

# **OpenShift Container Platform 4.13**

## **Operators**

Working with Operators in OpenShift Container Platform

Last Updated: 2023-10-12

## OpenShift Container Platform 4.13 Operators

Working with Operators in OpenShift Container Platform

## **Legal Notice**

Copyright © 2023 Red Hat, Inc.

The text of and illustrations in this document are licensed by Red Hat under a Creative Commons Attribution–Share Alike 3.0 Unported license ("CC-BY-SA"). An explanation of CC-BY-SA is available at

http://creativecommons.org/licenses/by-sa/3.0/

. In accordance with CC-BY-SA, if you distribute this document or an adaptation of it, you must provide the URL for the original version.

Red Hat, as the licensor of this document, waives the right to enforce, and agrees not to assert, Section 4d of CC-BY-SA to the fullest extent permitted by applicable law.

Red Hat, Red Hat Enterprise Linux, the Shadowman logo, the Red Hat logo, JBoss, OpenShift, Fedora, the Infinity logo, and RHCE are trademarks of Red Hat, Inc., registered in the United States and other countries.

Linux ® is the registered trademark of Linus Torvalds in the United States and other countries.

Java <sup>®</sup> is a registered trademark of Oracle and/or its affiliates.

XFS <sup>®</sup> is a trademark of Silicon Graphics International Corp. or its subsidiaries in the United States and/or other countries.

MySQL ® is a registered trademark of MySQL AB in the United States, the European Union and other countries.

Node.js ® is an official trademark of Joyent. Red Hat is not formally related to or endorsed by the official Joyent Node.js open source or commercial project.

The OpenStack <sup>®</sup> Word Mark and OpenStack logo are either registered trademarks/service marks or trademarks/service marks of the OpenStack Foundation, in the United States and other countries and are used with the OpenStack Foundation's permission. We are not affiliated with, endorsed or sponsored by the OpenStack Foundation, or the OpenStack community.

All other trademarks are the property of their respective owners.

## **Abstract**

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

## **Table of Contents**

CHAPTER 1. OPERATORS OVERVIEW	14
1.1. FOR DEVELOPERS	14
1.2. FOR ADMINISTRATORS	14
1.3. NEXT STEPS	15
CHAPTER 2. UNDERSTANDING OPERATORS	16
2.1. WHAT ARE OPERATORS?	16
2.1.1. Why use Operators?	16
2.1.2. Operator Framework	16
2.1.3. Operator maturity model	17
2.2. OPERATOR FRAMEWORK PACKAGING FORMAT	18
2.2.1. Bundle format	18
2.2.1.1. Manifests	18
Additionally supported objects	18
2.2.1.2. Annotations	19
2.2.1.3. Dependencies	20
2.2.1.4. About the opm CLI	21
2.2.2. File-based catalogs	21
2.2.2.1. Directory structure	22
2.2.2.2. Schemas	23
2.2.2.2.1. olm.package schema	24
2.2.2.2. olm.channel schema	24
2.2.2.3. olm.bundle schema	25
2.2.2.3. Properties	26
2.2.2.3.1. olm.package property	26
2.2.2.3.2. olm.gvk property	26
2.2.2.3.3. olm.package.required	27
2.2.2.3.4. olm.gvk.required	27
2.2.2.4. Example catalog	27
2.2.2.5. Guidelines	28
2.2.2.5.1. Immutable bundles	28
2.2.2.5.2. Source control	29
2.2.2.6. CLI usage	29
2.2.2.7. Automation	29
2.2.3. RukPak (Technology Preview)	29
2.2.3.1. Bundle	30
2.2.3.1.1. Bundle immutability	30
Further immutability considerations	31
2.2.3.1.2. Plain bundle spec	32
2.2.3.1.3. Registry bundle spec	33
2.2.3.2. BundleDeployment	33
2.2.3.3. Provisioner	34
2.3. OPERATOR FRAMEWORK GLOSSARY OF COMMON TERMS	34
2.3.1. Common Operator Framework terms	34
2.3.1.1. Bundle	34
2.3.1.2. Bundle image	34
2.3.1.3. Catalog source	34
2.3.1.4. Channel	34
2.3.1.5. Channel head	34
2.3.1.6. Cluster service version	34
2.3.1.7. Dependency	35

