OpenShift Container Platform 4.17

Extensions

Working with extensions in OpenShift Container Platform using Operator Lifecycle Manager (OLM) v1. OLM v1 is a Technology Preview feature only.

Last Updated: 2024-09-27

OpenShift Container Platform 4.17 Extensions

Working with extensions in OpenShift Container Platform using Operator Lifecycle Manager (OLM) v1. OLM v1 is a Technology Preview feature only.

Abstract

This document provides information about installing, managing, and configuring extensions and Operators on OpenShift Container Platform. Operator Lifecycle Manager (OLM) v1 is a Technology Preview feature only.

Table of Contents

CHAPTER 1. EXTENSIONS OVERVIEW

IMPORTANT

Operator Lifecycle Manager (OLM) v1 is a Technology Preview feature only. Technology Preview features are not supported with Red Hat production service level agreements (SLAs) and might not be functionally complete. Red Hat does not recommend using them in production. These features provide early access to upcoming product features, enabling customers to test functionality and provide feedback during the development process.

For more information about the support scope of Red Hat Technology Preview features, see Technology Preview Features Support Scope .

Extensions enable cluster administrators to extend capabilities for users on their OpenShift Container Platform cluster.

Operator Lifecycle Manager (OLM) has been included with OpenShift Container Platform 4 since its initial release. OpenShift Container Platform 4.17 includes components for a next-generation iteration of OLM as a Generally Available (GA) feature, known during this phase as *OLM v1*. This updated framework evolves many of the concepts that have been part of previous versions of OLM and adds new capabilities.

1.1. HIGHLIGHTS

Administrators can explore the following highlights:

Fully declarative model that supports GitOps workflows

OLM v1 simplifies extension management through two key APIs:

- A new **ClusterExtension** API streamlines management of installed extensions, which includes Operators via the **registry+v1** bundle format, by consolidating user-facing APIs into a single object. This API is provided as **clusterextension.olm.operatorframework.io** by the new Operator Controller component. Administrators and SREs can use the API to automate processes and define desired states by using GitOps principles.

  NOTE

  Earlier Technology Preview phases of OLM v1 introduced a new **Operator** API; this API is renamed **ClusterExtension** in OpenShift Container Platform 4.16 to address the following improvements:

  - More accurately reflects the simplified functionality of extending a cluster's capabilities
  - Better represents a more flexible packaging format
  - **Cluster** prefix clearly indicates that **ClusterExtension** objects are cluster-scoped, a change from existing OLM where Operators could be either namespace-scoped or cluster-scoped

- The **Catalog** API, provided by the new catalogd component, serves as the foundation for OLM v1, unpacking catalogs for on-cluster clients so that users can discover installable content, such as Kubernetes extensions and Operators. This provides increased visibility into all available Operator bundle versions, including their details, channels, and update edges.

IMPORTANT

Currently, Operator Lifecycle Manager (OLM) v1 cannot authenticate private registries, such as the Red Hat-provided Operator catalogs. This is a known issue. As a result, the OLM v1 procedures that rely on having the Red Hat Operators catalog installed do not work. (**OCPBUGS-36364**)

For more information, see Operator Controller and Catalogd.

Improved control over extension updates

With improved insight into catalog content, administrators can specify target versions for installation and updates. This grants administrators more control over the target version of extension updates. For more information, see Updating an cluster extension.

Flexible extension packaging format

Administrators can use file-based catalogs to install and manage extensions, such as OLM-based Operators, similar to the existing OLM experience.
In addition, bundle size is no longer constrained by the etcd value size limit. For more information, see Installing extensions.

Secure catalog communication

OLM v1 uses HTTPS encryption for catalogd server responses.

1.2. PURPOSE

The mission of Operator Lifecycle Manager (OLM) has been to manage the lifecycle of cluster extensions centrally and declaratively on Kubernetes clusters. Its purpose has always been to make installing, running, and updating functional extensions to the cluster easy, safe, and reproducible for cluster and platform-as-a-service (PaaS) administrators throughout the lifecycle of the underlying cluster.

The initial version of OLM, which launched with OpenShift Container Platform 4 and is included by default, focused on providing unique support for these specific needs for a particular type of cluster extension, known as Operators. Operators are classified as one or more Kubernetes controllers, shipping with one or more API extensions, as **CustomResourceDefinition** (CRD) objects, to provide additional functionality to the cluster.

After running in production clusters for many releases, the next-generation of OLM aims to encompass lifecycles for cluster extensions that are not just Operators.

CHAPTER 2. ARCHITECTURE

2.1. OLM V1 COMPONENTS OVERVIEW

IMPORTANT

Operator Lifecycle Manager (OLM) v1 is a Technology Preview feature only. Technology Preview features are not supported with Red Hat production service level agreements (SLAs) and might not be functionally complete. Red Hat does not recommend using them in production. These features provide early access to upcoming product features, enabling customers to test functionality and provide feedback during the development process.

For more information about the support scope of Red Hat Technology Preview features, see Technology Preview Features Support Scope .

Operator Lifecycle Manager (OLM) v1 comprises the following component projects:

Operator Controller

Operator Controller is the central component of OLM v1 that extends Kubernetes with an API through which users can install and manage the lifecycle of Operators and extensions. It consumes information from catalogd.

Catalogd

Catalogd is a Kubernetes extension that unpacks file-based catalog (FBC) content packaged and shipped in container images for consumption by on-cluster clients. As a component of the OLM v1 microservices architecture, catalogd hosts metadata for Kubernetes extensions packaged by the authors of the extensions, and as a result helps users discover installable content.

2.2. OPERATOR CONTROLLER

IMPORTANT

Operator Lifecycle Manager (OLM) v1 is a Technology Preview feature only. Technology Preview features are not supported with Red Hat production service level agreements (SLAs) and might not be functionally complete. Red Hat does not recommend using them in production. These features provide early access to upcoming product features, enabling customers to test functionality and provide feedback during the development process.

For more information about the support scope of Red Hat Technology Preview features, see Technology Preview Features Support Scope .

Operator Controller is the central component of Operator Lifecycle Manager (OLM) v1 and consumes the other OLM v1 component, catalogd. It extends Kubernetes with an API through which users can install Operators and extensions.

2.2.1. ClusterExtension API

Operator Controller provides a new **ClusterExtension** API object that is a single resource representing an instance of an installed extension, which includes Operators via the **registry+v1** bundle format. This **clusterextension.olm.operatorframework.io** API streamlines management of installed extensions by consolidating user-facing APIs into a single object.

IMPORTANT

In OLM v1, **ClusterExtension** objects are cluster-scoped. This differs from existing OLM where Operators could be either namespace-scoped or cluster-scoped, depending on the configuration of their related **Subscription** and **OperatorGroup** objects.

For more information about the earlier behavior, see *Multitenancy and Operator colocation*.

Example ClusterExtension object

```
apiVersion: olm.operatorframework.io/v1alpha1
kind: ClusterExtension
metadata:
  name: <operator_name>
spec:
  packageName: <package_name>
  installNamespace: <namespace_name>
  channel: <channel_name>
  version: <version_number>
```

Additional resources

- Operator Lifecycle Manager (OLM) → Multitenancy and Operator colocation

2.2.1.1. Example custom resources (CRs) that specify a target version

In Operator Lifecycle Manager (OLM) v1, cluster administrators can declaratively set the target version of an Operator or extension in the custom resource (CR).

You can define a target version by specifying any of the following fields:

- Channel
- Version number
- Version range

If you specify a channel in the CR, OLM v1 installs the latest version of the Operator or extension that can be resolved within the specified channel. When updates are published to the specified channel, OLM v1 automatically updates to the latest release that can be resolved from the channel.

Example CR with a specified channel

```
apiVersion: olm.operatorframework.io/v1alpha1
kind: ClusterExtension
metadata:
  name: pipelines-operator
spec:
  packageName: openshift-pipelines-operator-rh
  installNamespace: <namespace_name>
  serviceAccount:
    name: <service_account>
  channel: latest 1
```

1 Installs the latest release that can be resolved from the specified channel. Updates to the channel are automatically installed.

If you specify the Operator or extension's target version in the CR, OLM v1 installs the specified version. When the target version is specified in the CR, OLM v1 does not change the target version when updates are published to the catalog.

If you want to update the version of the Operator that is installed on the cluster, you must manually edit the Operator's CR. Specifying an Operator's target version pins the Operator's version to the specified release.

Example CR with the target version specified

```
apiVersion: olm.operatorframework.io/v1alpha1
kind: ClusterExtension
metadata:
  name: pipelines-operator
spec:
  packageName: openshift-pipelines-operator-rh
  installNamespace: <namespace_name>
  serviceAccount:
    name: <service_account>
  version: "1.11.1" 1
```

1 Specifies the target version. If you want to update the version of the Operator or extension that is installed, you must manually update this field the CR to the desired target version.

If you want to define a range of acceptable versions for an Operator or extension, you can specify a version range by using a comparison string. When you specify a version range, OLM v1 installs the latest version of an Operator or extension that can be resolved by the Operator Controller.

Example CR with a version range specified

```
apiVersion: olm.operatorframework.io/v1alpha1
kind: ClusterExtension
metadata:
  name: pipelines-operator
spec:
  packageName: openshift-pipelines-operator-rh
  installNamespace: <namespace_name>
  serviceAccount:
    name: <service_account>
  version: ">1.11.1" 1
```

1 Specifies that the desired version range is greater than version **1.11.1**. For more information, see "Support for version ranges".

After you create or update a CR, apply the configuration file by running the following command:

Command syntax

```
$ oc apply -f <extension_name>.yaml
```

2.2.2. Object ownership for cluster extensions

In Operator Lifecycle Manager (OLM) v1, a Kubernetes object can only be owned by a single **ClusterExtension** object at a time. This ensures that objects within an OpenShift Container Platform cluster are managed consistently and prevents conflicts between multiple cluster extensions attempting to control the same object.

2.2.2.1. Single ownership

The core ownership principle enforced by OLM v1 is that each object can only have one cluster extension as its owner. This prevents overlapping or conflicting management by multiple cluster extensions, ensuring that each object is uniquely associated with only one bundle.

Implications of single ownership

- Bundles that provide a **CustomResourceDefinition** (CRD) object can only be installed once.
  Bundles provide CRDs, which are part of a **ClusterExtension** object. This means you can install a bundle only once in a cluster. Attempting to install another bundle that provides the same CRD results in failure, as each custom resource can have only one cluster extension as its owner.

- Cluster extensions cannot share objects.
  The single-owner policy of OLM v1 means that cluster extensions cannot share ownership of any objects. If one cluster extension manages a specific object, such as a **Deployment**, **CustomResourceDefinition**, or **Service** object, another cluster extension cannot claim ownership of the same object. Any attempt to do so is blocked by OLM v1.

2.2.2.2. Error messages

When a conflict occurs due to multiple cluster extensions attempting to manage the same object, Operator Controller returns an error message indicating the ownership conflict, such as the following:

Example error message

```
CustomResourceDefinition 'logfilemetricexporters.logging.kubernetes.io' already exists in namespace
'kubernetes-logging' and cannot be managed by operator-controller
```

This error message signals that the object is already being managed by another cluster extension and cannot be reassigned or shared.

2.2.2.3. Considerations

As a cluster or extension administrator, review the following considerations:

Uniqueness of bundles

Ensure that Operator bundles providing the same CRDs are not installed more than once. This can prevent potential installation failures due to ownership conflicts.

Avoid object sharing

If you need different cluster extensions to interact with similar resources, ensure they are managing separate objects. Cluster extensions cannot jointly manage the same object due to the single-owner enforcement.

2.3. CATALOGD

IMPORTANT

Operator Lifecycle Manager (OLM) v1 is a Technology Preview feature only. Technology Preview features are not supported with Red Hat production service level agreements (SLAs) and might not be functionally complete. Red Hat does not recommend using them in production. These features provide early access to upcoming product features, enabling customers to test functionality and provide feedback during the development process.

For more information about the support scope of Red Hat Technology Preview features, see Technology Preview Features Support Scope .

Operator Lifecycle Manager (OLM) v1 uses the catalogd component and its resources to manage Operator and extension catalogs.

IMPORTANT

Currently, Operator Lifecycle Manager (OLM) v1 cannot authenticate private registries, such as the Red Hat-provided Operator catalogs. This is a known issue. As a result, the OLM v1 procedures that rely on having the Red Hat Operators catalog installed do not work. (**OCPBUGS-36364**)

2.3.1. About catalogs in OLM v1

You can discover installable content by querying a catalog for Kubernetes extensions, such as Operators and controllers, by using the catalogd component. Catalogd is a Kubernetes extension that unpacks catalog content for on-cluster clients and is part of the Operator Lifecycle Manager (OLM) v1 suite of microservices. Currently, catalogd unpacks catalog content that is packaged and distributed as container images.

IMPORTANT

If you try to install an Operator or extension that does not have unique name, the installation might fail or lead to an unpredictable result. This occurs for the following reasons:

- If mulitple catalogs are installed on a cluster, Operator Lifecycle Manager (OLM) v1 does not include a mechanism to specify a catalog when you install an Operator or extension.
- OLM v1 requires that all of the Operators and extensions that are available to install on a cluster use a unique name for their bundles and packages.

