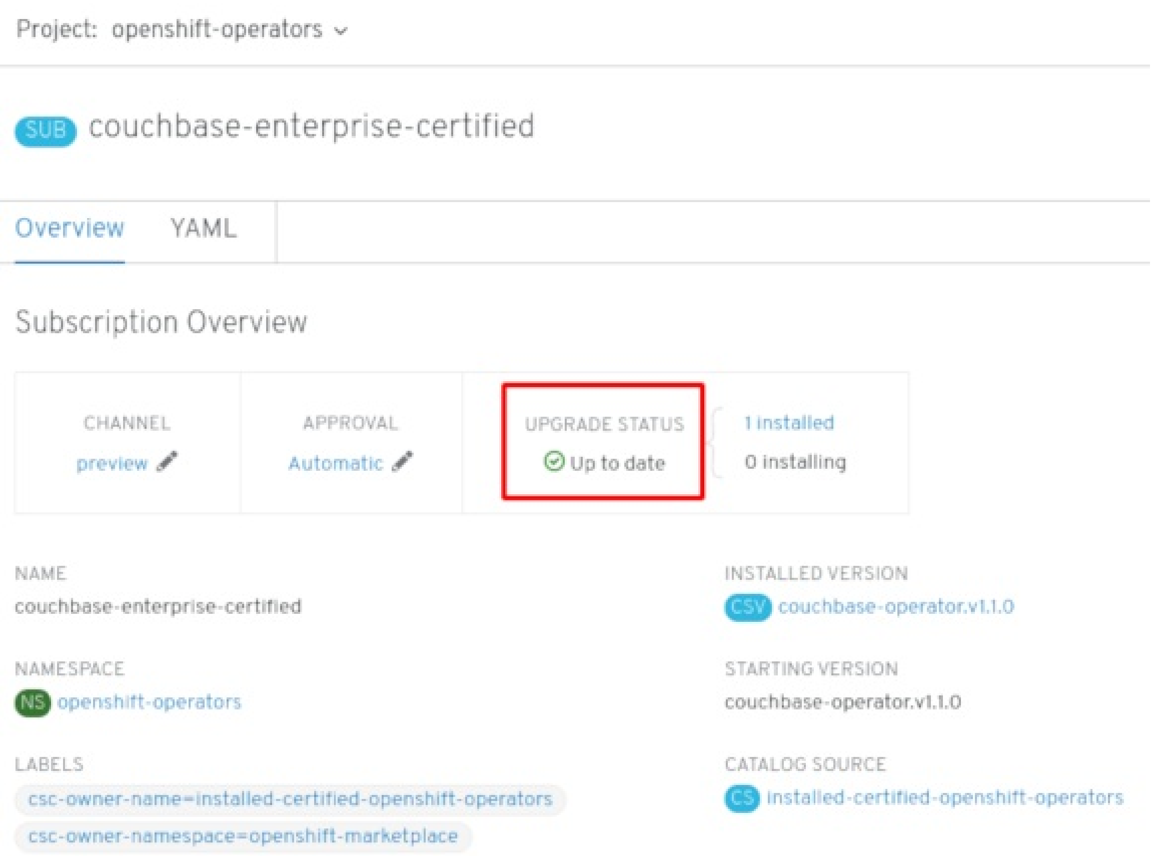
CREATED AT
a few seconds ago

1 OWNER
SUB couchbase-enterprise-certified

   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.

Figure 4.3. Subscription upgrade status Up to date



Project: openshift-operators

SUB couchbase-enterprise-certified

Overview YAML

Subscription Overview

CHANNEL
preview

APPROVAL
Automatic

UPGRADE STATUS
Up to date

1 installed
0 installing

NAME
couchbase-enterprise-certified

NAMESPACE
NS openshift-operators

LABELS
csc-owner-name=installed-certified-openshift-operators
csc-owner-namespace=openshift-marketplace

INSTALLED VERSION
CSV couchbase-operator.v1.1.0

STARTING VERSION
couchbase-operator.v1.1.0

CATALOG SOURCE
CS installed-certified-openshift-operators

7. After the Subscription’s upgrade status is **Up to date**, select **Operators → Installed Operators** to verify that the **Couchbase** ClusterServiceVersion (CSV) eventually shows up and its **Status** ultimately resolves to **InstallSucceeded** in the relevant namespace.

   NOTE

   For the **All namespaces…** Installation Mode, the status resolves to **InstallSucceeded** in the **openshift-operators** namespace, but the status is **Copied** if you check in other namespaces.

   If it does not:

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

4.1.2. Installing from the OperatorHub using the CLI

Instead of using the OpenShift Container Platform web console, you can install an Operator from the 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 **cluster-admin** permissions.

- Install the **oc** command to your local system.

Procedure

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

```
$ oc get packagemanifests -n openshift-marketplace
NAME                          CATALOG               AGE
3scale-operator               Red Hat Operators     91m
amq-online                    Red Hat Operators     91m
amq-streams                   Red Hat Operators     91m
...
couchbase-enterprise-certified    Certified Operators   91m
mariadb                       Certified Operators   91m
mongodb-enterprise            Certified Operators   91m
...
etcd                          Community Operators   91m
jaeger                        Community Operators   91m
kubefed                       Community Operators   91m
...
```

Note the CatalogSource(s) for your desired Operator(s).

2. Inspect your desired Operator to verify its supported InstallModes and available Channels:

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

3. The namespace to which you subscribe the Operator must have an OperatorGroup that matches the Operator’s InstallMode, 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 OperatorGroup in place.
However, if the Operator uses the **SingleNamespace** mode and you do not already have an appropriate OperatorGroup 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

```
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

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

   1 For **AllNamespaces** InstallMode usage, specify the **openshift-operators** namespace. Otherwise, specify the relevant single namespace for **SingleNamespace** InstallMode usage.

   2 Name of the Operator to subscribe to.

   3 Name of the CatalogSource that provides the Operator.

   4 Namespace of the CatalogSource. Use **openshift-marketplace** for the default OperatorHub CatalogSources.

5. Create the Subscription object:

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

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

Additional resources

- To install custom Operators to a cluster using the OperatorHub, you must first upload your Operator artifacts to Quay.io, then add your own **OperatorSource** to your cluster. Optionally, you can add Secrets to your Operator to provide authentication. After, you can manage the Operator in your cluster as you would any other Operator. For these steps, see Testing Operators.

Additional resources

- About OperatorGroups

4.2. OVERRIDING AN OPERATOR'S PROXY SETTINGS

If a cluster-wide egress proxy is configured, applications created from Operators using the Operator Lifecycle Manager (OLM) inherit the cluster-wide proxy settings on their Deployments and Pods. Cluster administrators can also override these proxy settings by configuring the Operator's Subscription.

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 **Create Operator Subscription** page, modify the Subscription object's YAML 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
   ```

   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 subscribed Operator's Deployments.

4. Click **Subscribe** to make the Operator available to the selected namespaces.
5. After the Operator's CSV 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
   ```

```
        - 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
...
```

Additional resources

- Configuring the cluster-wide proxy
- See the OpenShift Container Platform 4.2 Release Notes for details on known issue BZ#1751903 regarding unset environment variables when overriding an Operator's proxy settings.

CHAPTER 5. DELETING OPERATORS FROM A CLUSTER

The following describes how to delete Operators from a cluster using either the web console or the CLI.

5.1. 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.

3. When prompted by the **Remove Operator Subscription** window, you can select the **Also completely remove the Operator from the selected namespace** check box to remove all components related to the Operator. This removes the CSV, which in turn removes the Pods and Deployments associated with the Operator. Leaving this unchecked results in only the Subscription being removed.

4. Select **Remove**. This Operator will stop running and no longer receive updates.

5.2. 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
  currentCSV: jaeger-operator.v1.8.2
```

2. Delete the Operator’s Subscription (for example, **jaeger**):

```
$ oc delete subscription jaeger -n openshift-operators
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
clusterserviceversion.operators.coreos.com "jaeger-operator.v1.8.2" deleted
```

CHAPTER 6. 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.

6.1. CREATING AN ETCD CLUSTER USING AN OPERATOR

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

Prerequisites

- Access to an OpenShift Container Platform 4.2 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 ClusterServiceVersions (CSVs). CSVs are used to launch and manage the software provided by the Operator.