2.3.1.8. Index image	35
2.3.1.9. Install plan	35
2.3.1.10. Multitenancy	35
2.3.1.11. Operator group	35
2.3.1.12. Package	35
2.3.1.13. Registry	35
2.3.1.14. Subscription	35
2.3.1.15. Update graph	36
2.4. OPERATOR LIFECYCLE MANAGER (OLM)	36
2.4.1. Operator Lifecycle Manager concepts and resources	36
2.4.1.1. What is Operator Lifecycle Manager?	36
2.4.1.2. OLM resources	36
2.4.1.2.1. Cluster service version	37
2.4.1.2.2. Catalog source	37
2.4.1.2.2.1. Image template for custom catalog sources	40
2.4.1.2.2.2. Catalog health requirements	42
2.4.1.2.3. Subscription	42
2.4.1.2.4. Install plan	43
2.4.1.2.5. Operator groups	45
2.4.1.2.6. Operator conditions	45
2.4.2. Operator Lifecycle Manager architecture	46
2.4.2.1. Component responsibilities	46
2.4.2.2. OLM Operator	47
2.4.2.3. Catalog Operator	47
2.4.2.4. Catalog Registry	48
2.4.3. Operator Lifecycle Manager workflow	48
2.4.3.1. Operator installation and upgrade workflow in OLM	48
2.4.3.1.1. Example upgrade path	50
2.4.3.1.2. Skipping upgrades	50
2.4.3.1.3. Replacing multiple Operators	52
2.4.3.1.4. Z-stream support	53
2.4.4. Operator Lifecycle Manager dependency resolution	54
2.4.4.1. About dependency resolution	54
2.4.4.2. Operator properties	54
2.4.4.2.1. Arbitrary properties	55
2.4.4.3. Operator dependencies	55
2.4.4.4. Generic constraints	56
2.4.4.4.1. Common Expression Language (CEL) constraints	56
2.4.4.4.2. Compound constraints (all, any, not)	57
2.4.4.4.3. Nested compound constraints	58
2.4.4.5. Dependency preferences	59
2.4.4.5.1. Catalog priority	59
2.4.4.5.2. Channel ordering	60
2.4.4.5.3. Order within a channel	60
2.4.4.5.4. Other constraints	60
2.4.4.5.4.1. Subscription constraint	61
2.4.4.5.4.2. Package constraint	61
2.4.4.5.5. Additional resources	61
2.4.4.6. CRD upgrades	61
2.4.4.7. Dependency best practices	61
2.4.4.8. Dependency caveats	62
2.4.4.9. Example dependency resolution scenarios	63
Example: Deprecating dependent APIs	63

Example: Version deadlock	63
2.4.5. Operator groups	63
2.4.5.1. About Operator groups	63
2.4.5.2. Operator group membership	64
2.4.5.3. Target namespace selection	64
2.4.5.4. Operator group CSV annotations	65
2.4.5.5. Provided APIs annotation	66
2.4.5.6. Role-based access control	66
2.4.5.7. Copied CSVs	69
2.4.5.8. Static Operator groups	70
2.4.5.9. Operator group intersection	71
Rules for intersection	71
2.4.5.10. Limitations for multitenant Operator management	72
2.4.5.11. Troubleshooting Operator groups	73
Membership	73
2.4.6. Multitenancy and Operator colocation	73
2.4.6.1. Colocation of Operators in a namespace	73
2.4.7. Operator conditions	74
2.4.7.1. About Operator conditions	74
2.4.7.2. Supported conditions	74
2.4.7.2.1. Upgradeable condition	74
2.4.7.3. Additional resources	75
2.4.8. Operator Lifecycle Manager metrics	75
2.4.8.1. Exposed metrics	75
2.4.9. Webhook management in Operator Lifecycle Manager	76
2.4.9.1. Additional resources	76
2.5. UNDERSTANDING OPERATORHUB	77
2.5.1. About Operator Hub	77
2.5.2. OperatorHub architecture	77
2.5.2.1. OperatorHub custom resource	78
2.5.3. Additional resources	78
2.6. RED HAT-PROVIDED OPERATOR CATALOGS	78
2.6.1. About Operator catalogs	79
2.6.2. About Red Hat-provided Operator catalogs	80
2.7. OPERATORS IN MULTITENANT CLUSTERS	81
2.7.1. Default Operator install modes and behavior	81
2.7.2. Recommended solution for multitenant clusters	82
2.7.3. Operator colocation and Operator groups	83
2.8. CRDS	83
2.8.1. Extending the Kubernetes API with custom resource definitions	83
2.8.1.1. Custom resource definitions	83
2.8.1.2. Creating a custom resource definition	84
2.8.1.3. Creating cluster roles for custom resource definitions	85
2.8.1.4. Creating custom resources from a file	87
2.8.1.5. Inspecting custom resources	88
2.8.2. Managing resources from custom resource definitions	89
2.8.2.1. Custom resource definitions	89
2.8.2.2. Creating custom resources from a file	89
2.8.2.3. Inspecting custom resources	90
CHAPTER 3. USER TASKS	92
3.1. CREATING APPLICATIONS FROM INSTALLED OPERATORS	92
3.1.1. Creating an etcd cluster using an Operator	92