Additional resources

- File-based catalogs
- Adding a catalog to a cluster

- Red Hat-provided catalogs

CHAPTER 3. OPERATOR FRAMEWORK GLOSSARY OF COMMON TERMS

IMPORTANT

Operator Lifecycle Manager (OLM) v1 is a Technology Preview feature only. Technology Preview features are not supported with Red Hat production service level agreements (SLAs) and might not be functionally complete. Red Hat does not recommend using them in production. These features provide early access to upcoming product features, enabling customers to test functionality and provide feedback during the development process.

For more information about the support scope of Red Hat Technology Preview features, see Technology Preview Features Support Scope .

The following terms are related to the Operator Framework, including Operator Lifecycle Manager (OLM) v1.

3.1. COMMON OPERATOR FRAMEWORK TERMS

3.1.1. Bundle

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

3.1.2. Bundle image

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

3.1.3. Catalog source

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

3.1.4. Channel

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

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

3.1.5. Channel head

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

3.1.6. Cluster service version

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

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

3.1.7. Dependency

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

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

3.1.8. Index image

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

3.1.9. Install plan

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

3.1.10. Multitenancy

A *tenant* in OpenShift Container Platform is a user or group of users that share common access and privileges for a set of deployed workloads, typically represented by a namespace or project. You can use tenants to provide a level of isolation between different groups or teams.

When a cluster is shared by multiple users or groups, it is considered a *multitenant* cluster.

3.1.11. Operator group

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

3.1.12. Package

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

3.1.13. Registry

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

3.1.14. Subscription

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

3.1.15. Update graph

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

CHAPTER 4. CATALOGS

4.1. FILE-BASED CATALOGS

IMPORTANT

Operator Lifecycle Manager (OLM) v1 is a Technology Preview feature only. Technology Preview features are not supported with Red Hat production service level agreements (SLAs) and might not be functionally complete. Red Hat does not recommend using them in production. These features provide early access to upcoming product features, enabling customers to test functionality and provide feedback during the development process.

For more information about the support scope of Red Hat Technology Preview features, see Technology Preview Features Support Scope .

Operator Lifecycle Manager (OLM) v1 in OpenShift Container Platform supports *file-based catalogs* for discovering and sourcing cluster extensions, including Operators, on a cluster.

IMPORTANT

Currently, Operator Lifecycle Manager (OLM) v1 cannot authenticate private registries, such as the Red Hat-provided Operator catalogs. This is a known issue. As a result, the OLM v1 procedures that rely on having the Red Hat Operators catalog installed do not work. (**OCPBUGS-36364**)

4.1.1. Highlights

File-based catalogs are the latest iteration of the catalog format in Operator Lifecycle Manager (OLM). It is a plain text-based (JSON or YAML) and declarative config evolution of the earlier SQLite database format, and it is fully backwards compatible. The goal of this format is to enable Operator catalog editing, composability, and extensibility.

Editing

With file-based catalogs, users interacting with the contents of a catalog are able to make direct changes to the format and verify that their changes are valid. Because this format is plain text JSON or YAML, catalog maintainers can easily manipulate catalog metadata by hand or with widely known and supported JSON or YAML tooling, such as the **jq** CLI.
This editability enables the following features and user-defined extensions:

- Promoting an existing bundle to a new channel
- Changing the default channel of a package
- Custom algorithms for adding, updating, and removing upgrade edges

Composability

File-based catalogs are stored in an arbitrary directory hierarchy, which enables catalog composition. For example, consider two separate file-based catalog directories: **catalogA** and **catalogB**. A catalog maintainer can create a new combined catalog by making a new directory **catalogC** and copying **catalogA** and **catalogB** into it.
This composability enables decentralized catalogs. The format permits Operator authors to maintain

Operator-specific catalogs, and it permits maintainers to trivially build a catalog composed of individual Operator catalogs. File-based catalogs can be composed by combining multiple other catalogs, by extracting subsets of one catalog, or a combination of both of these.

NOTE

Duplicate packages and duplicate bundles within a package are not permitted. The **opm validate** command returns an error if any duplicates are found.

Because Operator authors are most familiar with their Operator, its dependencies, and its upgrade compatibility, they are able to maintain their own Operator-specific catalog and have direct control over its contents. With file-based catalogs, Operator authors own the task of building and maintaining their packages in a catalog. Composite catalog maintainers, however, only own the task of curating the packages in their catalog and publishing the catalog to users.

Extensibility

The file-based catalog specification is a low-level representation of a catalog. While it can be maintained directly in its low-level form, catalog maintainers can build interesting extensions on top that can be used by their own custom tooling to make any number of mutations.
For example, a tool could translate a high-level API, such as **(mode=semver)**, down to the low-level, file-based catalog format for upgrade edges. Or a catalog maintainer might need to customize all of the bundle metadata by adding a new property to bundles that meet a certain criteria.

While this extensibility allows for additional official tooling to be developed on top of the low-level APIs for future OpenShift Container Platform releases, the major benefit is that catalog maintainers have this capability as well.

4.1.2. Directory structure

File-based catalogs can be stored and loaded from directory-based file systems. The **opm** CLI loads the catalog by walking the root directory and recursing into subdirectories. The CLI attempts to load every file it finds and fails if any errors occur.

Non-catalog files can be ignored using **.indexignore** files, which have the same rules for patterns and precedence as **.gitignore** files.

Example .indexignore file

```
# Ignore everything except non-object .json and .yaml files
**/*
!*.json
!*.yaml
**/objects/*.json
**/objects/*.yaml
```

Catalog maintainers have the flexibility to choose their desired layout, but it is recommended to store each package’s file-based catalog blobs in separate subdirectories. Each individual file can be either JSON or YAML; it is not necessary for every file in a catalog to use the same format.

Basic recommended structure

```
catalog
├── packageA
```

```
│   └── index.yaml
├── packageB
│   ├── .indexignore
│   ├── index.yaml
│   └── objects
│       └── packageB.v0.1.0.clusterserviceversion.yaml
└── packageC
    └── index.json
    └── deprecations.yaml
```

This recommended structure has the property that each subdirectory in the directory hierarchy is a self-contained catalog, which makes catalog composition, discovery, and navigation trivial file system operations. The catalog can also be included in a parent catalog by copying it into the parent catalog's root directory.

4.1.3. Schemas

File-based catalogs use a format, based on the CUE language specification, that can be extended with arbitrary schemas. The following **_Meta** CUE schema defines the format that all file-based catalog blobs must adhere to:

_Meta schema

```
_Meta: {
  // schema is required and must be a non-empty string
  schema: string & !=""

  // package is optional, but if it's defined, it must be a non-empty string
  package?: string & !=""

  // properties is optional, but if it's defined, it must be a list of 0 or more properties
  properties?: [... #Property]
}

#Property: {
  // type is required
  type: string & !=""

  // value is required, and it must not be null
  value: !=null
}
```

NOTE

No CUE schemas listed in this specification should be considered exhaustive. The **opm validate** command has additional validations that are difficult or impossible to express concisely in CUE.

An Operator Lifecycle Manager (OLM) catalog currently uses three schemas (**olm.package**, **olm.channel**, and **olm.bundle**), which correspond to OLM's existing package and bundle concepts.

Each Operator package in a catalog requires exactly one **olm.package** blob, at least one **olm.channel** blob, and one or more **olm.bundle** blobs.

NOTE

All **olm.*** schemas are reserved for OLM-defined schemas. Custom schemas must use a unique prefix, such as a domain that you own.

4.1.3.1. olm.package schema

The **olm.package** schema defines package-level metadata for an Operator. This includes its name, description, default channel, and icon.

Example 4.1. olm.package schema

```
#Package: {
  schema: "olm.package"

  // Package name
  name: string & !=""

  // A description of the package
  description?: string

  // The package's default channel
  defaultChannel: string & !=""

  // An optional icon
  icon?: {
    base64data: string
    mediatype:  string
  }
}
```

4.1.3.2. olm.channel schema

The **olm.channel** schema defines a channel within a package, the bundle entries that are members of the channel, and the upgrade edges for those bundles.

If a bundle entry represents an edge in multiple **olm.channel** blobs, it can only appear once per channel.

It is valid for an entry's **replaces** value to reference another bundle name that cannot be found in this catalog or another catalog. However, all other channel invariants must hold true, such as a channel not having multiple heads.

Example 4.2. olm.channel schema

```
#Channel: {
  schema: "olm.channel"
  package: string & !=""
  name: string & !=""
  entries: [...#ChannelEntry]
}

#ChannelEntry: {
  // name is required. It is the name of an `olm.bundle` that
  // is present in the channel.
  name: string & !=""

  // replaces is optional. It is the name of bundle that is replaced
  // by this entry. It does not have to be present in the entry list.
  replaces?: string & !=""

  // skips is optional. It is a list of bundle names that are skipped by
  // this entry. The skipped bundles do not have to be present in the
  // entry list.
  skips?: [...string & !=""]

  // skipRange is optional. It is the semver range of bundle versions
  // that are skipped by this entry.
  skipRange?: string & !=""
}
```

WARNING

When using the **skipRange** field, the skipped Operator versions are pruned from the update graph and are longer installable by users with the **spec.startingCSV** property of **Subscription** objects.

You can update an Operator incrementally while keeping previously installed versions available to users for future installation by using both the **skipRange** and **replaces** field. Ensure that the **replaces** field points to the immediate previous version of the Operator version in question.

4.1.3.3. olm.bundle schema

Example 4.3. olm.bundle schema

```
#Bundle: {
  schema: "olm.bundle"
  package: string & !=""
  name: string & !=""
  image: string & !=""
  properties: [...#Property]
  relatedImages?: [...#RelatedImage]
}

#Property: {
  // type is required
  type: string & !=""

  // value is required, and it must not be null
  value: !=null
}
```

```
#RelatedImage: {
  // image is the image reference
  image: string & !=""

  // name is an optional descriptive name for an image that
  // helps identify its purpose in the context of the bundle
  name?: string & !=""
}
```

4.1.3.4. olm.deprecations schema

The optional **olm.deprecations** schema defines deprecation information for packages, bundles, and channels in a catalog. Operator authors can use this schema to provide relevant messages about their Operators, such as support status and recommended upgrade paths, to users running those Operators from a catalog.

An **olm.deprecations** schema entry contains one or more of the following **reference** types, which indicates the deprecation scope. After the Operator is installed, any specified messages can be viewed as status conditions on the related **Subscription** object.

Table 4.1. Deprecation **reference** types

Type	Scope	Status condition
olm.package	Represents the entire package	**PackageDeprecated**
olm.channel	Represents one channel	**ChannelDeprecated**
olm.bundle	Represents one bundle version	**BundleDeprecated**

Each **reference** type has their own requirements, as detailed in the following example.

Example 4.4. Example olm.deprecations schema with eachreference type

```
schema: olm.deprecations
package: my-operator (1)
entries:
  - reference:
      schema: olm.package (2)
    message: | (3)
    The 'my-operator' package is end of life. Please use the
    'my-operator-new' package for support.
  - reference:
      schema: olm.channel
      name: alpha (4)
    message: |
    The 'alpha' channel is no longer supported. Please switch to the
    'stable' channel.
  - reference:
      schema: olm.bundle
      name: my-operator.v1.68.0 (5)
    message: |
    my-operator.v1.68.0 is deprecated. Uninstall my-operator.v1.68.0 and
    install my-operator.v1.72.0 for support.
```

1. Each deprecation schema must have a **package** value, and that package reference must be unique across the catalog. There must not be an associated **name** field.

2. The **olm.package** schema must not include a **name** field, because it is determined by the **package** field defined earlier in the schema.

3. All **message** fields, for any **reference** type, must be a non-zero length and represented as an opaque text blob.

4. The **name** field for the **olm.channel** schema is required.

5. The **name** field for the **olm.bundle** schema is required.

NOTE

The deprecation feature does not consider overlapping deprecation, for example package versus channel versus bundle.

Operator authors can save **olm.deprecations** schema entries as a **deprecations.yaml** file in the same directory as the package's **index.yaml** file:

Example directory structure for a catalog with deprecations

```
my-catalog
└── my-operator
    ├── index.yaml
    └── deprecations.yaml
```

Additional resources

- Updating or filtering a file-based catalog image

4.1.4. Properties

Properties are arbitrary pieces of metadata that can be attached to file-based catalog schemas. The **type** field is a string that effectively specifies the semantic and syntactic meaning of the **value** field. The value can be any arbitrary JSON or YAML.

OLM defines a handful of property types, again using the reserved **olm.*** prefix.

4.1.4.1. olm.package property

The **olm.package** property defines the package name and version. This is a required property on bundles, and there must be exactly one of these properties. The **packageName** field must match the bundle's first-class **package** field, and the **version** field must be a valid semantic version.

Example 4.5. olm.package property

```
#PropertyPackage: {
  type: "olm.package"
  value: {
    packageName: string & !=""
    version: string & !=""
  }
}
```

4.1.4.2. olm.gvk property

The **olm.gvk** property defines the group/version/kind (GVK) of a Kubernetes API that is provided by this bundle. This property is used by OLM to resolve a bundle with this property as a dependency for other bundles that list the same GVK as a required API. The GVK must adhere to Kubernetes GVK validations.