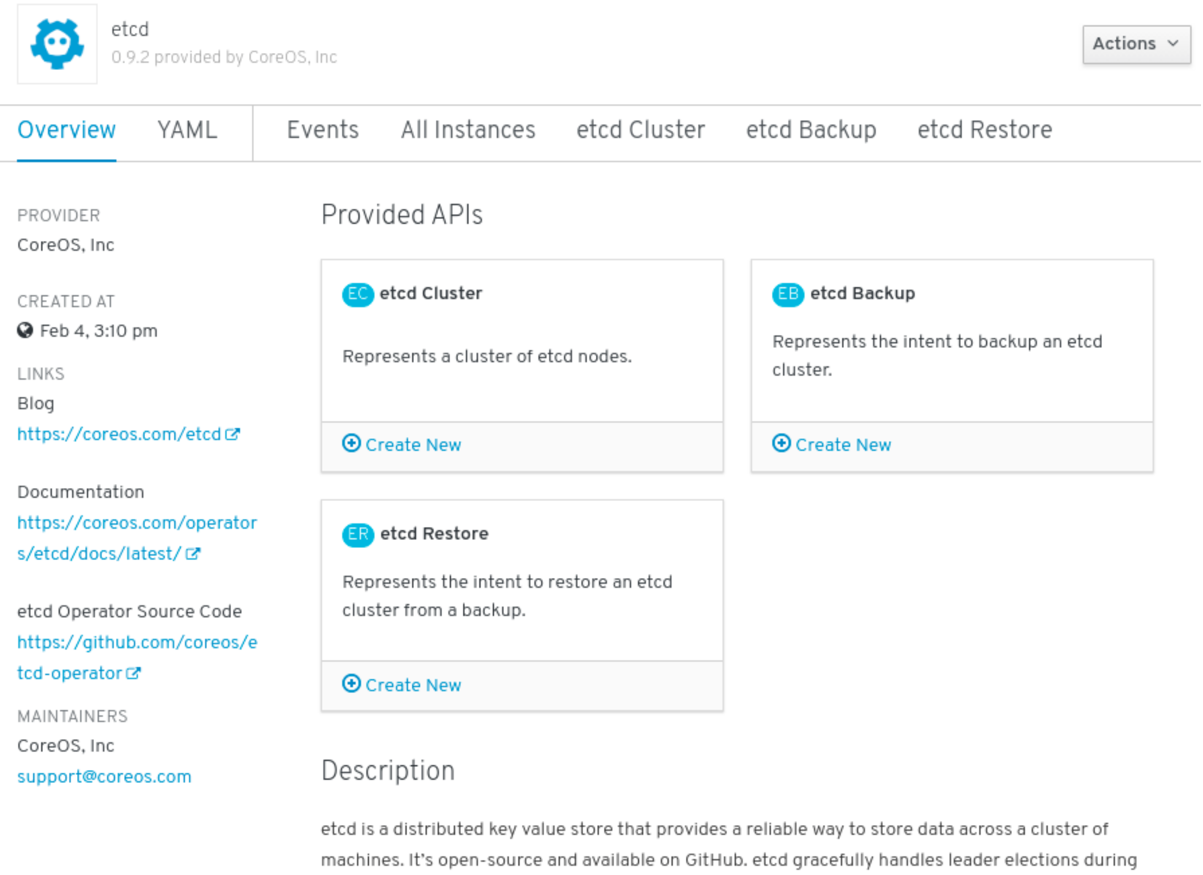
   #### TIP

   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:

Figure 6.1. etcd Operator overview



etcd
0.9.2 provided by CoreOS, Inc

Actions

Overview YAML | Events All Instances etcd Cluster etcd Backup etcd Restore

PROVIDER
CoreOS, Inc

CREATED AT
Feb 4, 3:10 pm

LINKS
Blog
https://coreos.com/etcd

Documentation
https://coreos.com/operators/etcd/docs/latest/

etcd Operator Source Code
https://github.com/coreos/etcd-operator

MAINTAINERS
CoreOS, Inc
support@coreos.com

Provided APIs

EC etcd Cluster
Represents a cluster of etcd nodes.
Create New

EB etcd Backup
Represents the intent to backup an etcd cluster.
Create New

ER etcd Restore
Represents the intent to restore an etcd cluster from a backup.
Create New

Description

etcd is a distributed key value store that provides a reliable way to store data across a cluster of machines. It's open-source and available on GitHub. etcd gracefully handles leader elections during

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 **Deployments** or **ReplicaSets**, but contain logic specific to managing etcd.

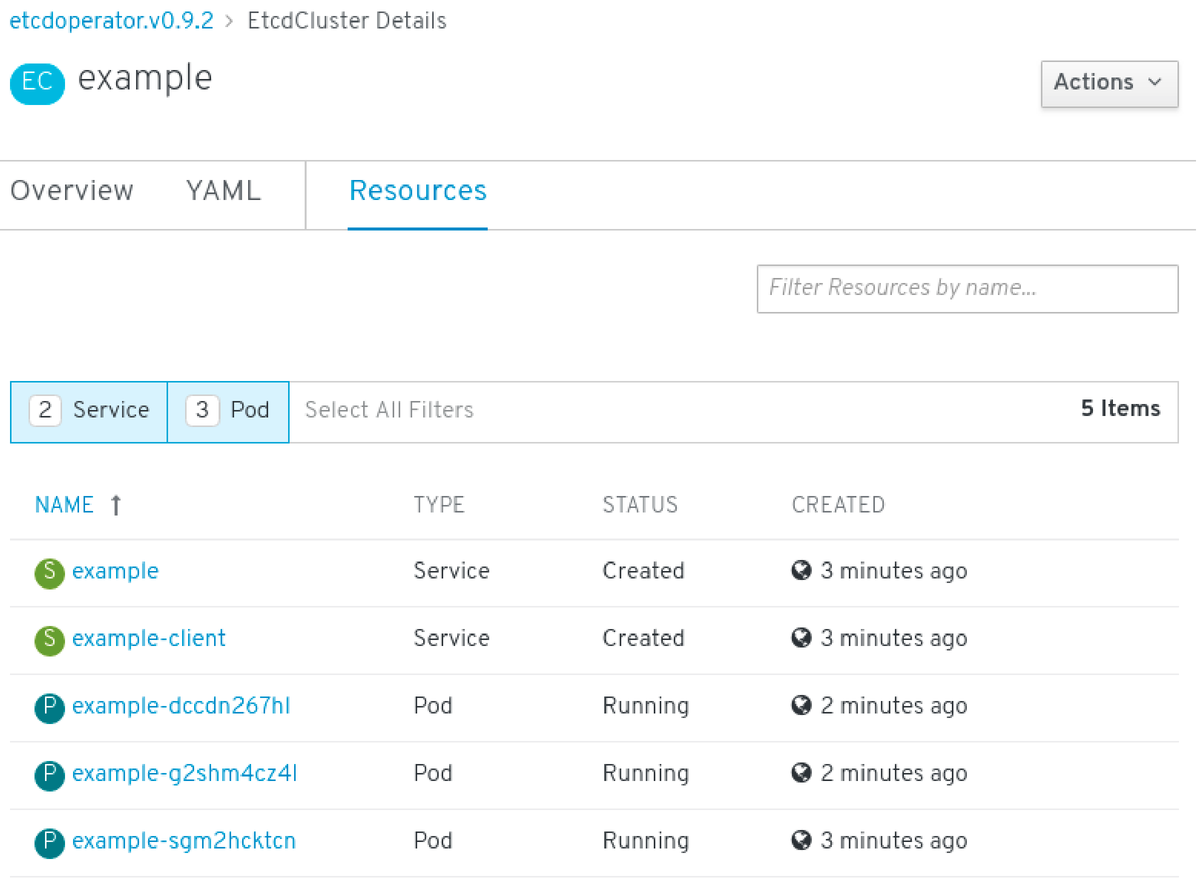
4. Create a new etcd cluster:

    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.

Figure 6.2. etcd Operator resources


etcdoperator.v0.9.2 > EtcdCluster Details

EC example

5 Items

NAME ↑	TYPE	STATUS	CREATED
S example	Service	Created	3 minutes ago
S example-client	Service	Created	3 minutes ago
P example-dccdn267hl	Pod	Running	2 minutes ago
P example-g2shm4cz4l	Pod	Running	2 minutes ago
P example-sgm2hcktcn	Pod	Running	3 minutes ago

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.