3.2. INSTALLING OPERATORS IN YOUR NAMESPACE	93
3.2.1. Prerequisites	93
3.2.2. About Operator installation with OperatorHub	93
3.2.3. Installing from OperatorHub using the web console	94
3.2.4. Installing from OperatorHub using the CLI	95
3.2.5. Installing a specific version of an Operator	98
CHAPTER 4. ADMINISTRATOR TASKS	102
4.1. ADDING OPERATORS TO A CLUSTER	102
4.1.1. About Operator installation with OperatorHub	102
4.1.2. Installing from OperatorHub using the web console	102
4.1.3. Installing from OperatorHub using the CLI	104
4.1.4. Installing a specific version of an Operator	107
4.1.5. Preparing for multiple instances of an Operator for multitenant clusters	110
4.1.6. Installing global Operators in custom namespaces	112
4.1.7. Pod placement of Operator workloads	113
4.1.8. Controlling where an Operator is installed	114
4.2. UPDATING INSTALLED OPERATORS	117
4.2.1. Preparing for an Operator update	118
4.2.2. Changing the update channel for an Operator	118
4.2.3. Manually approving a pending Operator update	119
4.3. DELETING OPERATORS FROM A CLUSTER	119
4.3.1. Deleting Operators from a cluster using the web console	119
4.3.2. Deleting Operators from a cluster using the CLI	120 121
4.3.3. Refreshing failing subscriptions 4.4. CONFIGURING OPERATOR LIFECYCLE MANAGER FEATURES	
	122
4.4.1. Disabling copied CSVs 4.5. CONFIGURING PROXY SUPPORT IN OPERATOR LIFECYCLE MANAGER	123 124
	124
4.5.1. Overriding proxy settings of an Operator 4.5.2. Injecting a custom CA certificate	124
4.6. VIEWING OPERATOR STATUS	128
4.6.1. Operator subscription condition types	128
4.6.2. Viewing Operator subscription status by using the CLI	128
4.6.3. Viewing Operator satisfication status by using the CLI	129
4.7. MANAGING OPERATOR CONDITIONS	131
4.7.1. Overriding Operator conditions	131
4.7.2. Updating your Operator to use Operator conditions	132
4.7.2.1. Setting defaults	132
4.7.3. Additional resources	132
4.8. ALLOWING NON-CLUSTER ADMINISTRATORS TO INSTALL OPERATORS	133
4.8.1. Understanding Operator installation policy	133
4.8.1.1. Installation scenarios	133
4.8.1.2. Installation workflow	134
4.8.2. Scoping Operator installations	134
4.8.2.1. Fine-grained permissions	136
4.8.3. Operator catalog access control	137
4.8.4. Troubleshooting permission failures	138
4.9. MANAGING CUSTOM CATALOGS	139
4.9.1. Prerequisites	139
4.9.2. File-based catalogs	139
4.9.2.1. Creating a file-based catalog image	140
4.9.2.2. Updating or filtering a file-based catalog image	142
4.9.3. SQLite-based catalogs	145