Example 4.6. **olm.gvk** property

```
#PropertyGVK: {
  type: "olm.gvk"
  value: {
    group: string & !=""
    version: string & !=""
    kind: string & !=""
  }
}
```

4.1.4.3. olm.package.required

The **olm.package.required** property defines the package name and version range of another package that this bundle requires. For every required package property a bundle lists, OLM ensures there is an Operator installed on the cluster for the listed package and in the required version range. The **versionRange** field must be a valid semantic version (semver) range.

Example 4.7. **olm.package.required** property

```
#PropertyPackageRequired: {
  type: "olm.package.required"
  value: {
    packageName: string & !=""
    versionRange: string & !=""
  }
}
```

4.1.4.4. olm.gvk.required

The **olm.gvk.required** property defines the group/version/kind (GVK) of a Kubernetes API that this bundle requires. For every required GVK property a bundle lists, OLM ensures there is an Operator installed on the cluster that provides it. The GVK must adhere to Kubernetes GVK validations.

Example 4.8. olm.gvk.required property

```
#PropertyGVKRequired: {
  type: "olm.gvk.required"
  value: {
    group: string & !=""
    version: string & !=""
    kind: string & !=""
  }
}
```

4.1.5. Example catalog

With file-based catalogs, catalog maintainers can focus on Operator curation and compatibility. Because Operator authors have already produced Operator-specific catalogs for their Operators, catalog maintainers can build their catalog by rendering each Operator catalog into a subdirectory of the catalog's root directory.

There are many possible ways to build a file-based catalog; the following steps outline a simple approach:

1. Maintain a single configuration file for the catalog, containing image references for each Operator in the catalog:

   Example catalog configuration file

   ```
   name: community-operators
   repo: quay.io/community-operators/catalog
   tag: latest
   references:
   - name: etcd-operator
     image: quay.io/etcd-operator/index@sha256:5891b5b522d5df086d0ff0b110fbd9d21bb4fc7163af34d08286a2e846f6be03
   - name: prometheus-operator
     image: quay.io/prometheus-operator/index@sha256:e258d248fda94c63753607f7c4494ee0fcbe92f1a76bfdac795c9d84101eb317
   ```

2. Run a script that parses the configuration file and creates a new catalog from its references:

   Example script

   ```
   name=$(yq eval '.name' catalog.yaml)
   mkdir "$name"
   yq eval '.name + "/" + .references[].name' catalog.yaml | xargs mkdir
   for l in $(yq e '.name as $catalog | .references[] | .image + "|" + $catalog + "/" + .name + "/index.yaml"' catalog.yaml); do
     image=$(echo $l | cut -d'|' -f1)
     file=$(echo $l | cut -d'|' -f2)
     opm render "$image" > "$file"
   done
   opm generate dockerfile "$name"
   ```

```
indexImage=$(yq eval '.repo + ":" + .tag' catalog.yaml)
docker build -t "$indexImage" -f "$name.Dockerfile" .
docker push "$indexImage"
```

4.1.6. Guidelines

Consider the following guidelines when maintaining file-based catalogs.

4.1.6.1. Immutable bundles

The general advice with Operator Lifecycle Manager (OLM) is that bundle images and their metadata should be treated as immutable.

If a broken bundle has been pushed to a catalog, you must assume that at least one of your users has upgraded to that bundle. Based on that assumption, you must release another bundle with an upgrade edge from the broken bundle to ensure users with the broken bundle installed receive an upgrade. OLM will not reinstall an installed bundle if the contents of that bundle are updated in the catalog.

However, there are some cases where a change in the catalog metadata is preferred:

- Channel promotion: If you already released a bundle and later decide that you would like to add it to another channel, you can add an entry for your bundle in another **olm.channel** blob.
- New upgrade edges: If you release a new **1.2.z** bundle version, for example **1.2.4**, but **1.3.0** is already released, you can update the catalog metadata for **1.3.0** to skip **1.2.4**.

4.1.6.2. Source control

Catalog metadata should be stored in source control and treated as the source of truth. Updates to catalog images should include the following steps:

1. Update the source-controlled catalog directory with a new commit.
2. Build and push the catalog image. Use a consistent tagging taxonomy, such as **:latest** or **:<target_cluster_version>**, so that users can receive updates to a catalog as they become available.

4.1.7. CLI usage

For instructions about creating file-based catalogs by using the **opm** CLI, see Managing custom catalogs.

For reference documentation about the **opm** CLI commands related to managing file-based catalogs, see CLI tools.

4.1.8. Automation

Operator authors and catalog maintainers are encouraged to automate their catalog maintenance with CI/CD workflows. Catalog maintainers can further improve on this by building GitOps automation to accomplish the following tasks:

- Check that pull request (PR) authors are permitted to make the requested changes, for example by updating their package’s image reference.
- Check that the catalog updates pass the **opm validate** command.

- Check that the updated bundle or catalog image references exist, the catalog images run successfully in a cluster, and Operators from that package can be successfully installed.
- Automatically merge PRs that pass the previous checks.
- Automatically rebuild and republish the catalog image.

4.2. RED HAT-PROVIDED CATALOGS

IMPORTANT

Operator Lifecycle Manager (OLM) v1 is a Technology Preview feature only. Technology Preview features are not supported with Red Hat production service level agreements (SLAs) and might not be functionally complete. Red Hat does not recommend using them in production. These features provide early access to upcoming product features, enabling customers to test functionality and provide feedback during the development process.

For more information about the support scope of Red Hat Technology Preview features, see Technology Preview Features Support Scope .

Red Hat provides several Operator catalogs that are included with OpenShift Container Platform by default.

IMPORTANT

Currently, Operator Lifecycle Manager (OLM) v1 cannot authenticate private registries, such as the Red Hat-provided Operator catalogs. This is a known issue. As a result, the OLM v1 procedures that rely on having the Red Hat Operators catalog installed do not work. (**OCPBUGS-36364**)

4.2.1. About Red Hat-provided Operator catalogs

The Red Hat-provided catalog sources are installed by default in the **openshift-marketplace** namespace, which makes the catalogs available cluster-wide in all namespaces.

The following Operator catalogs are distributed by Red Hat:

Catalog	Index image	Description
redhat-operators	**registry.redhat.io/redhat/redhat-operator-index:v4.17**	Red Hat products packaged and shipped by Red Hat. Supported by Red Hat.
certified-operators	**registry.redhat.io/redhat/certified-operator-index:v4.17**	Products from leading independent software vendors (ISVs). Red Hat partners with ISVs to package and ship. Supported by the ISV.

Catalog	Index image	Description
redhat-marketplace	**registry.redhat.io/redhat/redhat-marketplace-index:v4.17**	Certified software that can be purchased from Red Hat Marketplace.
community-operators	**registry.redhat.io/redhat/community-operator-index:v4.17**	Software maintained by relevant representatives in the redhat-openshift-ecosystem/community-operators-prod/operators GitHub repository. No official support.

During a cluster upgrade, the index image tag for the default Red Hat-provided catalog sources are updated automatically by the Cluster Version Operator (CVO) so that Operator Lifecycle Manager (OLM) pulls the updated version of the catalog. For example during an upgrade from OpenShift Container Platform 4.8 to 4.9, the **spec.image** field in the **CatalogSource** object for the **redhat-operators** catalog is updated from:

```
registry.redhat.io/redhat/redhat-operator-index:v4.8
```

to:

```
registry.redhat.io/redhat/redhat-operator-index:v4.9
```

4.3. MANAGING CATALOGS

IMPORTANT

Operator Lifecycle Manager (OLM) v1 is a Technology Preview feature only. Technology Preview features are not supported with Red Hat production service level agreements (SLAs) and might not be functionally complete. Red Hat does not recommend using them in production. These features provide early access to upcoming product features, enabling customers to test functionality and provide feedback during the development process.

For more information about the support scope of Red Hat Technology Preview features, see Technology Preview Features Support Scope .

Cluster administrators can add *catalogs*, or curated collections of Operators and Kubernetes extensions, to their clusters. Operator authors publish their products to these catalogs. When you add a catalog to your cluster, you have access to the versions, patches, and over-the-air updates of the Operators and extensions that are published to the catalog.

You can manage catalogs and extensions declaratively from the CLI by using custom resources (CRs).

File-based catalogs are the latest iteration of the catalog format in Operator Lifecycle Manager (OLM). It is a plain text-based (JSON or YAML) and declarative config evolution of the earlier SQLite database format, and it is fully backwards compatible.

IMPORTANT

Kubernetes periodically deprecates certain APIs that are removed in subsequent releases. As a result, Operators are unable to use removed APIs starting with the version of OpenShift Container Platform that uses the Kubernetes version that removed the API.

If your cluster is using custom catalogs, see Controlling Operator compatibility with OpenShift Container Platform versions for more details about how Operator authors can update their projects to help avoid workload issues and prevent incompatible upgrades.

4.3.1. About catalogs in OLM v1

You can discover installable content by querying a catalog for Kubernetes extensions, such as Operators and controllers, by using the catalogd component. Catalogd is a Kubernetes extension that unpacks catalog content for on-cluster clients and is part of the Operator Lifecycle Manager (OLM) v1 suite of microservices. Currently, catalogd unpacks catalog content that is packaged and distributed as container images.

IMPORTANT

If you try to install an Operator or extension that does not have unique name, the installation might fail or lead to an unpredictable result. This occurs for the following reasons:

- If mulitple catalogs are installed on a cluster, Operator Lifecycle Manager (OLM) v1 does not include a mechanism to specify a catalog when you install an Operator or extension.
- OLM v1 requires that all of the Operators and extensions that are available to install on a cluster use a unique name for their bundles and packages.

Additional resources

- File-based catalogs

4.3.2. Red Hat-provided Operator catalogs in OLM v1

Operator Lifecycle Manager (OLM) v1 does not include Red Hat-provided Operator catalogs by default. If you want to add a Red Hat-provided catalog to your cluster, create a custom resource (CR) for the catalog and apply it to the cluster. The following custom resource (CR) examples show how to create a catalog resources for OLM v1.

IMPORTANT

- Currently, Operator Lifecycle Manager (OLM) v1 cannot authenticate private registries, such as the Red Hat-provided Operator catalogs. This is a known issue. As a result, the OLM v1 procedures that rely on having the Red Hat Operators catalog installed do not work. (OCPBUGS-36364)

- If you want to use a catalog that is hosted on a private registry, such as Red Hat-provided Operator catalogs from **registry.redhat.io**, you must have a pull secret scoped to the **openshift-catalogd** namespace.

  For more information, see "Creating a pull secret for catalogs hosted on a secure registry".

Example Red Hat Operators catalog

```
apiVersion: catalogd.operatorframework.io/v1alpha1
kind: ClusterCatalog
metadata:
  name: redhat-operators
spec:
  source:
    type: image
    image:
      ref: registry.redhat.io/redhat/redhat-operator-index:v4.17
      pullSecret: <pull_secret_name>
      pollInterval: <poll_interval_duration> (1)
```

1 Specify the interval for polling the remote registry for newer image digests. The default value is **24h**. Valid units include seconds (**s**), minutes (**m**), and hours (**h**). To disable polling, set a zero value, such as **0s**.

Example Certified Operators catalog

```
apiVersion: catalogd.operatorframework.io/v1alpha1
kind: ClusterCatalog
metadata:
  name: certified-operators
spec:
  source:
    type: image
    image:
      ref: registry.redhat.io/redhat/certified-operator-index:v4.17
      pullSecret: <pull_secret_name>
      pollInterval: 24h
```

Example Community Operators catalog

```
apiVersion: catalogd.operatorframework.io/v1alpha1
kind: ClusterCatalog
metadata:
  name: community-operators
spec:
```

```
source:
  type: image
  image:
    ref: registry.redhat.io/redhat/community-operator-index:v4.17
    pullSecret: <pull_secret_name>
    pollInterval: 24h
```

The following command adds a catalog to your cluster:

Command syntax

```
$ oc apply -f <catalog_name>.yaml 1
```

1 Specifies the catalog CR, such as **redhat-operators.yaml**.

4.3.3. Creating a pull secret for catalogs hosted on a private registry

If you want to use a catalog that is hosted on a private registry, such as Red Hat-provided Operator catalogs from **registry.redhat.io**, you must have a pull secret scoped to the **openshift-catalogd** namespace.

Catalogd cannot read global pull secrets from OpenShift Container Platform clusters. Catalogd can read references to secrets only in the namespace where it is deployed.

IMPORTANT

Currently, Operator Lifecycle Manager (OLM) v1 cannot authenticate private registries, such as the Red Hat-provided Operator catalogs. This is a known issue. As a result, the OLM v1 procedures that rely on having the Red Hat Operators catalog installed do not work. (OCPBUGS-36364)

Prerequisites

- Login credentials for the secure registry
- Docker or Podman installed on your workstation

Procedure

- If you already have a **.dockercfg** file with login credentials for the secure registry, create a pull secret by running the following command:

  ```
  $ oc create secret generic <pull_secret_name> \
      --from-file=.dockercfg=<file_path>/.dockercfg \
      --type=kubernetes.io/dockercfg \
      --namespace=openshift-catalogd
  ```

  Example 4.9. Example command

  ```
  $ oc create secret generic redhat-cred \
      --from-file=.dockercfg=/home/<username>/.dockercfg \
      --type=kubernetes.io/dockercfg \
  ```

```
    --namespace=openshift-catalogd
```

- If you already have a **$HOME/.docker/config.json** file with login credentials for the secured registry, create a pull secret by running the following command:

```
$ oc create secret generic <pull_secret_name> \
    --from-file=.dockerconfigjson=<file_path>/.docker/config.json \
    --type=kubernetes.io/dockerconfigjson \
    --namespace=openshift-catalogd
```

Example 4.10. Example command

```
$ oc create secret generic redhat-cred \
    --from-file=.dockerconfigjson=/home/<username>/.docker/config.json \
    --type=kubernetes.io/dockerconfigjson \
    --namespace=openshift-catalogd
```

- If you do not have a Docker configuration file with login credentials for the secure registry, create a pull secret by running the following command:

```
$ oc create secret docker-registry <pull_secret_name> \
    --docker-server=<registry_server> \
    --docker-username=<username> \
    --docker-password=<password> \
    --docker-email=<email> \
    --namespace=openshift-catalogd
```

Example 4.11. Example command

```
$ oc create secret docker-registry redhat-cred \
    --docker-server=registry.redhat.io \
    --docker-username=username \
    --docker-password=password \
    --docker-email=user@example.com \
    --namespace=openshift-catalogd
```

4.3.4. Adding a catalog to a cluster

To add a catalog to a cluster, create a catalog custom resource (CR) and apply it to the cluster.