CHAPTER 7. 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 Source resources regarding their state and actions performed. This helps users better understand the healthiness of their Operators.

7.1. CONDITION TYPES

Subscriptions can report the following condition types:

Table 7.1. Subscription condition types

Condition	Description
CatalogSourcesUnhealthy	Some or all of the Catalog Sources to be used in resolution are unhealthy.
InstallPlanMissing	A Subscription’s InstallPlan is missing.
InstallPlanPending	A Subscription’s InstallPlan is pending installation.
InstallPlanFailed	A Subscription’s InstallPlan has failed.

7.2. VIEWING OPERATOR STATUS USING THE CLI

You can view Operator status using the CLI.

Procedure

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

```
$ oc describe sub <subscription_name>
```

2. In the command output, find the **Conditions** section:

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

CHAPTER 8. CREATING POLICY FOR OPERATOR INSTALLATIONS AND UPGRADES

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 ClusterServiceVersion (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 OperatorGroup 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.

8.1. UNDERSTANDING OPERATOR INSTALLATION POLICY

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

APIService and **CustomResourceDefinition** resources are always created by OLM using the **cluster-admin** role. A service account associated with an OperatorGroup 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 OperatorGroup 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.

8.1.1. Installation scenarios

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

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

- A cluster administrator creates a new OperatorGroup 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 OperatorGroups that do not specify a service account, the default behavior remains and Operator installs and upgrades are permitted.

- A cluster administrator updates an existing OperatorGroup 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 OperatorGroup 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 OperatorGroup. The default behavior remains and Operator installs and upgrades are permitted.

8.1.2. Installation workflow

When an OperatorGroup is tied to a service account and an Operator is installed or upgraded, OLM uses the following workflow:

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

2. OLM fetches the OperatorGroup tied to this Subscription.

3. OLM determines that the OperatorGroup 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 OperatorGroup.

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.

8.2. SCOPING OPERATOR INSTALLATIONS

To provide scoping rules to Operator installations and upgrades on OLM, associate a service account with an OperatorGroup.

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 RoleBinding(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 in the designated namespace. This OperatorGroup targets the designated namespace to ensure that its tenancy is confined to it. In addition, OperatorGroups allow a user to specify a service account. Specify the ServiceAccount 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 OperatorGroup and therefore to the service account specified.

4. Create a Subscription 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
EOF
```

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

Any Operator tied to this OperatorGroup 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 appropriate errors.

8.2.1. Fine-grained permissions

OLM uses the service account specified in OperatorGroup 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.

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

NOTE

OLM does not currently provide the complete list of errors on the first try, but may be added in a future release.

CHAPTER 9. USING OPERATOR LIFECYCLE MANAGER ON RESTRICTED NETWORKS

When OpenShift Container Platform is installed on restricted networks, Operator Lifecycle Manager (OLM) can no longer use the default OperatorHub sources as they require full Internet connectivity. Cluster administrators can disable those default sources and create local mirrors so that OLM can install and manage Operators from the local sources instead.

9.1. CONFIGURING OPERATORHUB FOR RESTRICTED NETWORKS

Cluster administrators can configure OLM and OperatorHub to use local content in restricted network environments.

Prerequisites

- Cluster administrator access to an OpenShift Container Platform cluster and its internal registry.
- Separate workstation without network restrictions.
- If pushing images to the OpenShift Container Platform cluster's internal registry, the registry must be exposed with a route.
- **podman** version 1.4.4+

Procedure

1. **Disable the default OperatorSources.**
   Add **disableAllDefaultSources: true** to the spec:

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

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

2. **Retrieve package lists.**
   To get the list of packages that are available for the default OperatorSources, run the following **curl** commands from your workstation without network restrictions:

   ```
   $ curl https://quay.io/cnr/api/v1/packages?namespace=redhat-operators > packages.txt
   $ curl https://quay.io/cnr/api/v1/packages?namespace=community-operators >> packages.txt
   $ curl https://quay.io/cnr/api/v1/packages?namespace=certified-operators >> packages.txt
   ```

   Each package in the new **packages.txt** is an Operator that you could add to your restricted network catalog. From this list, you could either pull every Operator or a subset that you would like to expose to users.

3. **Pull Operator content.**
   For a given Operator in the package list, you must pull the latest released content:

   ```
   $ curl https://quay.io/cnr/api/v1/packages/<namespace>/<operator_name>/<release>
   ```

This example uses the etcd Operator:

a. Retrieve the digest:

```
$ curl https://quay.io/cnr/api/v1/packages/community-operators/etcd/0.0.12
```

b. From that JSON, take the digest and use it to pull the gzipped archive:

```
$ curl -XGET https://quay.io/cnr/api/v1/packages/community-
operators/etcd/blobs/sha256/8108475ee5e83a0187d6d0a729451ef1ce6d34c44a868a2001
51c36f3232822b \
    -o etcd.tar.gz
```

c. To pull the information out, you must untar the archive into a **manifests/<operator_name>/** directory with all the other Operators that you want. For example, to untar to an existing directory called **manifests/etcd/**:

```
$ mkdir -p manifests/etcd/ 1
$ tar -xf etcd.tar.gz -C manifests/etcd/
```

1 Create different subdirectories for each extracted archive so that files are not overwritten by subsequent extractions for other Operators.

4. **Break apart bundle.yaml content, if necessary.**
In your new **manifests/<operator_name>** directory, the goal is to get your bundle in the following directory structure:

```
manifests/
└── etcd
    ├── 0.0.12
    │   ├── clusterserviceversion.yaml
    │   └── customresourcedefinition.yaml
    └── package.yaml
```

If you see files already in this structure, you can skip this step. However, if you instead see only a single file called **bundle.yaml**, you must first break this file up to conform to the required structure.

You must separate the CSV content under **data.clusterServiceVersion** (each file in the list), the CRD content under **data.customResourceDefinition** (each file in the list), and the package content under **data.Package** into their own files.

a. For the CSV file creation, find the following lines in the **bundle.yaml** file:

```
data:
  clusterServiceVersions: |
```

Omit those lines, but save a new file consisting of the full CSV resource content beginning with the following lines, removing the prepended **-** character:

Example clusterserviceversion.yaml file snippet

```
apiVersion: operators.coreos.com/v1alpha1
kind: ClusterServiceVersion
[...]
```

b. For the CRD file creation, find the following line in the **bundle.yaml** file:

```
  customResourceDefinitions: |
```

Omit this line, but save new files consisting of each, full CRD resource content beginning with the following lines, removing the prepended **-** character:

Example **customresourcedefinition.yaml** file snippet

```
apiVersion: apiextensions.k8s.io/v1beta1
kind: CustomResourceDefinition
[...]
```

c. For the package file creation, find the following line in the **bundle.yaml** file:

```
  packages: |
```

Omit this line, but save a new file consisting of the package content beginning with the following lines, removing the prepended **-** character, and ending with a **packageName** entry:

Example **package.yaml** file

```
channels:
- currentCSV: etcdoperator.v0.9.4
  name: singlenamespace-alpha
- currentCSV: etcdoperator.v0.9.4-clusterwide
  name: clusterwide-alpha
defaultChannel: singlenamespace-alpha
packageName: etcd
```

5. **Identify images required by the Operators you want to use.**
Inspect the CSV files of each Operator for **image:** fields to identify the pull specs for any images required by the Operator, and note them for use in a later step.

For example, in the following **deployments** spec of an etcd Operator CSV:

```
  spec:
   serviceAccountName: etcd-operator
   containers:
   - name: etcd-operator
     command:
     - etcd-operator
     - --create-crd=false
     image: quay.io/coreos/etcd-
operator@sha256:bd944a211eaf8f31da5e6d69e8541e7cada8f16a9f7a5a570b224789978199
43 1
     env:
     - name: MY_POD_NAMESPACE
  valueFrom:
    fieldRef:
      fieldPath: metadata.namespace
- name: MY_POD_NAME
  valueFrom:
    fieldRef:
      fieldPath: metadata.name
```

1 Image required by Operator.

6. **Create an Operator catalog image.**

   a. Save the following to a Dockerfile, for example named **custom-registry.Dockerfile**:

   ```
   FROM registry.redhat.io/openshift4/ose-operator-registry:v4.2.24 AS builder