4.9.3.1. Creating a SQLite-based index image	145
4.9.3.2. Updating a SQLite-based index image	146
4.9.3.3. Filtering a SQLite-based index image	148
4.9.4. Catalog sources and pod security admission	149
4.9.4.1. Migrating SQLite database catalogs to the file-based catalog format	150
4.9.4.2. Rebuilding SQLite database catalog images	151
4.9.4.3. Configuring catalogs to run with elevated permissions	151
4.9.5. Adding a catalog source to a cluster	153
4.9.6. Accessing images for Operators from private registries	155
4.9.7. Disabling the default OperatorHub catalog sources	159
4.9.8. Removing custom catalogs	160
4.10. USING OPERATOR LIFECYCLE MANAGER ON RESTRICTED NETWORKS	160
4.10.1. Prerequisites	161
4.10.2. Disabling the default OperatorHub catalog sources	162
4.10.3. Mirroring an Operator catalog	162
4.10.4. Adding a catalog source to a cluster	162
4.11. CATALOG SOURCE POD SCHEDULING	164
4.11.1. Disabling default CatalogSource objects at a local level	165
4.11.2. Overriding the node selector for catalog source pods	166
4.11.3. Overriding the priority class name for catalog source pods	166
4.11.4. Overriding tolerations for catalog source pods	167
4.12. MANAGING PLATFORM OPERATORS (TECHNOLOGY PREVIEW)	167
4.12.1. About platform Operators	168
4.12.1.1. Technology Preview restrictions for platform Operators	168
4.12.2. Prerequisites	169
4.12.3. Installing platform Operators during cluster creation	170
4.12.4. Installing platform Operators after cluster creation	172
4.12.5. Deleting platform Operators	174
CHAPTER 5. DEVELOPING OPERATORS	176
5.1. ABOUT THE OPERATOR SDK	176
5.1.1. What are Operators?	176
5.1.2. Development workflow	176
5.1.3. Additional resources	177
5.2. INSTALLING THE OPERATOR SDK CLI	177
5.2.1. Installing the Operator SDK CLI on Linux	177
5.2.2. Installing the Operator SDK CLI on macOS	178
5.3. GO-BASED OPERATORS	179
5.3.1. Getting started with Operator SDK for Go-based Operators	179
5.3.1.1. Prerequisites	179
•	
5.3.1.2. Creating and deploying Go-based Operators	180
·	180 181
5.3.1.2. Creating and deploying Go-based Operators	
5.3.1.2. Creating and deploying Go-based Operators 5.3.1.3. Next steps	181
5.3.1.2. Creating and deploying Go-based Operators 5.3.1.3. Next steps 5.3.2. Operator SDK tutorial for Go-based Operators	181 181
<ul><li>5.3.1.2. Creating and deploying Go-based Operators</li><li>5.3.1.3. Next steps</li><li>5.3.2. Operator SDK tutorial for Go-based Operators</li><li>5.3.2.1. Prerequisites</li></ul>	181 181 182
<ul> <li>5.3.1.2. Creating and deploying Go-based Operators</li> <li>5.3.1.3. Next steps</li> <li>5.3.2. Operator SDK tutorial for Go-based Operators</li> <li>5.3.2.1. Prerequisites</li> <li>5.3.2.2. Creating a project</li> <li>5.3.2.2.1. PROJECT file</li> <li>5.3.2.2.2. About the Manager</li> </ul>	181 182 182 183 183
<ul> <li>5.3.1.2. Creating and deploying Go-based Operators</li> <li>5.3.1.3. Next steps</li> <li>5.3.2. Operator SDK tutorial for Go-based Operators</li> <li>5.3.2.1. Prerequisites</li> <li>5.3.2.2. Creating a project</li> <li>5.3.2.2.1. PROJECT file</li> <li>5.3.2.2.2. About the Manager</li> <li>5.3.2.2.3. About multi-group APIs</li> </ul>	181 181 182 182 183
<ul> <li>5.3.1.2. Creating and deploying Go-based Operators</li> <li>5.3.1.3. Next steps</li> <li>5.3.2. Operator SDK tutorial for Go-based Operators</li> <li>5.3.2.1. Prerequisites</li> <li>5.3.2.2. Creating a project</li> <li>5.3.2.2.1. PROJECT file</li> <li>5.3.2.2.2. About the Manager</li> <li>5.3.2.2.3. About multi-group APIs</li> <li>5.3.2.3. Creating an API and controller</li> </ul>	181 182 182 183 183 183
<ul> <li>5.3.1.2. Creating and deploying Go-based Operators</li> <li>5.3.1.3. Next steps</li> <li>5.3.2. Operator SDK tutorial for Go-based Operators</li> <li>5.3.2.1. Prerequisites</li> <li>5.3.2.2. Creating a project</li> <li>5.3.2.2.1. PROJECT file</li> <li>5.3.2.2.2. About the Manager</li> <li>5.3.2.2.3. About multi-group APIs</li> <li>5.3.2.3. Creating an API and controller</li> <li>5.3.2.3.1. Defining the API</li> </ul>	181 182 182 183 183 183 184 184
5.3.1.2. Creating and deploying Go-based Operators 5.3.1.3. Next steps 5.3.2. Operator SDK tutorial for Go-based Operators 5.3.2.1. Prerequisites 5.3.2.2. Creating a project 5.3.2.2.1. PROJECT file 5.3.2.2.2. About the Manager 5.3.2.2.3. About multi-group APIs 5.3.2.3. Creating an API and controller 5.3.2.3.1. Defining the API 5.3.2.3.2. Generating CRD manifests	181 182 182 183 183 184 184 185
<ul> <li>5.3.1.2. Creating and deploying Go-based Operators</li> <li>5.3.1.3. Next steps</li> <li>5.3.2. Operator SDK tutorial for Go-based Operators</li> <li>5.3.2.1. Prerequisites</li> <li>5.3.2.2. Creating a project</li> <li>5.3.2.2.1. PROJECT file</li> <li>5.3.2.2.2. About the Manager</li> <li>5.3.2.2.3. About multi-group APIs</li> <li>5.3.2.3. Creating an API and controller</li> <li>5.3.2.3.1. Defining the API</li> </ul>	181 182 182 183 183 183 184 184

5.3.2.4.1. Resources watched by the controller	190
5.3.2.4.2. Controller configurations	191
5.3.2.4.3. Reconcile loop	191
5.3.2.4.4. Permissions and RBAC manifests	192
5.3.2.5. Enabling proxy support	192
5.3.2.6. Running the Operator	193
5.3.2.6.1. Running locally outside the cluster	194
5.3.2.6.2. Running as a deployment on the cluster	194
5.3.2.6.3. Bundling an Operator and deploying with Operator Lifecycle Manager	195
5.3.2.6.3.1. Bundling an Operator	195
5.3.2.6.3.2. Deploying an Operator with Operator Lifecycle Manager	197
5.3.2.7. Creating a custom resource	198
5.3.2.8. Additional resources	200
5.3.3. Project layout for Go-based Operators	200
5.3.3.1. Go-based project layout	200
5.3.4. Updating Go-based Operator projects for newer Operator SDK versions	201
5.3.4.1. Updating Go-based Operator projects for Operator SDK 1.28.0	201
5.3.4.2. Additional resources	202
5.4. ANSIBLE-BASED OPERATORS	202
5.4.1. Getting started with Operator SDK for Ansible-based Operators	202
5.4.1.1. Prerequisites	202
5.4.1.2. Creating and deploying Ansible-based Operators	203
5.4.1.3. Next steps	205
5.4.2. Operator SDK tutorial for Ansible-based Operators	205
5.4.2.1. Prerequisites	205
5.4.2.2. Creating a project	206
5.4.2.2.1. PROJECT file	206
5.4.2.3. Creating an API	206
5.4.2.4. Modifying the manager	207
5.4.2.5. Enabling proxy support	208
5.4.2.6. Running the Operator	209
5.4.2.6.1. Running locally outside the cluster	209
5.4.2.6.2. Running as a deployment on the cluster	210
5.4.2.6.3. Bundling an Operator and deploying with Operator Lifecycle Manager	211
5.4.2.6.3.1. Bundling an Operator	211
5.4.2.6.3.2. Deploying an Operator with Operator Lifecycle Manager	212
5.4.2.7. Creating a custom resource	213
5.4.2.8. Additional resources	216
5.4.3. Project layout for Ansible-based Operators	216
5.4.3.1. Ansible-based project layout	216
5.4.4. Updating projects for newer Operator SDK versions	217
5.4.4.1. Updating Ansible-based Operator projects for Operator SDK 1.28.0	217
5.4.4.2. Additional resources	218
5.4.5. Ansible support in Operator SDK	218
5.4.5.1. Custom resource files	218
5.4.5.2. watches.yaml file	219
5.4.5.2.1. Advanced options	220
5.4.5.3. Extra variables sent to Ansible	221
5.4.5.4. Ansible Runner directory	222
5.4.6. Kubernetes Collection for Ansible	222
5.4.6.1. Installing the Kubernetes Collection for Ansible	222
5.4.6.2. Testing the Kubernetes Collection locally	223
5.4.6.3. Next steps	225