IMPORTANT

Currently, Operator Lifecycle Manager (OLM) v1 cannot authenticate private registries, such as the Red Hat-provided Operator catalogs. This is a known issue. As a result, the OLM v1 procedures that rely on having the Red Hat Operators catalog installed do not work. (OCPBUGS-36364)

Prerequisites

- If you want to use a catalog that is hosted on a private registry, such as Red Hat-provided Operator catalogs from **registry.redhat.io**, you must have a pull secret scoped to the **openshift-catalogd** namespace.

  Catalogd cannot read global pull secrets from OpenShift Container Platform clusters. Catalogd can read references to secrets only in the namespace where it is deployed.

Procedure

1. Create a catalog custom resource (CR), similar to the following example:

   #### Example **redhat-operators.yaml**

   ```
   apiVersion: catalogd.operatorframework.io/v1alpha1
   kind: ClusterCatalog
   metadata:
     name: redhat-operators
   spec:
     source:
       type: image
       image:
         ref: registry.redhat.io/redhat/redhat-operator-index:v4.17 1
         pullSecret: <pull_secret_name> 2
         pollInterval: <poll_interval_duration> 3
   ```

   1 Specify the catalog's image in the **spec.source.image** field.

   2 If your catalog is hosted on a secure registry, such as **registry.redhat.io**, you must create a pull secret scoped to the **openshift-catalog** namespace.

   3 Specify the interval for polling the remote registry for newer image digests. The default value is **24h**. Valid units include seconds (**s**), minutes (**m**), and hours (**h**). To disable polling, set a zero value, such as **0s**.

2. Add the catalog to your cluster by running the following command:

   ```
   $ oc apply -f redhat-operators.yaml
   ```

   #### Example output

   ```
   catalog.catalogd.operatorframework.io/redhat-operators created
   ```

Verification

- Run the following commands to verify the status of your catalog:

  a. Check if you catalog is available by running the following command:

     ```
     $ oc get clustercatalog
     ```

     #### Example output

```
NAME               AGE
redhat-operators   20s
```

b. Check the status of your catalog by running the following command:

```
$ oc describe clustercatalog
```

Example output

```
Name:         redhat-operators
Namespace:
Labels:       <none>
Annotations:  <none>
API Version:  catalogd.operatorframework.io/v1alpha1
Kind:         ClusterCatalog
Metadata:
  Creation Timestamp:  2024-06-10T17:34:53Z
  Finalizers:
    catalogd.operatorframework.io/delete-server-cache
  Generation:        1
  Resource Version:  46075
  UID:               83c0db3c-a553-41da-b279-9b3cddaa117d
Spec:
  Source:
    Image:
      Pull Secret:  redhat-cred
      Ref:          registry.redhat.io/redhat/redhat-operator-index:v4.17
    Type:           image
Status: (1)
  Conditions:
    Last Transition Time:  2024-06-10T17:35:15Z
    Message:
    Reason:                UnpackSuccessful (2)
    Status:                True
    Type:                  Unpacked
  Content URL:             https://catalogd-catalogserver.openshift-
catalogd.svc/catalogs/redhat-operators/all.json
  Observed Generation:     1
  Phase:                   Unpacked (3)
  Resolved Source:
    Image:
      Last Poll Attempt:  2024-06-10T17:35:10Z
      Ref:                registry.redhat.io/redhat/redhat-operator-index:v4.17
      Resolved Ref:       registry.redhat.io/redhat/redhat-operator-
index@sha256:f2ccc079b5e490a50db532d1dc38fd659322594dcf3e653d650ead0e862029
d9 (4)
    Type:                 image
Events:                   <none>
```

(1) Describes the status of the catalog.

(2) Displays the reason the catalog is in the current state.

(3) Displays the phase of the installation process.

4 Displays the image reference of the catalog.

4.3.5. Deleting a catalog

You can delete a catalog by deleting its custom resource (CR).

Prerequisites

- You have a catalog installed.

Procedure

- Delete a catalog by running the following command:

  ```
  $ oc delete clustercatalog <catalog_name>
  ```

  Example output

  ```
  catalog.catalogd.operatorframework.io "my-catalog" deleted
  ```

Verification

- Verify the catalog is deleted by running the following command:

  ```
  $ oc get clustercatalog
  ```

4.4. CREATING CATALOGS

IMPORTANT

Operator Lifecycle Manager (OLM) v1 is a Technology Preview feature only. Technology Preview features are not supported with Red Hat production service level agreements (SLAs) and might not be functionally complete. Red Hat does not recommend using them in production. These features provide early access to upcoming product features, enabling customers to test functionality and provide feedback during the development process.

For more information about the support scope of Red Hat Technology Preview features, see Technology Preview Features Support Scope .

Catalog maintainers can create new catalogs in the file-based catalog format for use with Operator Lifecycle Manager (OLM) v1 on OpenShift Container Platform.

IMPORTANT

Currently, Operator Lifecycle Manager (OLM) v1 cannot authenticate private registries, such as the Red Hat-provided Operator catalogs. This is a known issue. As a result, the OLM v1 procedures that rely on having the Red Hat Operators catalog installed do not work. (**OCPBUGS-36364**)

4.4.1. Creating a file-based catalog image

You can use the **opm** CLI to create a catalog image that uses the plain text *file-based catalog* format (JSON or YAML), which replaces the deprecated SQLite database format.

Prerequisites

- You have installed the **opm** CLI.
- You have **podman** version 1.9.3+.
- A bundle image is built and pushed to a registry that supports Docker v2-2.

Procedure

1. Initialize the catalog:

   a. Create a directory for the catalog by running the following command:

   ```
   $ mkdir <catalog_dir>
   ```

   b. Generate a Dockerfile that can build a catalog image by running the **opm generate dockerfile** command:

   ```
   $ opm generate dockerfile <catalog_dir> \
       -i registry.redhat.io/openshift4/ose-operator-registry-rhel9:v4.17 (1)
   ```

   (1) Specify the official Red Hat base image by using the **-i** flag, otherwise the Dockerfile uses the default upstream image.

   The Dockerfile must be in the same parent directory as the catalog directory that you created in the previous step:

   Example directory structure

   ```
   . (1)
   ├── <catalog_dir> (2)
   └── <catalog_dir>.Dockerfile (3)
   ```

   (1) Parent directory

   (2) Catalog directory

   (3) Dockerfile generated by the **opm generate dockerfile** command

   c. Populate the catalog with the package definition for your Operator by running the **opm init** command:

   ```
   $ opm init <operator_name> \ (1)
       --default-channel=preview \ (2)
       --description=./README.md \ (3)
   ```

```
    --icon=./operator-icon.svg \ 4
    --output yaml \ 5
    > <catalog_dir>/index.yaml 6
```

1 Operator, or package, name

2 Channel that subscriptions default to if unspecified

3 Path to the Operator's **README.md** or other documentation

4 Path to the Operator's icon

5 Output format: JSON or YAML

6 Path for creating the catalog configuration file

This command generates an **olm.package** declarative config blob in the specified catalog configuration file.

2. Add a bundle to the catalog by running the **opm render** command:

```
$ opm render <registry>/<namespace>/<bundle_image_name>:<tag> \ 1
    --output=yaml \
    >> <catalog_dir>/index.yaml 2
```

1 Pull spec for the bundle image

2 Path to the catalog configuration file

NOTE

Channels must contain at least one bundle.

3. Add a channel entry for the bundle. For example, modify the following example to your specifications, and add it to your **<catalog_dir>/index.yaml** file:

Example channel entry

```
---
schema: olm.channel
package: <operator_name>
name: preview
entries:
  - name: <operator_name>.v0.1.0 1
```

1 Ensure that you include the period (**.**) after **<operator_name>** but before the **v** in the version. Otherwise, the entry fails to pass the **opm validate** command.

4. Validate the file-based catalog:

   a. Run the **opm validate** command against the catalog directory:

```
$ opm validate <catalog_dir>
```

b. Check that the error code is **0**:

```
$ echo $?
```

Example output

```
0
```

5. Build the catalog image by running the **podman build** command:

```
$ podman build . \
    -f <catalog_dir>.Dockerfile \
    -t <registry>/<namespace>/<catalog_image_name>:<tag>
```

6. Push the catalog image to a registry:

   a. If required, authenticate with your target registry by running the **podman login** command:

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

   b. Push the catalog image by running the **podman push** command:

```
$ podman push <registry>/<namespace>/<catalog_image_name>:<tag>
```

Additional resources

- **opm** CLI reference

4.4.2. Updating or filtering a file-based catalog image

You can use the **opm** CLI to update or filter a catalog image that uses the file-based catalog format. By extracting the contents of an existing catalog image, you can modify the catalog as needed, for example:

- Adding packages
- Removing packages
- Updating existing package entries
- Detailing deprecation messages per package, channel, and bundle

You can then rebuild the image as an updated version of the catalog.

> **NOTE**
>
> Alternatively, if you already have a catalog image on a mirror registry, you can use the oc-mirror CLI plugin to automatically prune any removed images from an updated source version of that catalog image while mirroring it to the target registry.
>
> For more information about the oc-mirror plugin and this use case, see the "Keeping your mirror registry content updated" section, and specifically the "Pruning images" subsection, of "Mirroring images for a disconnected installation using the oc-mirror plugin".

Prerequisites

- You have the following on your workstation:
  - The **opm** CLI.
  - **podman** version 1.9.3+.
  - A file-based catalog image.
  - A catalog directory structure recently initialized on your workstation related to this catalog. If you do not have an initialized catalog directory, create the directory and generate the Dockerfile. For more information, see the "Initialize the catalog" step from the "Creating a file-based catalog image" procedure.

Procedure

1. Extract the contents of the catalog image in YAML format to an **index.yaml** file in your catalog directory:

   ```
   $ opm render <registry>/<namespace>/<catalog_image_name>:<tag> \
       -o yaml > <catalog_dir>/index.yaml
   ```

   > **NOTE**
   >
   > Alternatively, you can use the **-o json** flag to output in JSON format.

2. Modify the contents of the resulting **index.yaml** file to your specifications:

   > **IMPORTANT**
   >
   > After a bundle has been published in a catalog, assume that one of your users has installed it. Ensure that all previously published bundles in a catalog have an update path to the current or newer channel head to avoid stranding users that have that version installed.

   - To add an Operator, follow the steps for creating package, bundle, and channel entries in the "Creating a file-based catalog image" procedure.
   - To remove an Operator, delete the set of **olm.package**, **olm.channel**, and **olm.bundle** blobs that relate to the package. The following example shows a set that must be deleted to remove the **example-operator** package from the catalog:

     Example 4.12. Example removed entries

```
---
defaultChannel: release-2.7
icon:
  base64data: <base64_string>
  mediatype: image/svg+xml
name: example-operator
schema: olm.package
---
entries:
- name: example-operator.v2.7.0
  skipRange: '>=2.6.0 <2.7.0'
- name: example-operator.v2.7.1
  replaces: example-operator.v2.7.0
  skipRange: '>=2.6.0 <2.7.1'
- name: example-operator.v2.7.2
  replaces: example-operator.v2.7.1
  skipRange: '>=2.6.0 <2.7.2'
- name: example-operator.v2.7.3
  replaces: example-operator.v2.7.2
  skipRange: '>=2.6.0 <2.7.3'
- name: example-operator.v2.7.4
  replaces: example-operator.v2.7.3
  skipRange: '>=2.6.0 <2.7.4'
name: release-2.7
package: example-operator
schema: olm.channel
---
image: example.com/example-inc/example-operator-bundle@sha256:<digest>
name: example-operator.v2.7.0
package: example-operator
properties:
- type: olm.gvk
  value:
    group: example-group.example.io
    kind: MyObject
    version: v1alpha1
- type: olm.gvk
  value:
    group: example-group.example.io
    kind: MyOtherObject
    version: v1beta1
- type: olm.package
  value:
    packageName: example-operator
    version: 2.7.0
- type: olm.bundle.object
  value:
    data: <base64_string>
- type: olm.bundle.object
  value:
    data: <base64_string>
relatedImages:
- image: example.com/example-inc/example-related-image@sha256:<digest>
  name: example-related-image
schema: olm.bundle
---
```

- To add or update deprecation messages for an Operator, ensure there is a **deprecations.yaml** file in the same directory as the package's **index.yaml** file. For information on the **deprecations.yaml** file format, see "olm.deprecations schema".