   COPY manifests manifests

   RUN /bin/initializer -o ./bundles.db

   FROM registry.access.redhat.com/ubi7/ubi

   COPY --from=builder /registry/bundles.db /bundles.db
   COPY --from=builder /usr/bin/registry-server /registry-server
   COPY --from=builder /bin/grpc_health_probe /bin/grpc_health_probe

   EXPOSE 50051

   ENTRYPOINT ["/registry-server"]

   CMD ["--database", "bundles.db"]
   ```

   b. Use the **podman** command to create and tag the container image from the Dockerfile:

   ```
   $ podman build -f custom-registry.Dockerfile \
       -t <local_registry_host_name>:<local_registry_host_port>/<namespace>/custom-
   registry 1
   ```

   1 Tag the image for the internal registry of the restricted network OpenShift Container Platform cluster and any namespace.

7. **Push the Operator catalog image to a registry.**
   Your new Operator catalog image must be pushed to a registry that the restricted network OpenShift Container Platform cluster can access. This can be the internal registry of the cluster itself or another registry that the cluster has network access to, such as an on-premise Quay Enterprise registry.

   For this example, login and push the image to the internal registry OpenShift Container Platform cluster:

   ```
   $ podman push <local_registry_host_name>:
   <local_registry_host_port>/<namespace>/custom-registry
   ```

8. **Create a CatalogSource pointing to the new Operator catalog image.**

   a. Save the following to a file, for example **my-operator-catalog.yaml**:

```
apiVersion: operators.coreos.com/v1alpha1
kind: CatalogSource
metadata:
  name: my-operator-catalog
  namespace: openshift-marketplace
spec:
  displayName: My Operator Catalog
  sourceType: grpc
  image: <local_registry_host_name>:<local_registry_host_port>/<namespace>/custom-registry:latest
```

   b. Create the CatalogSource resource:

```
$ oc create -f my-operator-catalog.yaml
```

   c. Verify the CatalogSource and package manifest are created successfully:

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

# oc get catalogsource -n openshift-marketplace
NAME DISPLAY TYPE PUBLISHER AGE
my-operator-catalog My Operator Catalog grpc 5s

# oc get packagemanifest -n openshift-marketplace
NAME CATALOG AGE
etcd My Operator Catalog 34s
```

   You should also be able to view them from the **OperatorHub** page in the web console.

9. **Mirror the images required by the Operators you want to use.**

   a. Determine the images defined by the Operator(s) that you are expecting. This example uses the etcd Operator, requiring the **quay.io/coreos/etcd-operator** image.

> IMPORTANT
>
> This procedure only shows mirroring Operator images themselves and not Operand images, which are the components that an Operator manages. Operand images must be mirrored as well; see each Operator's documentation to identify the required Operand images.

   b. To use mirrored images, you must first create an ImageContentSourcePolicy for each image to change the source location of the Operator catalog image. For example:

```
apiVersion: operator.openshift.io/v1alpha1
kind: ImageContentSourcePolicy
metadata:
  name: etcd-operator
```

```
spec:
  repositoryDigestMirrors:
  - mirrors:
    - <local_registry_host_name>:<local_registry_host_port>/coreos/etcd-operator
    source: quay.io/coreos/etcd-operator
```

c. Use the **oc image mirror** command from your workstation without network restrictions to pull the image from the source registry and push to the internal registry without being stored locally:

```
$ oc image mirror quay.io/coreos/etcd-operator \
    <local_registry_host_name>:<local_registry_host_port>/coreos/etcd-operator
```

You can now install the Operator from the OperatorHub on your restricted network OpenShift Container Platform cluster.

Additional resources

- For details on exposing your OpenShift Container Platform cluster's internal registry to off-cluster access, see Exposing the registry.
- For details on accessing the internal registry, see Accessing the registry.

CHAPTER 10. CRDS

10.1. 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).

10.1.1. 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 **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 cluster's RBAC policy 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 an Operator's lifecycle, 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.

10.1.2. 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 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.

10.1.3. 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, 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. The 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
```

10.1.4. Creating Custom Resources from a file

After a Custom Resource Definition (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 Custom Resource Definition.

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
```

10.1.5. 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

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
$ oc get ct -o yaml

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

10.2. MANAGING RESOURCES FROM CUSTOM RESOURCE DEFINITIONS

This guide describes how developers can manage Custom Resources (CRs) that come from Custom Resource Definitions (CRDs).

10.2.1. 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 **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 an Operator’s lifecycle, 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.

10.2.2. Creating Custom Resources from a file

After a Custom Resource Definition (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 Custom Resource Definition.

   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
   ```

10.2.3. 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

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
```

```
$ oc get ct -o yaml

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

11.1. 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: the Operator SDK (the **operator-sdk** CLI tool and **controller-runtime** library API) and the Operator Lifecycle Manager (OLM).

NOTE

OpenShift Container Platform 4 supports Operator SDK v0.7.0 or later.

11.1.1. 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:

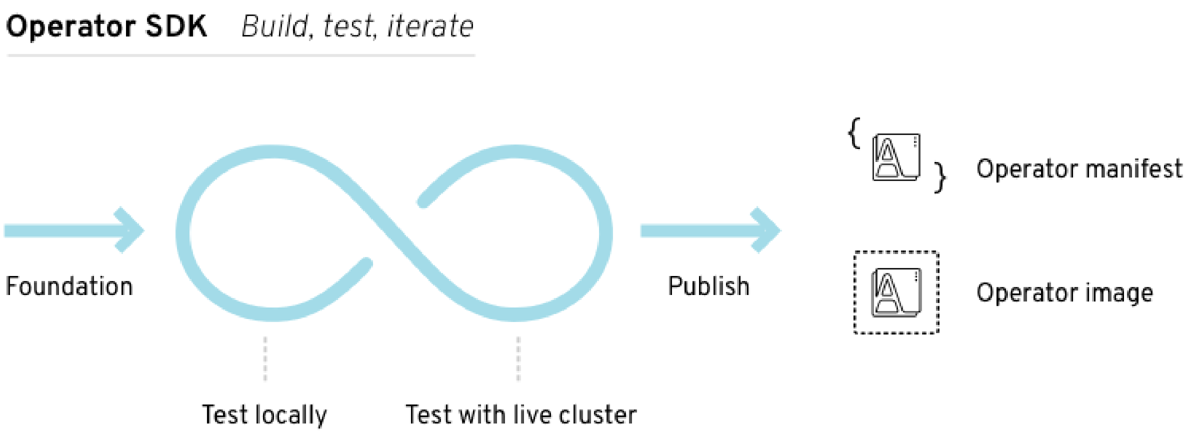
- High-level APIs and abstractions to write the operational logic more intuitively
- Tools for scaffolding and code generation to quickly bootstrap a new project
- Extensions to cover common Operator use cases

11.1.1.1. 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 11.1. 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.

11.1.1.2. 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: ""})
```

11.1.1.3. 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.

11.1.2. 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.

11.1.2.1. Installing from GitHub release

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

Prerequisites

- **docker** v17.03+
- OpenShift CLI (**oc**) v4.1+ installed
- Access to a cluster based on Kubernetes v1.11.3+
- Access to a container registry

Procedure

1. Set the release version variable:

```
RELEASE_VERSION=v0.8.0
```

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 maintainer's public key on your workstation, you will get the following error:

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

11.1.2.2. Installing from Homebrew

You can install the SDK CLI using Homebrew.

Prerequisites

- Homebrew
- **docker** v17.03+
- OpenShift CLI (**oc**) v4.1+ installed
- Access to a cluster based on Kubernetes v1.11.3+
- 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
   ```

11.1.2.3. Compiling and installing from source

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

Prerequisites

- dep v0.5.0+
- Git
- Go v1.10+
- **docker** v17.03+
- OpenShift CLI (**oc**) v4.1+ installed
- Access to a cluster based on Kubernetes v1.11.3+
- 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
```

11.1.3. Building a Go-based Memcached 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 4.2
- Access to the cluster using an account with **cluster-admin** permissions
- OpenShift CLI (**oc**) v4.1+ 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 --dep-manager dep
$ 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. **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** CR:

- Create a Memcached Deployment if it does not exist.
- Ensure that the Deployment size is the same as specified by the **Memcached** CR spec.
- Update the **Memcached** CR status with the names of the Memcached Pods.