5.4.7. Using Ansible inside an Operator	225
5.4.7.1. Custom resource files	225
5.4.7.2. Testing an Ansible-based Operator locally	226
5.4.7.3. Testing an Ansible-based Operator on the cluster	228
5.4.7.4. Ansible logs	230
5.4.7.4.1. Viewing Ansible logs	230
5.4.7.4.2. Enabling full Ansible results in logs	231
5.4.7.4.3. Enabling verbose debugging in logs	231
5.4.8. Custom resource status management	231
5.4.8.1. About custom resource status in Ansible-based Operators	231
5.4.8.2. Tracking custom resource status manually	232
5.5. HELM-BASED OPERATORS	233
5.5.1. Getting started with Operator SDK for Helm-based Operators	233
5.5.1.1. Prerequisites	233
5.5.1.2. Creating and deploying Helm-based Operators	233
5.5.1.3. Next steps	235
5.5.2. Operator SDK tutorial for Helm-based Operators	235
5.5.2.1. Prerequisites	235
5.5.2.2. Creating a project	236
5.5.2.2.1. Existing Helm charts	236
5.5.2.2. PROJECT file	238
5.5.2.3. Understanding the Operator logic	238
5.5.2.3.1. Sample Helm chart	238
5.5.2.3.2. Modifying the custom resource spec	239
5.5.2.4. Enabling proxy support	239
5.5.2.5. Running the Operator	241
5.5.2.5.1. Running locally outside the cluster	241
5.5.2.5.2. Running as a deployment on the cluster	241
5.5.2.5.3. Bundling an Operator and deploying with Operator Lifecycle Manager	242
5.5.2.5.3.1. Bundling an Operator	242
5.5.2.5.3.2. Deploying an Operator with Operator Lifecycle Manager	244
5.5.2.6. Creating a custom resource	245
5.5.2.7. Additional resources	247
5.5.3. Project layout for Helm-based Operators	247
5.5.3.1. Helm-based project layout	247
5.5.4. Updating Helm-based projects for newer Operator SDK versions	248
5.5.4.1. Updating Helm-based Operator projects for Operator SDK 1.28.0	248
5.5.4.2. Additional resources	249
5.5.5. Helm support in Operator SDK	249
5.5.5.1. Helm charts	249
5.5.6. Operator SDK tutorial for Hybrid Helm Operators	250
5.5.6.1. Prerequisites	250
5.5.6.2. Creating a project	251
5.5.6.3. Creating a Helm API	251
5.5.6.3.1. Operator logic for the Helm API	252
5.5.6.3.2. Custom Helm reconciler configurations using provided library APIs	252
5.5.6.4. Creating a Go API	253
5.5.6.4.1. Defining the API	253
5.5.6.4.2. Controller implementation	254
5.5.6.4.3. Differences in main.go	255
5.5.6.4.4. Permissions and RBAC manifests	256
5.5.6.5. Running locally outside the cluster	258
5.5.6.6. Running as a deployment on the cluster	259