3. Save your changes.

4. Validate the catalog:

```
$ opm validate <catalog_dir>
```

5. Rebuild the catalog:

```
$ podman build . \
    -f <catalog_dir>.Dockerfile \
    -t <registry>/<namespace>/<catalog_image_name>:<tag>
```

6. Push the updated catalog image to a registry:

```
$ podman push <registry>/<namespace>/<catalog_image_name>:<tag>
```

Verification

1. In the web console, navigate to the OperatorHub configuration resource in the **Administration → Cluster Settings → Configuration** page.

2. Add the catalog source or update the existing catalog source to use the pull spec for your updated catalog image.
For more information, see "Adding a catalog source to a cluster" in the "Additional resources" of this section.

3. After the catalog source is in a **READY** state, navigate to the **Operators → OperatorHub** page and check that the changes you made are reflected in the list of Operators.

Additional resources

- Packaging format → Schemas → olm.deprecations schema

- Mirroring images for a disconnected installation using the oc-mirror plugin → Keeping your mirror registry content updated

- Adding a catalog source to a cluster

CHAPTER 5. CLUSTER EXTENSIONS

5.1. MANAGING CLUSTER EXTENSIONS

IMPORTANT

Operator Lifecycle Manager (OLM) v1 is a Technology Preview feature only. Technology Preview features are not supported with Red Hat production service level agreements (SLAs) and might not be functionally complete. Red Hat does not recommend using them in production. These features provide early access to upcoming product features, enabling customers to test functionality and provide feedback during the development process.

For more information about the support scope of Red Hat Technology Preview features, see Technology Preview Features Support Scope .

After a catalog has been added to your cluster, you have access to the versions, patches, and over-the-air updates of the extensions and Operators that are published to the catalog.

You can manage extensions declaratively from the CLI using custom resources (CRs).

IMPORTANT

Currently, Operator Lifecycle Manager (OLM) v1 cannot authenticate private registries, such as the Red Hat-provided Operator catalogs. This is a known issue. As a result, the OLM v1 procedures that rely on having the Red Hat Operators catalog installed do not work. (**OCPBUGS-36364**)

5.1.1. Supported extensions

Currently, Operator Lifecycle Manager (OLM) v1 supports installing cluster extensions that meet all of the following criteria:

- The extension must use the **registry+v1** bundle format introduced in existing OLM.
- The extension must support installation via the **AllNamespaces** install mode.
- The extension must not use webhooks.
- The extension must not declare dependencies by using any of the following file-based catalog properties:
  - **olm.gvk.required**
  - **olm.package.required**
  - **olm.constraint**

OLM v1 checks that the extension you want to install meets these constraints. If the extension that you want to install does not meet these constraints, an error message is printed in the cluster extension’s conditions.

IMPORTANT

Operator Lifecycle Manager (OLM) v1 does not support the **OperatorConditions** API introduced in existing OLM.

If an extension relies on only the **OperatorConditions** API to manage updates, the extension might not install correctly. Most extensions that rely on this API fail at start time, but some might fail during reconciliation.

As a workaround, you can pin your extension to a specific version. When you want to update your extension, consult the extension's documentation to find out when it is safe to pin the extension to a new version.

Additional resources

- Operator conditions

5.1.2. Finding Operators to install from a catalog

After you add a catalog to your cluster, you can query the catalog to find Operators and extensions to install. Before you can query catalogs, you must port forward the catalog server service.

Prerequisites

- You have added a catalog to your cluster.
- You have installed the **jq** CLI tool.

Procedure

1. Port forward the catalog server service in the **openshift-catalogd** namespace by running the following command:

```
$ oc -n openshift-catalogd port-forward svc/catalogd-catalogserver 8080:443
```

2. In a new terminal window or tab, download the catalog's JSON file locally by running the following command:

```
$ curl -L -k https://localhost:8080/catalogs/<catalog_name>/all.json \
  -C - -o /<path>/<catalog_name>.json
```

Example 5.1. Example command

```
$ curl -L -k https://localhost:8080/catalogs/redhat-operators/all.json \
  -C - -o /home/username/catalogs/rhoc.json
```

3. Run one of the following commands to return a list of Operators and extensions in a catalog.

IMPORTANT

Currently, Operator Lifecycle Manager (OLM) v1 supports installing cluster extensions that meet all of the following criteria:

- The extension must use the **registry+v1** bundle format introduced in existing OLM.
- The extension must support installation via the **AllNamespaces** install mode.
- The extension must not use webhooks.
- The extension must not declare dependencies by using any of the following file-based catalog properties:
  - **olm.gvk.required**
  - **olm.package.required**
  - **olm.constraint**

OLM v1 checks that the extension you want to install meets these constraints. If the extension that you want to install does not meet these constraints, an error message is printed in the cluster extension's conditions.

- Get a list of all the Operators and extensions from the local catalog file by running the following command:

```
$ jq -s '.[] | select(.schema == "olm.package") | .name' \
  /<path>/<filename>.json
```

Example 5.2. Example command

```
$ jq -s '.[] | select(.schema == "olm.package") | .name' \
  /home/username/catalogs/rhoc.json
```

Example 5.3. Example output

```
NAME                                          AGE
"3scale-operator"
"advanced-cluster-management"
"amq-broker-rhel8"
"amq-online"
"amq-streams"
"amq7-interconnect-operator"
"ansible-automation-platform-operator"
"ansible-cloud-addons-operator"
"apicast-operator"
"aws-efs-csi-driver-operator"
"aws-load-balancer-operator"
"bamoe-businessautomation-operator"
"bamoe-kogito-operator"
```

```
"bare-metal-event-relay"
"businessautomation-operator"
...
```

- Get list of packages that support **AllNamespaces** install mode and do not use webhooks from the local catalog file by running the following command:

```
$ jq -c 'select(.schema == "olm.bundle") | \
  {"package":.package, "version":.properties[] | \
  select(.type == "olm.bundle.object").value.data | @base64d | fromjson | \
  select(.kind == "ClusterServiceVersion" and (.spec.installModes[] | \
  select(.type == "AllNamespaces" and .supported == true) != null) \
  and .spec.webhookdefinitions == null).spec.version}' \
  /<path>/<catalog_name>.json
```

Example 5.4. Example output

```
{"package":"3scale-operator","version":"0.10.0-mas"}
{"package":"3scale-operator","version":"0.10.5"}
{"package":"3scale-operator","version":"0.11.0-mas"}
{"package":"3scale-operator","version":"0.11.1-mas"}
{"package":"3scale-operator","version":"0.11.2-mas"}
{"package":"3scale-operator","version":"0.11.3-mas"}
{"package":"3scale-operator","version":"0.11.5-mas"}
{"package":"3scale-operator","version":"0.11.6-mas"}
{"package":"3scale-operator","version":"0.11.7-mas"}
{"package":"3scale-operator","version":"0.11.8-mas"}
{"package":"amq-broker-rhel8","version":"7.10.0-opr-1"}
{"package":"amq-broker-rhel8","version":"7.10.0-opr-2"}
{"package":"amq-broker-rhel8","version":"7.10.0-opr-3"}
{"package":"amq-broker-rhel8","version":"7.10.0-opr-4"}
{"package":"amq-broker-rhel8","version":"7.10.1-opr-1"}
{"package":"amq-broker-rhel8","version":"7.10.1-opr-2"}
{"package":"amq-broker-rhel8","version":"7.10.2-opr-1"}
{"package":"amq-broker-rhel8","version":"7.10.2-opr-2"}
...
```

4. Inspect the contents of an Operator or extension’s metadata by running the following command:

```
$ jq -s '.[] | select( .schema == "olm.package") | \
  select( .name == "<package_name>")' /<path>/<catalog_name>.json
```

Example 5.5. Example command

```
$ jq -s '.[] | select( .schema == "olm.package") | \
  select( .name == "openshift-pipelines-operator-rh")' \
  /home/username/rhoc.json
```

Example 5.6. Example output

```
{
  "defaultChannel": "stable",
  "icon": {
    "base64data": "PHN2ZyB4bWxu..."
    "mediatype": "image/png"
  },
  "name": "openshift-pipelines-operator-rh",
  "schema": "olm.package"
}
```

5.1.2.1. Common catalog queries

You can query catalogs by using the **jq** CLI tool.

Table 5.1. Common package queries

Query	Request

Available packages in a catalog	`$ jq -s '.[] | select( .schema == "olm.package") | \`
	`  .name' <catalog_name>.json`

Packages that support **AllNamespaces** install mode and do not use webhooks	`$ jq -c 'select(.schema == "olm.bundle") | \`
	`  {"package":.package, "version":.properties[] | \`
	`  select(.type == "olm.bundle.object").value.data | \`
	`  @base64d | fromjson | \`
	`  select(.kind == "ClusterServiceVersion" and (.spec.installModes[] | \`
	`  select(.type == "AllNamespaces" and .supported == true) != null) \`
	`  and .spec.webhookdefinitions == null).spec.version}' \`
	`  <catalog_name>.json`

Package metadata	`$ jq -s '.[] | select( .schema == "olm.package") | \`
	`  select( .name == "<package_name>")' <catalog_name>.json`

Catalog blobs in a package	`$ jq -s '.[] | select( .package == "<package_name>")' \`
	`  <catalog_name>.json`

Table 5.2. Common channel queries

Query	Request

Channels in a package	`$ jq -s '.[] | select( .schema == "olm.channel" ) | \`
	`  select( .package == "<package_name>") | .name' \`
	`  <catalog_name>.json`

Query	Request
Versions in a channel	`$ jq -s '.[] | select( .package == "<package_name>" ) | \` `select( .schema == "olm.channel" ) | \` `select( .name == "<channel_name>" ) | \` `.entries | .[] | .name' <catalog_name>.json`
• Latest version in a channel • Upgrade path	`$ jq -s '.[] | select( .schema == "olm.channel" ) | \` `select ( .name == "<channel>") | \` `select( .package == "<package_name>")' \` `<catalog_name>.json`

Table 5.3. Common bundle queries

Query	Request
Bundles in a package	`$ jq -s '.[] | select( .schema == "olm.bundle" ) | \` `select( .package == "<package_name>") | .name' \` `<catalog_name>.json`
• Bundle dependencies • Available APIs	`$ jq -s '.[] | select( .schema == "olm.bundle" ) | \` `select ( .name == "<bundle_name>") | \` `select( .package == "<package_name>")' \` `<catalog_name>.json`

5.1.3. Creating a service account to manage cluster extensions

Unlike existing Operator Lifecycle Manager (OLM), OLM v1 does not have permissions to install, update, and manage cluster extensions. Cluster administrators must create a service account and assign the role-based access controls (RBAC) required to install, update, and manage cluster extensions.

IMPORTANT

There is a known issue in OLM v1. If you do not assign the correct role-based access controls (RBAC) to an extension's service account, OLM v1 gets stuck and reconciliation stops.

Currently, OLM v1 does not have tools to help extension administrators find the correct RBAC for a service account.

Because OLM v1 is a Technology Preview feature and must not be used on production clusters, you can avoid this issue by using the more permissive RBAC included in the documentation.

This RBAC is intended for testing purposes only. Do not use it on production clusters.

Prerequisites

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

Procedure

1. Create a service account, similar to the following example:

```
apiVersion: v1
kind: ServiceAccount
metadata:
  name: <extension>-installer
  namespace: <namespace>
```

Example 5.7. Example extension-service-account.yaml file

```
apiVersion: v1
kind: ServiceAccount
metadata:
  name: pipelines-installer
  namespace: pipelines
```

2. Apply the service account by running the following command:

```
$ oc apply -f extension-service-account.yaml
```

3. Create a cluster role and assign RBAC, similar to the following example:

WARNING

The following cluster role does not follow the principle of least privilege. This cluster role is intended for testing purposes only. Do not use it on production clusters.

```
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRole
metadata:
  name: <extension>-installer-clusterrole
rules:
- apiGroups: ["*"]
  resources: ["*"]
  verbs: ["*"]
```

Example 5.8. Example pipelines-cluster-role.yaml file

```
apiVersion: rbac.authorization.k8s.io/v1
```

```
kind: ClusterRole
metadata:
  name: pipelines-installer-clusterrole
rules:
- apiGroups: ["*"]
  resources: ["*"]
  verbs: ["*"]
```

4. Add the cluster role to the cluster by running the following command:

```
$ oc apply -f pipelines-role.yaml
```

5. Bind the permissions granted by the cluster role to the service account by creating a cluster role binding, similar to the following example:

```
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRoleBinding
metadata:
  name: <extension>-installer-binding
roleRef:
  apiGroup: rbac.authorization.k8s.io
  kind: ClusterRole
  name: <extension>-installer-clusterrole
subjects:
- kind: ServiceAccount
  name: <extension>-installer
  namespace: <namespace>
```

Example 5.9. Example pipelines-cluster-role-binding.yaml file

```
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRoleBinding
metadata:
  name: pipelines-installer-binding
roleRef:
  apiGroup: rbac.authorization.k8s.io
  kind: ClusterRole
  name: pipelines-installer-clusterrole
subjects:
- kind: ServiceAccount
  name: pipelines-installer
  namespace: pipelines
```

6. Apply the cluster role binding by running the following command:

```
$ oc apply -f pipelines-cluster-role-binding.yaml
```

5.1.4. Installing a cluster extension from a catalog

You can install an extension from a catalog by creating a custom resource (CR) and applying it to the cluster. Operator Lifecycle Manager (OLM) v1 supports installing cluster extensions, including existing

OLM Operators via the **registry+v1** bundle format, that are scoped to the cluster. For more information, see *Supported extensions*.