The next 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 Deployments, 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 **Reconcile()** the reconcile **Request** may be requeued and the loop may 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
```

4. **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:

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

E. Setup RBAC and deploy **memcached-operator**:

```
$ 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 **memcached-operator** is up and running:

```
$ oc get deployment
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 up local --namespace=default
```

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

5. **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**:

```
$ cat deploy/crds/cache_v1alpha1_memcached_cr.yaml
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 **memcached-operator** creates the Deployment for the CR:

```
$ oc get deployment
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 and CR status to confirm the status is updated with the **memcached** Pod names:

```
$ 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

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

6. **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
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
NAME                DESIRED   CURRENT   UP-TO-DATE   AVAILABLE   AGE
example-memcached   4         4         4            4           5m
```

7. **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
```

11.1.4. Managing a Memcached Operator using the Operator Lifecycle Manager

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

The 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 4.2 Preview OLM enabled
- 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 the OLM to function.

   You can use the following command to generate CSV manifests:

   ```
   $ operator-sdk olm-catalog gen-csv --csv-version 0.0.1
   ```

   NOTE

   This command is run from the **memcached-operator**/ directory that was created when you built the Memcached Operator.

   For the purpose of this guide, we will continue with this predefined manifest file for the next steps. You can alter the image field within this manifest to reflect the image you built in previous steps, but it is unnecessary.

NOTE

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

2. **Deploy the Operator.**

   a. Create an OperatorGroup that specifies the namespaces that the Operator will target. Create the following OperatorGroup 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
   ```

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

   ```
   $ curl -Lo memcachedoperator.0.0.1.csv.yaml
   https://raw.githubusercontent.com/operator-framework/getting-
   started/master/memcachedoperator.0.0.1.csv.yaml
   $ oc apply -f memcachedoperator.0.0.1.csv.yaml
   $ oc get csv memcachedoperator.v0.0.1 -n default -o json | jq '.status'
   ```

   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 type that the Operator manages:

   ```
   $ oc create -f deploy/crds/cache_v1alpha1_memcached_crd.yaml
   $ oc create -f deploy/service_account.yaml
   $ oc create -f deploy/role.yaml
   $ oc create -f deploy/role_binding.yaml
   ```

   NOTE

   These files were generated into the **deploy/** directory by the Operator SDK when you built the Memcached Operator.

   Because the 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.

3. **Create an application instance.**
   The Memcached Operator is now running in the **default** namespace. Users interact with Operators via instances of **CustomResources**; in this case, the resource has the kind **Memcached**. Native Kubernetes RBAC also applies to **CustomResources**, providing administrators control over who can interact with each Operator.

Creating instances of Memcached in this namespace will now trigger the Memcached Operator to instantiate pods running the memcached server that are managed by the Operator. The more **CustomResources** you create, the more unique instances of Memcached 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
NAME                      AGE
memcached-for-drupal      22s
memcached-for-wordpress   27s

$ oc get pods
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. **Update an application.**
   Manually apply an update to the Operator by creating a new Operator manifest with a **replaces** field that references the old Operator manifest. The OLM ensures that all resources being managed by the old Operator have their ownership moved to the new Operator without fear of any programs stopping execution. It is up to the Operators themselves to execute any data migrations required to upgrade resources to run under a new version of the Operator.

   The following command demonstrates applying a new Operator manifest file using a new version of the Operator and shows that the pods remain executing:

```
$ curl -Lo memcachedoperator.0.0.2.csv.yaml https://raw.githubusercontent.com/operator-framework/getting-started/master/memcachedoperator.0.0.2.csv.yaml
$ oc apply -f memcachedoperator.0.0.2.csv.yaml
$ oc get pods
NAME                                       READY     STATUS    RESTARTS   AGE
memcached-app-operator-66b5777b79-pnsfj    1/1       Running   0          3s
memcached-for-drupal-5476487c46-qbd66      1/1       Running   0          14m
memcached-for-wordpress-65b75fd8c9-7b9x7   1/1       Running   0          14m
```

11.1.5. Additional resources

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

11.2. 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.

11.2.1. 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's 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.

11.2.1.1. 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 11.1. 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 **k8s_status** Ansible module by default, which includes**condition** information to the CR's **status**.
annotations	Kubernetes-specific annotations to be appended to the CR.

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

Table 11.2. 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: "foo.example.com/v1alpha1"
kind: "Foo"
metadata:
  name: "example"
annotations:
  ansible.operator-sdk/reconcile-period: "30s"
```

11.2.1.2. Watches file

The Watches file contains a list of mappings from Custom Resources (CRs), identified by its **Group**, **Version**, and **Kind**, to an Ansible role or playbook. The Operator expects this mapping file in a predefined location, **/opt/ansible/watches.yaml**.

Table 11.3. Watches 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 simply 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 file

```
- version: v1alpha1 ①
  group: foo.example.com
  kind: Foo
  role: /opt/ansible/roles/Foo

- version: v1alpha1 ②
  group: bar.example.com
  kind: Bar
  playbook: /opt/ansible/playbook.yml

- version: v1alpha1 ③
  group: baz.example.com
  kind: Baz
  playbook: /opt/ansible/baz.yml
  reconcilePeriod: 0
  manageStatus: false
```

① Simple example mapping **Foo** to the **Foo** role.

② Simple example mapping **Bar** to a playbook.

③ More complex example for the **Baz** kind. Disables re-queuing and managing the CR status in the playbook.

11.2.1.2.1. Advanced options

Advanced features can be enabled by adding them to your Watches file per GVK (group, version, and kind). 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 Custom Resource (CR). The options that can be overridden have the annotation specified below.

Table 11.4. Advanced Watches 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’s **status** section.		**true**
Watch dependent resources	**watchDependentResources**	Allows the Operator to dynamically watch resources that are created by Ansible.		**true**

Feature	YAML key	Description	Annotation for override	Default value
Watch cluster-scoped resources	**watchClusterScopedResources**	Allows the Operator to watch cluster-scoped resources that are created by Ansible.		**false**
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 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
```

11.2.1.3. 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 }}"
```

11.2.1.4. 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.

11.2.2. 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.

11.2.2.1. Installing from GitHub release

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

Prerequisites

- **docker** v17.03+
- OpenShift CLI (**oc**) v4.1+ installed
- Access to a cluster based on Kubernetes v1.11.3+
- Access to a container registry

Procedure

1. Set the release version variable:

   ```
   RELEASE_VERSION=v0.8.0
   ```

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 maintainer’s public key on your workstation, you will get the following error:

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

11.2.2.2. Installing from Homebrew

You can install the SDK CLI using Homebrew.

Prerequisites

- Homebrew
- **docker** v17.03+
- OpenShift CLI (**oc**) v4.1+ installed
- Access to a cluster based on Kubernetes v1.11.3+
- 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
   ```

11.2.2.3. Compiling and installing from source

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

Prerequisites

- dep v0.5.0+
- Git
- Go v1.10+
- **docker** v17.03+
- OpenShift CLI (**oc**) v4.1+ installed
- Access to a cluster based on Kubernetes v1.11.3+
- 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
```

11.2.3. 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 4.2) using an account with **cluster-admin** permissions
- OpenShift CLI (**oc**) v4.1+ installed
- **ansible** v2.6.0+
- **ansible-runner** v1.1.0+
- **ansible-runner-http** v1.0.0+

Procedure

1. **Create a new Operator project,** either namespace-scoped or cluster-scoped, using the **operator-sdk new** command. Choose one of the following:

    a. A *namespace-scoped Operator* (the default) 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 its 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 APIVersion **example.com/v1apha1** and Kind **Memcached**.

    b. A *cluster-scoped Operator* watches and manages resources cluster-wide, which can be useful in certain cases. For example, the **cert-manager** operator is often deployed with cluster-scoped permissions and watches so that it can manage issuing certificates for an entire cluster.
    To create your **memcached-operator** project to be cluster-scoped and change to its directory, use the following commands:

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

    Using the **--cluster-scoped** flag scaffolds the new Operator with the following modifications:

    - **deploy/operator.yaml**: Set **WATCH_NAMESPACE=""** instead of setting it to the Pod's namespace.
    - **deploy/role.yaml**: Use **ClusterRole** instead of **Role**.
    - **deploy/role_binding.yaml**:

- Use **ClusterRoleBinding** instead of **RoleBinding**.
- Set the subject namespace to **REPLACE_NAMESPACE**. This must be changed to the namespace in which the Operator is deployed.

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 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) the Operator will be watching. Deploy the **Memcached** CRD:

```
$ oc create -f deploy/crds/cache_v1alpha1_memcached_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. If you created your Operator using the **--cluster-scoped=true** flag, update the service account namespace in the generated **ClusterRoleBinding** to match where you are deploying your Operator:

      ```
      $ export OPERATOR_NAMESPACE=$(oc config view --minify -o
      jsonpath='{.contexts[0].context.namespace}')
      $ sed -i "s|REPLACE_NAMESPACE|$OPERATOR_NAMESPACE|g"
      deploy/role_binding.yaml
      ```

      If you are performing these steps on OSX, use the following commands instead:

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

      iv. Deploy the **memcached-operator**:

      ```
      $ 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
      ```

      v. Verify that the **memcached-operator** 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 up local
```

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

```
$ operator-sdk up 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
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
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
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
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
```

11.2.4. 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.

11.2.4.1. Installing the k8s Ansible module

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

Procedure

1. Install Ansible 2.6+:

```
$ sudo yum install ansible
```

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

```
$ pip install openshift
```

11.2.4.2. 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. Initialize a new Ansible-based Operator project:

   ```
   $ operator-sdk new --type ansible --kind Foo --api-version foo.example.com/v1alpha1 foo-operator
   Create foo-operator/tmp/init/galaxy-init.sh
   Create foo-operator/tmp/build/Dockerfile
   Create foo-operator/tmp/build/test-framework/Dockerfile
   Create foo-operator/tmp/build/go-test.sh
   Rendering Ansible Galaxy role [foo-operator/roles/Foo]...
   Cleaning up foo-operator/tmp/init
   Create foo-operator/watches.yaml
   Create foo-operator/deploy/rbac.yaml
   Create foo-operator/deploy/crd.yaml
   Create foo-operator/deploy/cr.yaml
   Create foo-operator/deploy/operator.yaml
   Run git init ...
   Initialized empty Git repository in /home/dymurray/go/src/github.com/dymurray/opsdk/foo-operator/.git/
   Run git init done
   ```

   ```
   $ cd foo-operator
   ```

2. Modify the **roles/Foo/tasks/main.yml** file with the desired Ansible logic. This example creates and deletes a namespace with the switch of a variable.

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

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

3. Modify the **roles/Foo/defaults/main.yml** file to set **state** to **present** by default.

   ```
   state: present
   ```

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

   ```
   - hosts: localhost
     roles:
       - Foo
   ```

5. Run the playbook:

```
$ ansible-playbook playbook.yml
 [WARNING]: provided hosts list is empty, only localhost is available. Note that the implicit
localhost does not match 'all'

PLAY [localhost] ****************************************************************************

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

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

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

6. Check that the namespace was created:

```
$ oc get namespace
NAME          STATUS    AGE
default       Active    28d
kube-public   Active    28d
kube-system   Active    28d
test          Active    3s
```

7. Rerun the playbook setting **state** to **absent**:

```
$ ansible-playbook playbook.yml --extra-vars state=absent
 [WARNING]: provided hosts list is empty, only localhost is available. Note that the implicit
localhost does not match 'all'

PLAY [localhost] ****************************************************************************

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

Task [Foo : set test namespace to absent]
changed: [localhost]

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

8. Check that the namespace was deleted:

```
$ oc get namespace
NAME          STATUS    AGE
default       Active    28d
kube-public   Active    28d
kube-system   Active    28d
```

11.2.4.3. Testing the k8s Ansible module inside an Operator

After you are familiar 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 file.

11.2.4.3.1. 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 up 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 **up 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: foo.example.com
     kind: Foo
   #  role: /opt/ansible/roles/Foo
     role: /home/user/foo-operator/Foo
   ```

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

   ```
   $ oc create -f deploy/crds/foo_v1alpha1_foo_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 **up local** command:

   ```
   $ operator-sdk up local
   [...]
   INFO[0000] Starting to serve on 127.0.0.1:8888
   INFO[0000] Watching foo.example.com/v1alpha1, Foo, default
   ```

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

   ```
   apiVersion: "foo.example.com/v1alpha1"
   kind: "Foo"
   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 sane defaults for the Operator.

5. Create a CR instance of **Foo** 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
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: "foo.example.com/v1alpha1"
kind: "Foo"
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
NAME          STATUS    AGE
default       Active    28d
kube-public   Active    28d
kube-system   Active    28d
```

11.2.4.3.2. 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 **foo-operator** image and push it to a registry:

```
$ operator-sdk build quay.io/example/foo-operator:v0.0.1
$ podman push quay.io/example/foo-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/foo-operator:v0.0.1|g' deploy/operator.yaml
```

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

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

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

```
$ oc create -f deploy/crds/foo_v1alpha1_foo_crd.yaml # if CRD doesn't 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 **foo-operator** is up and running:

```
$ oc get deployment
NAME                     DESIRED   CURRENT   UP-TO-DATE   AVAILABLE   AGE
foo-operator       1         1         1            1          1m
```

11.2.5. Managing Custom Resource status using the k8s_status Ansible module

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. 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 file with a **manageStatus** field set to **false**:

```
- version: v1
  group: api.example.com
  kind: Foo
  role: /opt/ansible/roles/Foo
  manageStatus: false
```

2. Then, use the **k8s_status** Ansible module to update the subresource. For example, to update with key **foo** and value **bar**, **k8s_status** can be used as shown:

```
- k8s_status:
  api_version: app.example.com/v1
  kind: Foo
  name: "{{ meta.name }}"
  namespace: "{{ meta.namespace }}"
  status:
    foo: bar
```

Additional resources

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

11.2.5.1. Using the k8s_status Ansible module when testing locally

If your Operator takes advantage of the **k8s_status** Ansible module and you want to test the Operator locally with the **operator-sdk up local** command, you must install the module in a location that Ansible expects. This is done with the **library** configuration option for Ansible.

For this example, assume the user is placing third-party Ansible modules in the **/usr/share/ansible/library/** directory.

Procedure

1. To install the **k8s_status** module, set the **ansible.cfg** file to search in the **/usr/share/ansible/library/** directory for installed Ansible modules:

```
$ echo "library=/usr/share/ansible/library/" >> /etc/ansible/ansible.cfg
```

2. Add the **k8s_status.py** file to the **/usr/share/ansible/library/** directory:

```
$ wget https://raw.githubusercontent.com/openshift/ocp-release-operator-sdk/master/library/k8s_status.py -O /usr/share/ansible/library/k8s_status.py
```

11.2.6. 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

11.3. 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.

11.3.1. 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's 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 object's spec field is a list of configuration options that are typically described in Helm's **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 chart's templates where 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). And because it obeys RBAC, you can more easily prevent production changes.

11.3.2. 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.

11.3.2.1. Installing from GitHub release

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

Prerequisites

- **docker** v17.03+
- OpenShift CLI (**oc**) v4.1+ installed
- Access to a cluster based on Kubernetes v1.11.3+
- Access to a container registry

Procedure

1. Set the release version variable:

   ```
   RELEASE_VERSION=v0.8.0
   ```

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 maintainer's public key on your workstation, you will get the following error:

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

11.3.2.2. Installing from Homebrew

You can install the SDK CLI using Homebrew.

Prerequisites

- Homebrew
- **docker** v17.03+
- OpenShift CLI (**oc**) v4.1+ installed
- Access to a cluster based on Kubernetes v1.11.3+
- 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
   ```

11.3.2.3. Compiling and installing from source

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

Prerequisites

- dep v0.5.0+
- Git
- Go v1.10+
- **docker** v17.03+
- OpenShift CLI (**oc**) v4.1+ installed
- Access to a cluster based on Kubernetes v1.11.3+
- 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
```

11.3.3. 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 4.2) using an account with **cluster-admin** permissions
- OpenShift CLI (**oc**) v4.1+ installed

Procedure

1. **Create a new Operator project,** either namespace-scoped or cluster-scoped, using the **operator-sdk new** command. Choose one of the following:

    a. A *namespace-scoped Operator* (the default) 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 APIVersion **example.com/v1apha1** and Kind **Nginx**.

    b. A *cluster-scoped Operator* watches and manages resources cluster-wide, which can be useful in certain cases. For example, the **cert-manager** operator is often deployed with cluster-scoped permissions and watches so that it can manage issuing certificates for an entire cluster.