5.5.6.7. Creating custom resources	259
5.5.6.8. Project layout	262
5.5.7. Updating Hybrid Helm-based projects for newer Operator SDK versions	263
5.5.7.1. Updating Hybrid Helm-based Operator projects for Operator SDK 1.28.0	264
5.5.7.2. Additional resources	264
5.6. JAVA-BASED OPERATORS	264
5.6.1. Getting started with Operator SDK for Java-based Operators	264
5.6.1.1. Prerequisites	265
5.6.1.2. Creating and deploying Java-based Operators	265
5.6.1.3. Next steps	267
5.6.2. Operator SDK tutorial for Java-based Operators	267
5.6.2.1. Prerequisites	267
5.6.2.2. Creating a project	268
5.6.2.2.1. PROJECT file	268
5.6.2.3. Creating an API and controller	268
5.6.2.3.1. Defining the API	269
5.6.2.3.2. Generating CRD manifests	270
5.6.2.3.3. Creating a Custom Resource	271
5.6.2.4. Implementing the controller	272
5.6.2.4.1. Reconcile loop	274
5.6.2.4.2. Defining labelsForMemcached	276
5.6.2.4.3. Define the createMemcachedDeployment	276
5.6.2.5. Running the Operator	276
5.6.2.5.1. Running locally outside the cluster	277
5.6.2.5.2. Running as a deployment on the cluster	278
5.6.2.5.3. Bundling an Operator and deploying with Operator Lifecycle Manager	280
5.6.2.5.3.1. Bundling an Operator	280
5.6.2.5.3.2. Deploying an Operator with Operator Lifecycle Manager	281
5.6.2.6. Additional resources	283
5.6.3. Project layout for Java-based Operators	283
5.6.3.1. Java-based project layout	283
5.6.4. Updating projects for newer Operator SDK versions	284
5.6.4.1. Updating Java-based Operator projects for Operator SDK 1.28.0	284
5.6.4.2. Additional resources	284
5.7. DEFINING CLUSTER SERVICE VERSIONS (CSVS)	284
5.7.1. How CSV generation works	285
5.7.1.1. Generated files and resources	285
5.7.1.2. Version management	286
5.7.2. Manually-defined CSV fields	286
5.7.2.1. Operator metadata annotations	287
Example use cases	289
5.7.3. Enabling your Operator for restricted network environments	289
5.7.4. Enabling your Operator for multiple architectures and operating systems	293
5.7.4.1. Architecture and operating system support for Operators	294
5.7.5. Setting a suggested namespace	295
5.7.6. Setting a suggested namespace with default node selector	295
5.7.7. Enabling Operator conditions	296
5.7.8. Defining webhooks	298
5.7.8.1. Webhook considerations for OLM	300
Certificate authority constraints	300
Admission webhook rules constraints	301
Conversion webhook constraints	301
5.7.9. Understanding your custom resource definitions (CRDs)	301

5.7.9.1. Owned CRDs	301
5.7.9.2. Required CRDs	304
5.7.9.3. CRD upgrades	305
5.7.9.3.1. Adding a new CRD version	305
5.7.9.3.2. Deprecating or removing a CRD version	306
5.7.9.4. CRD templates	307
5.7.9.5. Hiding internal objects	307
5.7.9.6. Initializing required custom resources	308
5.7.10. Understanding your API services	309
5.7.10.1. Owned API services	309
5.7.10.1.1. API service resource creation	310
5.7.10.1.2. API service serving certificates	310
5.7.10.2. Required API services	311
5.8. WORKING WITH BUNDLE IMAGES	311
5.8.1. Bundling an Operator	311
5.8.2. Deploying an Operator with Operator Lifecycle Manager	313
5.8.3. Publishing a catalog containing a bundled Operator	314
5.8.4. Testing an Operator upgrade on Operator Lifecycle Manager	317
5.8.5. Controlling Operator compatibility with OpenShift Container Platform versions	319
5.8.6. Additional resources	322
5.9. COMPLYING WITH POD SECURITY ADMISSION	322
5.9.1. Security context constraint synchronization with pod security standards	322
5.9.2. Ensuring Operator workloads run in namespaces set to the restricted pod security level	323
5.9.3. Managing pod security admission for Operator workloads that require escalated permissions	324
5.9.4. Additional resources	325
5.10. VALIDATING OPERATORS USING THE SCORECARD TOOL	325
5.10.1. About the scorecard tool	326
5.10.2. Scorecard configuration	326
5.10.3. Built-in scorecard tests	327
5.10.4. Running the scorecard tool	328
5.10.5. Scorecard output	329
5.10.6. Selecting tests	330
5.10.7. Enabling parallel testing	330
5.10.8. Custom scorecard tests	331
5.11. VALIDATING OPERATOR BUNDLES	334
5.11.1. About the bundle validate command	334
5.11.2. Built-in bundle validate tests	335
5.11.3. Running the bundle validate command	336
5.12. HIGH-AVAILABILITY OR SINGLE-NODE CLUSTER DETECTION AND SUPPORT	337
5.12.1. About the cluster high-availability mode API	337
5.12.2. Example API usage in Operator projects	338
5.13. CONFIGURING BUILT-IN MONITORING WITH PROMETHEUS	339
5.13.1. Prometheus Operator support	339
5.13.2. Exposing custom metrics for Go-based Operators	339
5.13.3. Exposing custom metrics for Ansible-based Operators	341
5.14. CONFIGURING LEADER ELECTION	345
5.14.1. Operator leader election examples	345
5.14.1.1. Leader-for-life election	345
5.14.1.2. Leader-with-lease election	346
5.15. OBJECT PRUNING UTILITY FOR GO-BASED OPERATORS	346
5.15.1. About the operator-lib pruning utility	346
5.15.2. Pruning utility configuration	347
5.16. MIGRATING PACKAGE MANIFEST PROJECTS TO BUNDLE FORMAT	348