IMPORTANT

Currently, Operator Lifecycle Manager (OLM) v1 cannot authenticate private registries, such as the Red Hat-provided Operator catalogs. This is a known issue. As a result, the OLM v1 procedures that rely on having the Red Hat Operators catalog installed do not work. (OCPBUGS-36364)

Prerequisites

- You have added a catalog to your cluster.
- You have downloaded a local copy of the catalog file.
- You have installed the **jq** CLI tool.
- You have created a service account and assigned enough role-based access controls (RBAC) to install, update, and manage the extension you want to install. For more information, see *Creating a service account*.

Procedure

1. Inspect a package for channel and version information from a local copy of your catalog file by completing the following steps:

   a. Get a list of channels from a selected package by running the following command:

```
$ jq -s '.[] | select( .schema == "olm.channel" ) | \
  select( .package == "<package_name>") | \
  .name' /<path>/<catalog_name>.json
```

Example 5.10. Example command

```
$ jq -s '.[] | select( .schema == "olm.channel" ) | \
  select( .package == "openshift-pipelines-operator-rh") | \
  .name' /home/username/rhoc.json
```

Example 5.11. Example output

```
"latest"
"pipelines-1.11"
"pipelines-1.12"
"pipelines-1.13"
"pipelines-1.14"
```

   b. Get a list of the versions published in a channel by running the following command:

```
$ jq -s '.[] | select( .package == "<package_name>" ) | \
  select( .schema == "olm.channel" ) | \
select( .name == "<channel_name>" ) | .entries | \
.[] | .name' /<path>/<catalog_name>.json
```

Example 5.12. Example command

```
$ jq -s '.[] | select( .package == "openshift-pipelines-operator-rh" ) | \
select( .schema == "olm.channel" ) | select( .name == "latest" ) | \
.entries | .[] | .name' /home/username/rhoc.json
```

Example 5.13. Example output

```
"openshift-pipelines-operator-rh.v1.12.0"
"openshift-pipelines-operator-rh.v1.12.1"
"openshift-pipelines-operator-rh.v1.12.2"
"openshift-pipelines-operator-rh.v1.13.0"
"openshift-pipelines-operator-rh.v1.13.1"
"openshift-pipelines-operator-rh.v1.11.1"
"openshift-pipelines-operator-rh.v1.12.0"
"openshift-pipelines-operator-rh.v1.12.1"
"openshift-pipelines-operator-rh.v1.12.2"
"openshift-pipelines-operator-rh.v1.13.0"
"openshift-pipelines-operator-rh.v1.14.1"
"openshift-pipelines-operator-rh.v1.14.2"
"openshift-pipelines-operator-rh.v1.14.3"
"openshift-pipelines-operator-rh.v1.14.4"
```

2. If you want to install your extension into a new namespace, run the following command:

```
$ oc adm new-project <new_namespace>
```

3. Create a CR, similar to the following example:

Example pipelines-operator.yaml CR

```
apiVersion: olm.operatorframework.io/v1alpha1
kind: ClusterExtension
metadata:
  name: pipelines-operator
spec:
  packageName: openshift-pipelines-operator-rh
  installNamespace: <namespace>
  serviceAccount:
    name: <service_account>
  channel: <channel>
  version: "<version>"
```

where:

<namespace>

Specifies the namespace where you want the bundle installed, such as **pipelines** or **my-extension**. Extensions are still cluster-scoped and might contain resources that are installed in different namespaces.

<service_account>

Specifies the name of the service account you created to install, update, and manage your extension.

<channel>

Optional: Specifies the channel, such as **pipelines-1.11** or **latest**, for the package you want to install or update.

<version>

Optional: Specifies the version or version range, such as **1.11.1**, **1.12.x**, or **>=1.12.1**, of the package you want to install or update. For more information, see "Example custom resources (CRs) that specify a target version" and "Support for version ranges".

IMPORTANT

If you try to install an Operator or extension that does not have unique name, the installation might fail or lead to an unpredictable result. This occurs for the following reasons:

- If mulitple catalogs are installed on a cluster, Operator Lifecycle Manager (OLM) v1 does not include a mechanism to specify a catalog when you install an Operator or extension.
- OLM v1 requires that all of the Operators and extensions that are available to install on a cluster use a unique name for their bundles and packages.

4. Apply the CR to the cluster by running the following command:

```
$ oc apply -f pipeline-operator.yaml
```

Example output

```
clusterextension.olm.operatorframework.io/pipelines-operator created
```

Verification

1. View the Operator or extension's CR in the YAML format by running the following command:

```
$ oc get clusterextension pipelines-operator -o yaml
```

Example 5.14. Example output

```
apiVersion: v1
items:
- apiVersion: olm.operatorframework.io/v1alpha1
  kind: ClusterExtension
  metadata:
    annotations:
      kubectl.kubernetes.io/last-applied-configuration: |
{"apiVersion":"olm.operatorframework.io/v1alpha1","kind":"ClusterExtension","metadata":
{"annotations":{},"name":"pipelines-operator"},"spec":
{"channel":"latest","installNamespace":"pipelines","packageName":"openshift-pipelines-
operator-rh","serviceAccount":{"name":"pipelines-installer"},"pollInterval":"30m"}}
    creationTimestamp: "2024-06-10T17:50:51Z"
    finalizers:
    - olm.operatorframework.io/cleanup-unpack-cache
    generation: 1
    name: pipelines-operator
    resourceVersion: "53324"
    uid: c54237be-cde4-46d4-9b31-d0ec6acc19bf
  spec:
    channel: latest
    installNamespace: pipelines
    packageName: openshift-pipelines-operator-rh
    serviceAccount:
      name: pipelines-installer
    upgradeConstraintPolicy: Enforce
  status:
    conditions:
    - lastTransitionTime: "2024-06-10T17:50:58Z"
      message: resolved to "registry.redhat.io/openshift-pipelines/pipelines-operator-
bundle@sha256:dd3d18367da2be42539e5dde8e484dac3df33ba3ce1d5bcf896838954f386
4ec"
      observedGeneration: 1
      reason: Success
      status: "True"
      type: Resolved
    - lastTransitionTime: "2024-06-10T17:51:11Z"
      message: installed from "registry.redhat.io/openshift-pipelines/pipelines-operator-
bundle@sha256:dd3d18367da2be42539e5dde8e484dac3df33ba3ce1d5bcf896838954f386
4ec"
      observedGeneration: 1
      reason: Success
      status: "True"
      type: Installed
    - lastTransitionTime: "2024-06-10T17:50:58Z"
      message: ""
      observedGeneration: 1
      reason: Deprecated
      status: "False"
      type: Deprecated
    - lastTransitionTime: "2024-06-10T17:50:58Z"
      message: ""
      observedGeneration: 1
      reason: Deprecated
      status: "False"
      type: PackageDeprecated
    - lastTransitionTime: "2024-06-10T17:50:58Z"
      message: ""
      observedGeneration: 1
      reason: Deprecated
      status: "False"
      type: ChannelDeprecated
    - lastTransitionTime: "2024-06-10T17:50:58Z"
      message: ""
      observedGeneration: 1
      reason: Deprecated
      status: "False"
      type: BundleDeprecated
    - lastTransitionTime: "2024-06-10T17:50:58Z"
      message: 'unpack successful:
      observedGeneration: 1
      reason: UnpackSuccess
      status: "True"
      type: Unpacked
    installedBundle:
      name: openshift-pipelines-operator-rh.v1.14.4
      version: 1.14.4
    resolvedBundle:
      name: openshift-pipelines-operator-rh.v1.14.4
      version: 1.14.4
```

where:

spec.channel
: Displays the channel defined in the CR of the extension.

spec.version
: Displays the version or version range defined in the CR of the extension.

status.conditions
: Displays information about the status and health of the extension.

type: Deprecated
: Displays whether one or more of following are deprecated:

  type: PackageDeprecated
  : Displays whether the resolved package is deprecated.

  type: ChannelDeprecated
  : Displays whether the resolved channel is deprecated.

  type: BundleDeprecated
  : Displays whether the resolved bundle is deprecated.

  The value of **False** in the **status** field indicates that the **reason: Deprecated** condition is not deprecated. The value of **True** in the **status** field indicates that the **reason: Deprecated** condition is deprecated.

installedBundle.name
: Displays the name of the bundle installed.

installedBundle.version
: Displays the version of the bundle installed.

resolvedBundle.name
: Displays the name of the resolved bundle.

resolvedBundle.version
: Displays the version of the resolved bundle.

Additional resources

- Supported extensions
- Creating a service account
- Example custom resources (CRs) that specify a target version
- Support for version ranges

5.1.5. Updating a cluster extension

You can update your cluster extension or Operator by manually editing the custom resource (CR) and applying the changes.

Prerequisites

- You have a catalog installed.
- You have downloaded a local copy of the catalog file.
- You have an Operator or extension installed.
- You have installed the **jq** CLI tool.

Procedure

1. Inspect a package for channel and version information from a local copy of your catalog file by completing the following steps:

   a. Get a list of channels from a selected package by running the following command:

   ```
   $ jq -s '.[] | select( .schema == "olm.channel" ) | \
     select( .package == "<package_name>") | \
     .name' /<path>/<catalog_name>.json
   ```

   Example 5.15. Example command

   ```
   $ jq -s '.[] | select( .schema == "olm.channel" ) | \
     select( .package == "openshift-pipelines-operator-rh") | \
     .name' /home/username/rhoc.json
   ```

   Example 5.16. Example output

   ```
   "latest"
   "pipelines-1.11"
   "pipelines-1.12"
   "pipelines-1.13"
   "pipelines-1.14"
   ```

   b. Get a list of the versions published in a channel by running the following command:

```
$ jq -s '.[] | select( .package == "<package_name>" ) | \
  select( .schema == "olm.channel" ) | \
  select( .name == "<channel_name>" ) | .entries | \
  .[] | .name' /<path>/<catalog_name>.json
```

Example 5.17. Example command

```
$ jq -s '.[] | select( .package == "openshift-pipelines-operator-rh" ) | \
select( .schema == "olm.channel" ) | select( .name == "latest" ) | \
.entries | .[] | .name' /home/username/rhoc.json
```

Example 5.18. Example output

```
"openshift-pipelines-operator-rh.v1.11.1"
"openshift-pipelines-operator-rh.v1.12.0"
"openshift-pipelines-operator-rh.v1.12.1"
"openshift-pipelines-operator-rh.v1.12.2"
"openshift-pipelines-operator-rh.v1.13.0"
"openshift-pipelines-operator-rh.v1.14.1"
"openshift-pipelines-operator-rh.v1.14.2"
"openshift-pipelines-operator-rh.v1.14.3"
"openshift-pipelines-operator-rh.v1.14.4"
```

2. Find out what version or channel is specified in your Operator or extension's CR by running the following command:

```
$ oc get clusterextension <operator_name> -o yaml
```

Example command

```
$ oc get clusterextension pipelines-operator -o yaml
```

Example 5.19. Example output

```
apiVersion: olm.operatorframework.io/v1alpha1
kind: ClusterExtension
metadata:
  annotations:
    kubectl.kubernetes.io/last-applied-configuration: |

{"apiVersion":"olm.operatorframework.io/v1alpha1","kind":"ClusterExtension","metadata":
{"annotations":{},"name":"pipelines-operator"},"spec":
{"channel":"latest","installNamespace":"openshift-operators","packageName":"openshift-
pipelines-operator-rh","pollInterval":"30m","version":"\u003c1.12"}}
  creationTimestamp: "2024-06-11T15:55:37Z"
  generation: 1
  name: pipelines-operator
  resourceVersion: "69776"
  uid: 6a11dff3-bfa3-42b8-9e5f-d8babbd6486f
spec:
  channel: latest
  installNamespace: openshift-operators
  packageName: openshift-pipelines-operator-rh
  upgradeConstraintPolicy: Enforce
  version: <1.12
status:
  conditions:
  - lastTransitionTime: "2024-06-11T15:56:09Z"
    message: installed from "registry.redhat.io/openshift-pipelines/pipelines-operator-
bundle@sha256:e09d37bb1e754db42324fd18c1cb3e7ce77e7b7fcbf4932d0535391579938
280"
    observedGeneration: 1
    reason: Success
    status: "True"
    type: Installed
  - lastTransitionTime: "2024-06-11T15:55:50Z"
    message: resolved to "registry.redhat.io/openshift-pipelines/pipelines-operator-
bundle@sha256:e09d37bb1e754db42324fd18c1cb3e7ce77e7b7fcbf4932d0535391579938
280"
    observedGeneration: 1
    reason: Success
    status: "True"
    type: Resolved
  - lastTransitionTime: "2024-06-11T15:55:50Z"
    message: ""
    observedGeneration: 1
    reason: Deprecated
    status: "False"
    type: Deprecated
  - lastTransitionTime: "2024-06-11T15:55:50Z"
    message: ""
    observedGeneration: 1
    reason: Deprecated
    status: "False"
    type: PackageDeprecated
  - lastTransitionTime: "2024-06-11T15:55:50Z"
    message: ""
    observedGeneration: 1
    reason: Deprecated
    status: "False"
    type: ChannelDeprecated
  - lastTransitionTime: "2024-06-11T15:55:50Z"
    message: ""
    observedGeneration: 1
    reason: Deprecated
    status: "False"
    type: BundleDeprecated
  installedBundle:
    name: openshift-pipelines-operator-rh.v1.11.1
    version: 1.11.1
  resolvedBundle:
    name: openshift-pipelines-operator-rh.v1.11.1
    version: 1.11.1
```

3. Edit your CR by using one of the following methods:

- If you want to pin your Operator or extension to specific version, such as **1.12.1**, edit your CR similar to the following example:

  Example pipelines-operator.yaml CR

  ```
  apiVersion: olm.operatorframework.io/v1alpha1
  kind: ClusterExtension
  metadata:
    name: pipelines-operator
  spec:
    packageName: openshift-pipelines-operator-rh
    installNamespace: <namespace>
    version: "1.12.1" 1
  ```

  1 Update the version from **1.11.1** to **1.12.1**

- If you want to define a range of acceptable update versions, edit your CR similar to the following example:

  Example CR with a version range specified

  ```
  apiVersion: olm.operatorframework.io/v1alpha1
  kind: ClusterExtension
  metadata:
    name: pipelines-operator
  spec:
    packageName: openshift-pipelines-operator-rh
    installNamespace: <namespace>
    version: ">1.11.1, <1.13" 1
  ```

  1 Specifies that the desired version range is greater than version **1.11.1** and less than **1.13**. For more information, see "Support for version ranges" and "Version comparison strings".