To create your **nginx-operator** project to be cluster-scoped, use the following command:

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

Using the **--cluster-scoped** flag scaffolds the new Operator with the following modifications:

- **deploy/operator.yaml**: Set **WATCH_NAMESPACE=""** instead of setting it to the Pod's namespace.
- **deploy/role.yaml**: Use **ClusterRole** instead of **Role**.
- **deploy/role_binding.yaml**:
  - Use **ClusterRoleBinding** instead of **RoleBinding**.
  - Set the subject namespace to **REPLACE_NAMESPACE**. This must be changed to the namespace in which the Operator is deployed.

2. **Customize the Operator logic.**
   For this example, the **nginx-operator** executes the following reconciliation logic for each **Nginx** Custom Resource (CR):

   - Create a Nginx Deployment if it does not exist.
   - Create a Nginx Service if it does not exist.
   - Create a 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, take a moment to review the Helm Chart developer documentation.

   b. **Understand the Nginx CR spec.**

Helm uses a concept called values to provide customizations to a Helm chart's defaults, which are defined in the Helm chart's **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_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_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 needs to know about the new custom resource definition (CRD) 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
$ docker 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. If you created your Operator using the **--cluster-scoped=true** flag, update the service account namespace in the generated **ClusterRoleBinding** to match where you are deploying your Operator:

```
$ export OPERATOR_NAMESPACE=$(oc config view --minify -o
jsonpath='{.contexts[0].context.namespace}')
$ sed -i "s|REPLACE_NAMESPACE|$OPERATOR_NAMESPACE|g"
deploy/role_binding.yaml
```

If you are performing these steps on OSX, use the following commands instead:

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

iv. Deploy the **nginx-operator**:

```
$ 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
```

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

```
$ oc get deployment
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 up local** command, the SDK looks in your local file system for this path.

i. Create a symlink at this location to point to your Helm chart's path:

```
$ 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 up local
```

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

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

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

```
$ oc apply -f deploy/crds/example_v1alpha1_nginx_cr.yaml
```

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

```
$ oc get deployment
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
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
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_v1alpha1_nginx_cr.yaml
apiVersion: "example.com/v1alpha1"
kind: "Nginx"
metadata:
  name: "example-nginx"
spec:
  replicaCount: 3

$ oc apply -f deploy/crds/example_v1alpha1_nginx_cr.yaml
```

Confirm that the Operator changes the Deployment size:

```
$ oc get deployment
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
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_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
```

11.3.4. Additional resources

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

11.4. GENERATING A CLUSTERSERVICEVERSION (CSV)

A *ClusterServiceVersion* (CSV) is a YAML manifest created from Operator metadata that assists the 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 like 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 **olm-catalog gen-csv** subcommand to generate a *ClusterServiceVersion* (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 Operator Lifecycle Manager (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's, and a new CSV is generated when upgrading Operator versions. Operator authors can use the **--csv-version** flag to have their Operators' state encapsulated in a CSV with the supplied semantic version:

```
$ operator-sdk olm-catalog gen-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**.

11.4.1. How CSV generation works

An Operator project’s **deploy/** directory 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 CSV. The following command:

```
$ operator-sdk olm-catalog gen-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 **olm-catalog gen-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 ClusterServiceVersion 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.

11.4.2. 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. Defaults to **deploy/operator.yaml**.
crd-cr-path-list (string(, string)*)	A list of CRD and CR manifest file paths. Defaults to **[deploy/crds/*_{crd,cr}.yaml]**.
rbac-path-list (string(, string)*)	A list of RBAC role manifest file paths. Defaults to **[deploy/role.yaml]**.

11.4.3. Manually-defined CSV fields

Many CSV fields cannot be populated using generated, non-SDK-specific manifests. These fields are mostly human-written, English metadata about the Operator and various Custom Resource Definitions (CRDs).

Operator authors must directly modify their CSV YAML file, adding personalized data to the following required fields. The Operator SDK gives a warning CSV generation when a lack of data in any of the required fields is detected.

Table 11.5. 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 Operator’s capability level according to the Operator maturity model, for example **Seamless Upgrades**.
spec.displayName	A public name to identify the Operator.
spec.description	A short description of the Operator’s functionality.
spec.keywords	Keywords describing the operator.
spec.maintainers	Human or organizational entities maintaining the Operator, with a **name** and **email**.
spec.provider	The Operators' provider (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**.

Field	Description
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 Pods and StatefulSets. ● **specDescriptors**: UI hints for inputs and outputs of the Operator.

Table 11.6. 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, for example **alpha**, **beta**, **stable**.

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; such Operator SDK functionality will be addressed in a future design document.

Additional resources

- Operator maturity model

11.4.4. 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 olm-catalog gen-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.

11.4.5. Understanding your Custom Resource Definitions (CRDs)

There are two types of Custom Resource Definitions (CRDs) that your Operator may use: ones that are *owned* by it and ones that it depends on, which are *required*.

11.4.5.1. Owned CRDs

The 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 ReplicaSets in another. Each one should be listed out in the CSV file.

Table 11.7. 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

Field	Description	Required/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, ConfigMaps that store internal state that should not be modified by a user should not appear here.	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 ConfigMap 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 ConfigMap, and orchestrates Services, StatefulSets, Pods and ConfigMaps:

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.
```

11.4.5.2. 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.

The 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 11.8. 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
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.
```

11.4.5.3. CRD templates

Users of your Operator will need to be aware of which options are required versus optional. You can provide templates for each of your 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>"}}}]
```

11.4.6. Understanding your API services

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

11.4.6.1. Owned APIServices

When a CSV owns an APIService, it is responsible for describing the deployment of the extension **api-server** that backs it and the **group-version-kinds** it provides.

An APIService 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 11.9. Owned APIService fields

Field	Description	Required/Optional
Group	Group that the APIService provides, for example **database.example.com**.	Required
Version	Version of the APIService, for example **v1alpha1**.	Required
Kind	A kind that the APIService is expected to provide.	Required
Name	The plural name for the APIService provided	Required
DeploymentName	Name of the deployment defined by your CSV that corresponds to your APIService (required for owned APIServices). During the CSV pending phase, the OLM Operator searches your CSV’s InstallStrategy 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 APIService name, for example **MongoDB Standalone**.	Required
Description	A short description of how this APIService is used by the Operator or a description of the functionality provided by the APIService.	Required
Resources	Your APIServices 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, ConfigMaps that store internal state that should not be modified by a user should not appear here.	Optional

Field	Description	Required/Optional
SpecDescriptors, **StatusDescriptors**, and **ActionDescriptors**	Essentially the same as for owned CRDs.	Optional

11.4.6.1.1. APIService Resource Creation

The Operator Lifecycle Manager (OLM) is responsible for creating or replacing the Service and APIService resources for each unique owned APIService:

- Service Pod selectors are copied from the CSV deployment matching the APIServiceDescription's **DeploymentName**.
- A new CA key/cert pair is generated for each installation and the base64-encoded CA bundle is embedded in the respective APIService resource.

11.4.6.1.2. APIService Serving Certs

The OLM handles generating a serving key/cert pair whenever an owned APIService is being installed. The serving certificate has a 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 APIService resource.

The cert 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 APIServiceDescription's **DeploymentName** field.

If one does not already exist, a VolumeMount with a matching name is also appended to all containers of that deployment. This allows users to define a VolumeMount with the expected name to accommodate any custom path requirements. The generated VolumeMount's path defaults to **/apiserver.local.config/certificates** and any existing VolumeMounts with the same path are replaced.

11.4.6.2. Required APIServices

The OLM ensures all required CSVs have an APIService that is available and all expected **group-version-kinds** are discoverable before attempting installation. This allows a CSV to rely on specific kinds provided by APIServices it does not own.