5.16.1. About packaging format migration	348
5.16.2. Migrating a package manifest project to bundle format	349
5.17. OPERATOR SDK CLI REFERENCE	350
5.17.1. bundle	351
5.17.1.1. validate	351
5.17.2. cleanup	351
·	352
5.17.3. completion	
5.17.4.1 cm:	352
5.17.4.1. api	352
5.17.5. generate	353
5.17.5.1. bundle	353
5.17.5.2. kustomize	354
5.17.5.2.1. manifests	354
5.17.6. init	355
5.17.7. run	355
5.17.7.1. bundle	355
5.17.7.2. bundle-upgrade	356
5.17.8. scorecard	357
CHAPTER 6. CLUSTER OPERATORS REFERENCE	359
6.1. CLUSTER BAREMETAL OPERATOR	359
Purpose	359
Project	359
6.2. BARE METAL EVENT RELAY	359
Purpose	359
Configuration objects	359
Project	360
CRD	360
6.3. CLOUD CREDENTIAL OPERATOR	360
Purpose	360
·	360
Project	
CRDs	360
Configuration objects	360
Additional resources	361
6.4. CLUSTER AUTHENTICATION OPERATOR	361
Purpose	361
Project	361
6.5. CLUSTER AUTOSCALER OPERATOR	361
Purpose	361
Project	361
CRDs	361
6.6. CLUSTER CLOUD CONTROLLER MANAGER OPERATOR	361
Purpose	361
Project	362
6.7. CLUSTER CAPI OPERATOR	362
Purpose	362
Project	362
CRDs	362
6.8. CLUSTER CONFIG OPERATOR	363
Purpose	363
Project	363
6.9. CLUSTER CSI SNAPSHOT CONTROLLER OPERATOR	363
Purpose	363
i diposs	505

Project	363
6.10. CLUSTER IMAGE REGISTRY OPERATOR	363
Purpose	363
Project	364
6.11. CLUSTER MACHINE APPROVER OPERATOR	364
Purpose	364
Project	364
6.12. CLUSTER MONITORING OPERATOR	364
Purpose	364
Project	364
CRDs	364
Configuration objects	365
6.13. CLUSTER NETWORK OPERATOR	365
Purpose	365
6.14. CLUSTER SAMPLES OPERATOR	365
Purpose	365
Project	366
6.15. CLUSTER STORAGE OPERATOR	366
Purpose	366
Project	366
Configuration	366
Notes	366
6.16. CLUSTER VERSION OPERATOR	366
Purpose	366
Project	367
6.17. CONSOLE OPERATOR	367
Purpose	367
Project	367
6.18. CONTROL PLANE MACHINE SET OPERATOR	367
Purpose	367
Project	367
CRDs	367
Additional resources	368
6.19. DNS OPERATOR	368
Purpose	368
Project	368
6.20. ETCD CLUSTER OPERATOR	368
Purpose	368
Project	368
CRDs	368
Configuration objects	368
6.21. INGRESS OPERATOR	368
Purpose	368
Project	368
CRDs	369
Configuration objects	369
Notes	369
6.22. INSIGHTS OPERATOR	369
Purpose	369
Project	370
Configuration	370
Notes	370
6.23. KUBERNETES API SERVER OPERATOR	370

Purpose	370
Project	370
CRDs	370
Configuration objects	370
6.24. KUBERNETES CONTROLLER MANAGER OPERATOR	370
Purpose	370
Project	371
6.25. KUBERNETES SCHEDULER OPERATOR	371
Purpose	371
Project	371
Configuration	371
6.26. KUBERNETES STORAGE VERSION MIGRATOR OPERATOR	371
Purpose	371
Project	371
6.27. MACHINE API OPERATOR	371
Purpose	371
Project	372
CRDs	372
6.28. MACHINE CONFIG OPERATOR	372
Purpose	372
Project	372
6.29. MARKETPLACE OPERATOR	373
Purpose	373
Project	373
6.30. NODE TUNING OPERATOR	373
Purpose	373
Project	374
Additional resources	374
6.31. OPENSHIFT API SERVER OPERATOR	374
Purpose	374
Project	374
CRDs	374
6.32. OPENSHIFT CONTROLLER MANAGER OPERATOR	374
Purpose	374
Project	375
6.33. OPERATOR LIFECYCLE MANAGER OPERATORS	375
Purpose	375
CRDs	375
OLM Operator	376
Catalog Operator	377
Catalog Registry	377
Additional resources	377
6.34. OPENSHIFT SERVICE CA OPERATOR	377
Purpose	377
Project	377
6.35. VSPHERE PROBLEM DETECTOR OPERATOR	377
Purpose	377
Configuration	378
Notes	378

## **CHAPTER 1. OPERATORS OVERVIEW**

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

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

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

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

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

## 1.1. FOR DEVELOPERS

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

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

#### Additional resources

Machine deletion lifecycle hook examples for Operator developers

## 1.2. FOR ADMINISTRATORS

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

- Manage custom catalogs.
- Allow non-cluster administrators to install Operators .
- Install an Operator from OperatorHub.
- View Operator status.