- If you want to update to the latest version that can be resolved from a channel, edit your CR similar to the following example:

  Example CR with a specified channel

  ```
  apiVersion: olm.operatorframework.io/v1alpha1
  kind: ClusterExtension
  metadata:
    name: pipelines-operator
  spec:
    packageName: openshift-pipelines-operator-rh
    installNamespace: <namespace>
    channel: pipelines-1.13 1
  ```

  1 Installs the latest release that can be resolved from the specified channel. Updates to the channel are automatically installed.

- If you want to specify a channel and version or version range, edit your CR similar to the following example:

  Example CR with a specified channel and version range

  ```
  apiVersion: olm.operatorframework.io/v1alpha1
  kind: ClusterExtension
  metadata:
    name: pipelines-operator
  spec:
    packageName: openshift-pipelines-operator-rh
    installNamespace: <namespace>
    channel: latest
    version: "<1.13"
  ```

  For more information, see "Example custom resources (CRs) that specify a target version".

4. Apply the update to the cluster by running the following command:

   ```
   $ oc apply -f pipelines-operator.yaml
   ```

   Example output

   ```
   clusterextension.olm.operatorframework.io/pipelines-operator configured
   ```

   TIP

   You can patch and apply the changes to your CR from the CLI by running the following command:

   ```
   $ oc patch clusterextension/pipelines-operator -p \
     '{"spec":{"version":"<1.13"}}' \
     --type=merge
   ```

   Example output

   ```
   clusterextension.olm.operatorframework.io/pipelines-operator patched
   ```

Verification

- Verify that the channel and version updates have been applied by running the following command:

  ```
  $ oc get clusterextension pipelines-operator -o yaml
  ```

  Example 5.20. Example output

  ```
  apiVersion: olm.operatorframework.io/v1alpha1
  kind: ClusterExtension
  metadata:
    annotations:
      kubectl.kubernetes.io/last-applied-configuration: |
  ```

```
{"apiVersion":"olm.operatorframework.io/v1alpha1","kind":"ClusterExtension","metadata":
{"annotations":{},"name":"pipelines-operator"},"spec":
{"channel":"latest","installNamespace":"openshift-operators","packageName":"openshift-
pipelines-operator-rh","pollInterval":"30m","version":"\u003c1.13"}}
  creationTimestamp: "2024-06-11T18:23:26Z"
  generation: 2
  name: pipelines-operator
  resourceVersion: "66310"
  uid: ce0416ba-13ea-4069-a6c8-e5efcbc47537
spec:
  channel: latest
  installNamespace: openshift-operators
  packageName: openshift-pipelines-operator-rh
  upgradeConstraintPolicy: Enforce
  version: <1.13
status:
  conditions:
  - lastTransitionTime: "2024-06-11T18:23:33Z"
    message: resolved to "registry.redhat.io/openshift-pipelines/pipelines-operator-
bundle@sha256:814742c8a7cc7e2662598e114c35c13993a7b423cfe92548124e43ea5d46
9f82"
    observedGeneration: 2
    reason: Success
    status: "True"
    type: Resolved
  - lastTransitionTime: "2024-06-11T18:23:52Z"
    message: installed from "registry.redhat.io/openshift-pipelines/pipelines-operator-
bundle@sha256:814742c8a7cc7e2662598e114c35c13993a7b423cfe92548124e43ea5d46
9f82"
    observedGeneration: 2
    reason: Success
    status: "True"
    type: Installed
  - lastTransitionTime: "2024-06-11T18:23:33Z"
    message: ""
    observedGeneration: 2
    reason: Deprecated
    status: "False"
    type: Deprecated
  - lastTransitionTime: "2024-06-11T18:23:33Z"
    message: ""
    observedGeneration: 2
    reason: Deprecated
    status: "False"
    type: PackageDeprecated
  - lastTransitionTime: "2024-06-11T18:23:33Z"
    message: ""
    observedGeneration: 2
    reason: Deprecated
    status: "False"
    type: ChannelDeprecated
  - lastTransitionTime: "2024-06-11T18:23:33Z"
    message: ""
    observedGeneration: 2
    reason: Deprecated
    status: "False"
    type: BundleDeprecated
  installedBundle:
    name: openshift-pipelines-operator-rh.v1.12.2
    version: 1.12.2
  resolvedBundle:
    name: openshift-pipelines-operator-rh.v1.12.2
    version: 1.12.2
```

Troubleshooting

- If you specify a target version or channel that is deprecated or does not exist, you can run the following command to check the status of your extension:

```
$ oc get clusterextension <operator_name> -o yaml
```

Example 5.21. Example output for a version that does not exist

```
apiVersion: olm.operatorframework.io/v1alpha1
kind: ClusterExtension
metadata:
  annotations:
    kubectl.kubernetes.io/last-applied-configuration: |

{"apiVersion":"olm.operatorframework.io/v1alpha1","kind":"ClusterExtension","metadata":
{"annotations":{},"name":"pipelines-operator"},"spec":
{"channel":"latest","installNamespace":"openshift-operators","packageName":"openshift-
pipelines-operator-rh","pollInterval":"30m","version":"3.0"}}
  creationTimestamp: "2024-06-11T18:23:26Z"
  generation: 3
  name: pipelines-operator
  resourceVersion: "71852"
  uid: ce0416ba-13ea-4069-a6c8-e5efcbc47537
spec:
  channel: latest
  installNamespace: openshift-operators
  packageName: openshift-pipelines-operator-rh
  upgradeConstraintPolicy: Enforce
  version: "3.0"
status:
  conditions:
  - lastTransitionTime: "2024-06-11T18:29:02Z"
    message: 'error upgrading from currently installed version "1.12.2": no package
      "openshift-pipelines-operator-rh" matching version "3.0" found in channel "latest"'
    observedGeneration: 3
    reason: ResolutionFailed
    status: "False"
    type: Resolved
  - lastTransitionTime: "2024-06-11T18:29:02Z"
    message: installation has not been attempted as resolution failed
    observedGeneration: 3
    reason: InstallationStatusUnknown
    status: Unknown
    type: Installed
  - lastTransitionTime: "2024-06-11T18:29:02Z"
    message: deprecation checks have not been attempted as resolution failed
    observedGeneration: 3
    reason: Deprecated
    status: Unknown
    type: Deprecated
  - lastTransitionTime: "2024-06-11T18:29:02Z"
    message: deprecation checks have not been attempted as resolution failed
    observedGeneration: 3
    reason: Deprecated
    status: Unknown
    type: PackageDeprecated
  - lastTransitionTime: "2024-06-11T18:29:02Z"
    message: deprecation checks have not been attempted as resolution failed
    observedGeneration: 3
    reason: Deprecated
    status: Unknown
    type: ChannelDeprecated
  - lastTransitionTime: "2024-06-11T18:29:02Z"
    message: deprecation checks have not been attempted as resolution failed
    observedGeneration: 3
    reason: Deprecated
    status: Unknown
    type: BundleDeprecated
```

Additional resources

- Upgrade edges

5.1.6. Deleting an Operator

You can delete an Operator and its custom resource definitions (CRDs) by deleting the **ClusterExtension** custom resource (CR).

Prerequisites

- You have a catalog installed.
- You have an Operator installed.

Procedure

- Delete an Operator and its CRDs by running the following command:

  ```
  $ oc delete clusterextension <operator_name>
  ```

  #### Example output

  ```
  clusterextension.olm.operatorframework.io "<operator_name>" deleted
  ```

Verification

- Run the following commands to verify that your Operator and its resources were deleted:
  - Verify the Operator is deleted by running the following command:

    ```
    $ oc get clusterextensions
    ```

    Example output

    ```
    No resources found
    ```

  - Verify that the Operator’s system namespace is deleted by running the following command:

    ```
    $ oc get ns <operator_name>-system
    ```

    Example output

    ```
    Error from server (NotFound): namespaces "<operator_name>-system" not found
    ```

5.2. UPGRADE EDGES

IMPORTANT

Operator Lifecycle Manager (OLM) v1 is a Technology Preview feature only. Technology Preview features are not supported with Red Hat production service level agreements (SLAs) and might not be functionally complete. Red Hat does not recommend using them in production. These features provide early access to upcoming product features, enabling customers to test functionality and provide feedback during the development process.

For more information about the support scope of Red Hat Technology Preview features, see Technology Preview Features Support Scope .

When determining upgrade edges, also known as upgrade paths or upgrade constraints, for an installed cluster extension, Operator Lifecycle Manager (OLM) v1 supports existing OLM semantics starting in OpenShift Container Platform 4.16. This support follows the behavior from existing OLM, including **replaces**, **skips**, and **skipRange** directives, with a few noted differences.

By supporting existing OLM semantics, OLM v1 now honors the upgrade graph from catalogs accurately.

IMPORTANT

Currently, Operator Lifecycle Manager (OLM) v1 cannot authenticate private registries, such as the Red Hat-provided Operator catalogs. This is a known issue. As a result, the OLM v1 procedures that rely on having the Red Hat Operators catalog installed do not work. (OCPBUGS-36364)

Differences from original existing OLM implementation

- If there are multiple possible successors, OLM v1 behavior differs in the following ways:
  - In existing OLM, the successor closest to the channel head is chosen.

    - In OLM v1, the successor with the highest semantic version (semver) is chosen.

- Consider the following set of file-based catalog (FBC) channel entries:

```
# ...
- name: example.v3.0.0
  skips: ["example.v2.0.0"]
- name: example.v2.0.0
  skipRange: >=1.0.0 <2.0.0
```

  If **1.0.0** is installed, OLM v1 behavior differs in the following ways:

    - Existing OLM will not detect an upgrade edge to **v2.0.0** because **v2.0.0** is skipped and not on the **replaces** chain.
    - OLM v1 will detect the upgrade edge because OLM v1 does not have a concept of a **replaces** chain. OLM v1 finds all entries that have a **replace**, **skip**, or **skipRange** value that covers the currently installed version.

Additional resources

- Existing OLM upgrade semantics

5.2.1. Support for version ranges

In Operator Lifecycle Manager (OLM) v1, you can specify a version range by using a comparison string in an Operator or extension's custom resource (CR). If you specify a version range in the CR, OLM v1 installs or updates to the latest version of the Operator that can be resolved within the version range.

Resolved version workflow

- The resolved version is the latest version of the Operator that satisfies the constraints of the Operator and the environment.
- An Operator update within the specified range is automatically installed if it is resolved successfully.
- An update is not installed if it is outside of the specified range or if it cannot be resolved successfully.

5.2.2. Version comparison strings

You can define a version range by adding a comparison string to the **spec.version** field in an Operator or extension's custom resource (CR). A comparison string is a list of space- or comma-separated values and one or more comparison operators enclosed in double quotation marks (**"**). You can add another comparison string by including an **OR**, or double vertical bar (**||**), comparison operator between the strings.

Table 5.4. Basic comparisons

Comparison operator	Definition
=	Equal to

Comparison operator	Definition
!=	Not equal to
>	Greater than
<	Less than
>=	Greater than or equal to
<=	Less than or equal to

You can specify a version range in an Operator or extension’s CR by using a range comparison similar to the following example:

Example version range comparison

```
apiVersion: olm.operatorframework.io/v1alpha1
kind: ClusterExtension
metadata:
  name: pipelines-operator
spec:
  packageName: openshift-pipelines-operator-rh
  installNamespace: <namespace_name>
  version: ">=1.11, <1.13"
```

You can use wildcard characters in all types of comparison strings. OLM v1 accepts **x**, **X**, and asterisks (*****) as wildcard characters. When you use a wildcard character with the equal sign (**=**) comparison operator, you define a comparison at the patch or minor version level.