Table 11.10. Required APIService fields

Field	Description	Required/Optional
Group	Group that the APIService provides, for example **database.example.com**.	Required
Version	Version of the APIService, for example **v1alpha1**.	Required
Kind	A kind that the APIService is expected to provide.	Required

Field	Description	Required/Optional
DisplayName	A human readable version of your APIService name, for example **MongoDB Standalone**.	Required
Description	A short description of how this APIService is used by the Operator or a description of the functionality provided by the APIService.	Required

11.5. 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.

11.5.1. 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.

11.5.2. 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.

11.5.2.1. 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 generated Operator's **cmd/manager/main.go** file, change the value of **metricsPort** in the line **var metricsPort int32 = 8383**.

11.5.3. ServiceMonitor resources

A ServiceMonitor is a Custom Resource Definition (CRD) 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 Custom Resource (CR) based on it.

Additional resources

- See the Prometheus Operator documentation for more information about the ServiceMonitor CRD.

11.5.3.1. Creating ServiceMonitor resources

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.
    }
    ...
}
```

11.6. 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 (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 how it takes for the leader Pod to be deleted from the node and step down (default **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.

11.6.1. 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 ConfigMap 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 Operator's namespace.

11.6.2. 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 since it cannot detect the Operator's namespace in order to create the ConfigMap for leader election. You can override this namespace by setting the Manager's **LeaderElectionNamespace** option.

11.7. OPERATOR SDK CLI REFERENCE

This guide documents the Operator SDK CLI commands and their syntax:

```
$ operator-sdk <command> [<subcommand>] [<argument>] [<flags>]
```

11.7.1. build

The **operator-sdk build** command compiles the code and builds the executables. After **build** completes, the image is built locally in **docker**. It must then be pushed to a remote registry.

Table 11.11. **build** arguments

Argument	Description
<image>	The container image to be built, e.g., **quay.io/example/operator:v0.0.1**.

Table 11.12. **build** flags

Flag	Description
--enable-tests (bool)	Enable in-cluster testing by adding test binary to the image.
--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.

Example output

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

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
```

11.7.2. completion

The **operator-sdk completion** command generates shell completions to make issuing CLI commands quicker and easier.

Table 11.13. **completion** subcommands

Subcommand	Description
bash	Generate bash completions.
zsh	Generate zsh completions.

Table 11.14. **completion** flags

Flag	Description
-h, --help	Usage help output.

Example output

```
$ operator-sdk completion bash

# bash completion for operator-sdk                         -*- shell-script -*-
...
# ex: ts=4 sw=4 et filetype=sh
```

11.7.3. 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 11.15. **print-deps** flags

Flag	Description
--as-file	Print packages and versions in **Gopkg.toml** format.

Example output

```
$ operator-sdk print-deps --as-file
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"
...
```

11.7.4. generate

The **operator-sdk generate** command invokes a specific generator to generate code as needed.

Table 11.16. **generate** subcommands

Subcommand	Description
k8s	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.

Example output

```
$ tree pkg/apis/app/v1alpha1/
pkg/apis/app/v1alpha1/
├── appservice_types.go
├── doc.go
├── register.go

$ operator-sdk generate k8s
Running code-generation for Custom Resource (CR) group versions: [app:v1alpha1]
Generating deepcopy funcs

$ tree pkg/apis/app/v1alpha1/
pkg/apis/app/v1alpha1/
├── appservice_types.go
├── doc.go
├── register.go
└── zz_generated.deepcopy.go
```

11.7.5. olm-catalog

The **operator-sdk olm-catalog** is the parent command for all Operator Lifecycle Manager (OLM) Catalog-related commands.

11.7.5.1. gen-csv

The **gen-csv** subcommand writes a Cluster Service Version (CSV) manifest and optionally Custom Resource Definition (CRD) files to **deploy/olm-catalog/<operator_name>/<csv_version>**.

Table 11.17. **olm-catalog gen-csv** flags

Flag	Description
--csv-version (string)	Semantic version of the CSV manifest. Required.
--from-version (string)	Semantic version of CSV manifest to use as a base for a new version.
--csv-config (string)	Path to CSV configuration file. Default: **deploy/olm-catalog/csv-config.yaml**.
--update-crds	Updates CRD manifests in **deploy/<operator_name>/<csv_version>** using the latest CRD manifests.

Example output

```
$ operator-sdk olm-catalog gen-csv --csv-version 0.1.0 --update-crds
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
```

11.7.6. 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 11.18. **new** arguments

Argument	Description
<project_name>	Name of the new project.

Table 11.19. **new** flags

Flag	Description
--api-version	CRD **APIVersion** in the format **$GROUP_NAME/$VERSION**, for example **app.example.com/v1alpha1**. Used with **ansible** or **helm** types.
--dep-manager [dep|modules]	Dependency manager the new project will use. Used with **go** type. (Default: **modules**)
--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.
--skip-git-init	Do not initialize the directory as a Git repository.
--type	Type of Operator to initialize: **go**, **ansible** or **helm**. (Default: **go**)

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
```

11.7.7. 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 11.20. **add** subcommands

Subcommand	Description
api	Adds a new API definition for a new Custom Resource (CR) under **pkg/apis** and generates the Customer Resource Definition (CRD) and Custom Resource (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.
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 11.21. **add api** flags

Flag	Description
--api-version (string)	CRD **APIVersion** in the format **$GROUP_NAME/$VERSION** (e.g., **app.example.com/v1alpha1**).
--kind (string)	CRD **Kind** (e.g., **AppService**).

Example **add api** output

```
$ operator-sdk add api --api-version app.example.com/v1alpha1 --kind AppService
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
pkg/apis/
├── addtoscheme_app_appservice.go
├── apis.go
```

```
 └── app
  └── v1alpha1
   ├── doc.go
   ├── register.go
   ├── types.go
```

Example add controller output

```
$ operator-sdk add controller --api-version app.example.com/v1alpha1 --kind AppService
Create pkg/controller/appservice/appservice_controller.go
Create pkg/controller/add_appservice.go

$ tree pkg/controller
pkg/controller/
├── add_appservice.go
├── appservice
│   └── appservice_controller.go
└── controller.go
```

Example add crd output

```
$ operator-sdk add crd --api-version app.example.com/v1alpha1 --kind AppService
Generating Custom Resource Definition (CRD) files
Create deploy/crds/app_v1alpha1_appservice_crd.yaml
Create deploy/crds/app_v1alpha1_appservice_cr.yaml
```

11.7.8. test

The **operator-sdk test** command can test the Operator locally.

11.7.8.1. local

The **local** subcommand runs Go tests built using the Operator SDK's test framework locally.

Table 11.22. **test local** arguments

Arguments	Description
<test_location> (string)	Location of e2e test files (e.g., **./test/e2e/**).

Table 11.23. **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**.

Flags	Description
--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 (e.g., **operator-test**). Default: **""**
--go-test-flags (string)	Extra arguments to pass to **go test** (e.g., **-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.

Example output

```
$ operator-sdk test local ./test/e2e/

# Output:
ok   github.com/operator-framework/operator-sdk-samples/memcached-operator/test/e2e 20.410s
```

11.7.9. up

The **operator-sdk up** command has subcommands that can launch the Operator in various ways.

11.7.9.1. local

The **local** subcommand launches the Operator on the local machine by building the Operator binary with the ability to access a Kubernetes cluster using a **kubeconfig** file.

Table 11.24. **up local** arguments

Arguments	Description
--kubeconfig (string)	The file path to a Kubernetes configuration file. Defaults: **$HOME/.kube/config**
--namespace (string)	The namespace where the Operator watches for changes. Default: **default**
--operator-flags	Flags that the local Operator may need. Example: **--flag1 value1 --flag2=value2**

Arguments	Description
-h, --help	Usage help output.

Example output

```
$ operator-sdk up 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 up 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 up 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.

11.8. APPENDICES

11.8.1. 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.

11.8.1.1. 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 Definitions under **pkg/apis/** and starts all controllers under **pkg/controllers/**.
pkg/apis/	Contains the directory tree that defines the APIs of the Custom Resource Definitions (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.

File/folders	Purpose
pkg/controller	This **pkg** contains the controller implementations. Users are expected to edit the **pkg/controller/<kind>/<kind>_controller.go** files to define the controller's reconcile logic 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.

11.8.1.2. 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**, and Helm chart location.