Table 5.5. Example wildcard characters in comparison strings

Wildcard comparison	Matching string
1.11.x	**>=1.11.0, <1.12.0**
>=1.12.X	**>=1.12.0**
<=2.x	**<3**
*****	**>=0.0.0**

You can make patch release comparisons by using the tilde (**~**) comparison operator. Patch release comparisons specify a minor version up to the next major version.

Table 5.6. Example patch release comparisons

Patch release comparison	Matching string
~1.11.0	**>=1.11.0, <1.12.0**
~1	**>=1, <2**
~1.12	**>=1.12, <1.13**
~1.12.x	**>=1.12.0, <1.13.0**
~1.x	**>=1, <2**

You can use the caret (**^**) comparison operator to make a comparison for a major release. If you make a major release comparison before the first stable release is published, the minor versions define the API's level of stability. In the semantic versioning (semver) specification, the first stable release is published as the **1.0.0** version.

Table 5.7. Example major release comparisons

Major release comparison	Matching string
^0	**>=0.0.0, <1.0.0**
^0.0	**>=0.0.0, <0.1.0**
^0.0.3	**>=0.0.3, <0.0.4**
^0.2	**>=0.2.0, <0.3.0**
^0.2.3	**>=0.2.3, <0.3.0**
^1.2.x	**>= 1.2.0, < 2.0.0**
^1.2.3	**>= 1.2.3, < 2.0.0**
^2.x	**>= 2.0.0, < 3**
^2.3	**>= 2.3, < 3**

5.2.3. Example custom resources (CRs) that specify a target version

In Operator Lifecycle Manager (OLM) v1, cluster administrators can declaratively set the target version of an Operator or extension in the custom resource (CR).

You can define a target version by specifying any of the following fields:

- Channel
- Version number

- Version range

If you specify a channel in the CR, OLM v1 installs the latest version of the Operator or extension that can be resolved within the specified channel. When updates are published to the specified channel, OLM v1 automatically updates to the latest release that can be resolved from the channel.

Example CR with a specified channel

```
apiVersion: olm.operatorframework.io/v1alpha1
kind: ClusterExtension
metadata:
  name: pipelines-operator
spec:
  packageName: openshift-pipelines-operator-rh
  installNamespace: <namespace_name>
  serviceAccount:
    name: <service_account>
  channel: latest 1
```

1 Installs the latest release that can be resolved from the specified channel. Updates to the channel are automatically installed.

If you specify the Operator or extension's target version in the CR, OLM v1 installs the specified version. When the target version is specified in the CR, OLM v1 does not change the target version when updates are published to the catalog.

If you want to update the version of the Operator that is installed on the cluster, you must manually edit the Operator's CR. Specifying an Operator's target version pins the Operator's version to the specified release.

Example CR with the target version specified

```
apiVersion: olm.operatorframework.io/v1alpha1
kind: ClusterExtension
metadata:
  name: pipelines-operator
spec:
  packageName: openshift-pipelines-operator-rh
  installNamespace: <namespace_name>
  serviceAccount:
    name: <service_account>
  version: "1.11.1" 1
```

1 Specifies the target version. If you want to update the version of the Operator or extension that is installed, you must manually update this field the CR to the desired target version.

If you want to define a range of acceptable versions for an Operator or extension, you can specify a version range by using a comparison string. When you specify a version range, OLM v1 installs the latest version of an Operator or extension that can be resolved by the Operator Controller.

Example CR with a version range specified

```
apiVersion: olm.operatorframework.io/v1alpha1
```

```
kind: ClusterExtension
metadata:
  name: pipelines-operator
spec:
  packageName: openshift-pipelines-operator-rh
  installNamespace: <namespace_name>
  serviceAccount:
    name: <service_account>
  version: ">1.11.1" (1)
```

(1) Specifies that the desired version range is greater than version **1.11.1**. For more information, see "Support for version ranges".

After you create or update a CR, apply the configuration file by running the following command:

Command syntax

```
$ oc apply -f <extension_name>.yaml
```

5.2.4. Forcing an update or rollback

OLM v1 does not support automatic updates to the next major version or rollbacks to an earlier version. If you want to perform a major version update or rollback, you must verify and force the update manually.

WARNING

You must verify the consequences of forcing a manual update or rollback. Failure to verify a forced update or rollback might have catastrophic consequences such as data loss.

Prerequisites

- You have a catalog installed.
- You have an Operator or extension installed.
- You have created a service account and assigned enough role-based access controls (RBAC) to install, update, and manage the extension you want to install. For more information, see *Creating a service account* .

Procedure

1. Edit the custom resource (CR) of your Operator or extension as shown in the following example:

   Example CR

   ```
   apiVersion: olm.operatorframework.io/v1alpha1
   kind: Operator
   metadata:
   ```

```
  name: <operator_name> (1)
spec:
  packageName: <package_name> (2)
  installNamespace: <namespace_name>
  serviceAccount:
    name: <service_account>
  version: <version> (3)
  upgradeConstraintPolicy: Ignore (4)
```

1. Specifies the name of the Operator or extension, such as **pipelines-operator**
2. Specifies the package name, such as **openshift-pipelines-operator-rh**.
3. Specifies the blocked update or rollback version.
4. Optional: Specifies the upgrade constraint policy. To force an update or rollback, set the field to **Ignore**. If unspecified, the default setting is **Enforce**.

2. Apply the changes to your Operator or extensions CR by running the following command:

```
$ oc apply -f <extension_name>.yaml
```

Additional resources

- Support for version ranges

5.3. CUSTOM RESOURCE DEFINITION (CRD) UPGRADE SAFETY

IMPORTANT

Operator Lifecycle Manager (OLM) v1 is a Technology Preview feature only. Technology Preview features are not supported with Red Hat production service level agreements (SLAs) and might not be functionally complete. Red Hat does not recommend using them in production. These features provide early access to upcoming product features, enabling customers to test functionality and provide feedback during the development process.

For more information about the support scope of Red Hat Technology Preview features, see Technology Preview Features Support Scope .

When you update a custom resource definition (CRD) that is provided by a cluster extension, Operator Lifecycle Manager (OLM) v1 runs a CRD upgrade safety preflight check to ensure backwards compatibility with previous versions of that CRD. The CRD update must pass the validation checks before the change is allowed to progress on a cluster.

Additional resources

- Updating a cluster extension

5.3.1. Prohibited CRD upgrade changes

The following changes to an existing custom resource definition (CRD) are caught by the CRD upgrade safety preflight check and prevent the upgrade:

- A new required field is added to an existing version of the CRD
- An existing field is removed from an existing version of the CRD
- An existing field type is changed in an existing version of the CRD
- A new default value is added to a field that did not previously have a default value
- The default value of a field is changed
- An existing default value of a field is removed
- New enum restrictions are added to an existing field which did not previously have enum restrictions
- Existing enum values from an existing field are removed
- The minimum value of an existing field is increased in an existing version
- The maximum value of an existing field is decreased in an existing version
- Minimum or maximum field constraints are added to a field that did not previously have constraints

NOTE

The rules for changes to minimum and maximum values apply to **minimum**, **minLength**, **minProperties**, **minItems**, **maximum**, **maxLength**, **maxProperties**, and **maxItems** constraints.

The following changes to an existing CRD are reported by the CRD upgrade safety preflight check and prevent the upgrade, though the operations are technically handled by the Kubernetes API server:

- The scope changes from **Cluster** to **Namespace** or from **Namespace** to **Cluster**
- An existing stored version of the CRD is removed

If the CRD upgrade safety preflight check encounters one of the prohibited upgrade changes, it logs an error for each prohibited change detected in the CRD upgrade.

TIP

In cases where a change to the CRD does not fall into one of the prohibited change categories, but is also unable to be properly detected as allowed, the CRD upgrade safety preflight check will prevent the upgrade and log an error for an "unknown change".

5.3.2. Allowed CRD upgrade changes

The following changes to an existing custom resource definition (CRD) are safe for backwards compatibility and will not cause the CRD upgrade safety preflight check to halt the upgrade:

- Adding new enum values to the list of allowed enum values in a field

- An existing required field is changed to optional in an existing version
- The minimum value of an existing field is decreased in an existing version
- The maximum value of an existing field is increased in an existing version
- A new version of the CRD is added with no modifications to existing versions

5.3.3. Disabling CRD upgrade safety preflight check

The custom resource definition (CRD) upgrade safety preflight check can be disabled by adding the **preflight.crdUpgradeSafety.disabled** field with a value of **true** to the **ClusterExtension** object that provides the CRD.

WARNING

Disabling the CRD upgrade safety preflight check could break backwards compatibility with stored versions of the CRD and cause other unintended consequences on the cluster.

You cannot disable individual field validators. If you disable the CRD upgrade safety preflight check, all field validators are disabled.

NOTE

The following checks are handled by the Kubernetes API server:

- The scope changes from **Cluster** to **Namespace** or from **Namespace** to **Cluster**
- An existing stored version of the CRD is removed

After disabling the CRD upgrade safety preflight check via Operator Lifecycle Manager (OLM) v1, these two operations are still prevented by Kubernetes.

Prerequisites

- You have a cluster extension installed.

Procedure

1. Edit the **ClusterExtension** object of the CRD:

```
$ oc edit clusterextension <clusterextension_name>
```

2. Set the **preflight.crdUpgradeSafety.disabled** field to **true**:

Example 5.22. Example ClusterExtension object

```
apiVersion: olm.operatorframework.io/v1alpha1
kind: ClusterExtension
```

```
metadata:
  name: clusterextension-sample
spec:
  installNamespace: default
  packageName: argocd-operator
  version: 0.6.0
  preflight:
    crdUpgradeSafety:
      disabled: true (1)
```

(1) Set to **true**.

5.3.4. Examples of unsafe CRD changes

The following examples demonstrate specific changes to sections of an example custom resource definition (CRD) that would be caught by the CRD upgrade safety preflight check.

For the following examples, consider a CRD object in the following starting state:

Example 5.23. Example CRD object

```
apiVersion: apiextensions.k8s.io/v1
kind: CustomResourceDefinition
metadata:
  annotations:
    controller-gen.kubebuilder.io/version: v0.13.0
  name: example.test.example.com
spec:
  group: test.example.com
  names:
    kind: Sample
    listKind: SampleList
    plural: samples
    singular: sample
  scope: Namespaced
  versions:
  - name: v1alpha1
    schema:
      openAPIV3Schema:
        properties:
          apiVersion:
            type: string
          kind:
            type: string
          metadata:
            type: object
          spec:
            type: object
          status:
            type: object
          pollInterval:
            type: string
        type: object
    served: true
    storage: true
    subresources:
      status: {}
```

5.3.4.1. Scope change

In the following custom resource definition (CRD) example, the **scope** field is changed from **Namespaced** to **Cluster**:

Example 5.24. Example scope change in a CRD

```
spec:
  group: test.example.com
  names:
    kind: Sample
    listKind: SampleList
    plural: samples
    singular: sample
  scope: Cluster
  versions:
  - name: v1alpha1
```

Example 5.25. Example error output

```
validating upgrade for CRD "test.example.com" failed: CustomResourceDefinition
test.example.com failed upgrade safety validation. "NoScopeChange" validation failed: scope
changed from "Namespaced" to "Cluster"
```

5.3.4.2. Removal of a stored version

In the following custom resource definition (CRD) example, the existing stored version, **v1alpha1**, is removed:

Example 5.26. Example removal of a stored version in a CRD

```
  versions:
  - name: v1alpha2
    schema:
      openAPIV3Schema:
        properties:
          apiVersion:
            type: string
          kind:
            type: string
          metadata:
            type: object
          spec:
            type: object
            status:
              type: object
            pollInterval:
              type: string
          type: object
```

Example 5.27. Example error output

```
validating upgrade for CRD "test.example.com" failed: CustomResourceDefinition
test.example.com failed upgrade safety validation. "NoStoredVersionRemoved" validation failed:
stored version "v1alpha1" removed
```

5.3.4.3. Removal of an existing field

In the following custom resource definition (CRD) example, the **pollInterval** property field is removed from the **v1alpha1** schema:

Example 5.28. Example removal of an existing field in a CRD

```
versions:
- name: v1alpha1
  schema:
    openAPIV3Schema:
      properties:
        apiVersion:
          type: string
        kind:
          type: string
        metadata:
          type: object
        spec:
          type: object
        status:
          type: object
      type: object
```

Example 5.29. Example error output

```
validating upgrade for CRD "test.example.com" failed: CustomResourceDefinition
test.example.com failed upgrade safety validation. "NoExistingFieldRemoved" validation failed:
crd/test.example.com version/v1alpha1 field/^.spec.pollInterval may not be removed
```

5.3.4.4. Addition of a required field

In the following custom resource definition (CRD) example, the **pollInterval** property has been changed to a required field:

Example 5.30. Example addition of a required field in a CRD

```
versions:
- name: v1alpha2
  schema:
    openAPIV3Schema:
      properties:
        apiVersion:
          type: string
        kind:
          type: string
        metadata:
          type: object
        spec:
          type: object
        status:
          type: object
        pollInterval:
          type: string
      type: object
      required:
      - pollInterval
```

Example 5.31. Example error output

```
validating upgrade for CRD "test.example.com" failed: CustomResourceDefinition
test.example.com failed upgrade safety validation. "ChangeValidator" validation failed: version
"v1alpha1", field "^": new required fields added: [pollInterval]
```