- Manage Operator conditions.
- Upgrade installed Operators.
- Delete installed Operators.
- Configure proxy support.
- Use Operator Lifecycle Manager on restricted networks .

To know all about the cluster Operators that Red Hat provides, see Cluster Operators reference.

## 1.3. NEXT STEPS

To understand more about Operators, see What are Operators?

## **CHAPTER 2. UNDERSTANDING OPERATORS**

## 2.1. WHAT ARE OPERATORS?

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

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

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

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

## 2.1.1. Why use Operators?

Operators provide:

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

#### Why deploy on Kubernetes?

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

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

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

## How do Operators compare with service brokers?

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

#### 2.1.2. Operator Framework

The Operator Framework is a family of tools and capabilities to deliver on the customer experience

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

#### **Operator SDK**

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

## **Operator Lifecycle Manager**

Operator Lifecycle Manager (OLM) controls the installation, upgrade, and role-based access control (RBAC) of Operators in a cluster. It is deployed by default in OpenShift Container Platform 4.13.

#### **Operator Registry**

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

#### **OperatorHub**

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

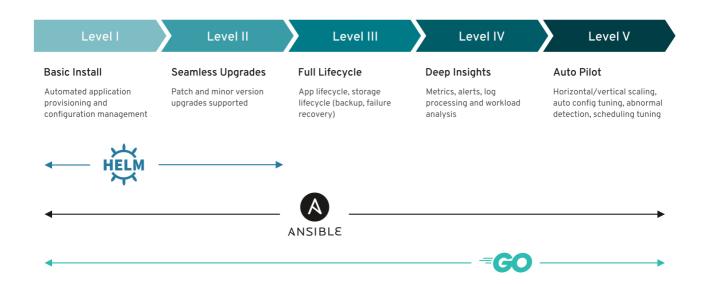
These tools are designed to be composable, so you can use any that are useful to you.

## 2.1.3. Operator maturity model

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

One can however generalize the scale of the maturity of the encapsulated operations of an Operator for certain set of capabilities that most Operators can include. To this end, the following Operator maturity model defines five phases of maturity for generic Day 2 operations of an Operator:

Figure 2.1. Operator maturity model



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

## 2.2. OPERATOR FRAMEWORK PACKAGING FORMAT

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

#### 2.2.1. Bundle format

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

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

Operator metadata can include:

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

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

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

#### 2.2.1.1. Manifests

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

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

#### Example bundle format layout



## Additionally supported objects

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

## Supported optional object types

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

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

## Lifecycle for optional objects

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

#### 2.2.1.2. Annotations

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

## Example annotations.yaml

#### annotations:

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

- The media type or format of the Operator bundle. The **registry+v1** format means it contains a CSV and its associated Kubernetes objects.
- The path in the image to the directory that contains the Operator manifests. This label is reserved for future use and currently defaults to **manifests**/. The value **manifests.v1** implies that the bundle contains Operator manifests.
- The path in the image to the directory that contains metadata files about the bundle. This label is reserved for future use and currently defaults to **metadata**/. The value **metadata.v1** implies that this bundle has Operator metadata.
- The package name of the bundle.
- The list of channels the bundle is subscribing to when added into an Operator Registry.
- The default channel an Operator should be subscribed to when installed from a registry.



#### **NOTE**

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

## 2.2.1.3. Dependencies

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

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

#### olm.package

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

#### olm.gvk

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

#### olm.constraint

This type declares generic constraints on arbitrary Operator properties.

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

## Example dependencies.yaml file

dependencies:

- type: olm.package

value:

packageName: prometheus

version: ">0.27.0" - type: olm.gvk

value:

group: etcd.database.coreos.com

kind: EtcdCluster version: v1beta2

#### Additional resources

• Operator Lifecycle Manager dependency resolution

## 2.2.1.4. About the opm CLI

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

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

• See CLI tools for steps on installing the **opm** CLI.

## 2.2.2. File-based catalogs

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.



## **IMPORTANT**

As of OpenShift Container Platform 4.11, the default Red Hat-provided Operator catalog releases in the file-based catalog format. The default Red Hat-provided Operator catalogs for OpenShift Container Platform 4.6 through 4.10 released in the deprecated SQLite database format.

The **opm** subcommands, flags, and functionality related to the SQLite database format are also deprecated and will be removed in a future release. The features are still supported and must be used for catalogs that use the deprecated SQLite database format.

Many of the **opm** subcommands and flags for working with the SQLite database format, such as **opm index prune**, do not work with the file-based catalog format. For more information about working with file-based catalogs, see Managing custom catalogs and Mirroring images for a disconnected installation using the oc-mirror plugin .

## 2.2.2.1. 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
```

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 could also be included in a parent catalog by copying it into the parent catalog's root directory.

#### 2.2.2.2. 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.

## 2.2.2.1. olm.package schema

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

## Example 2.1. 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
    }

}
```

#### 2.2.2.2. 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.

A bundle can included as an entry in multiple **olm.channel** blobs, but it can have only one entry 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 2.2. 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 & !=""
```

## 2.2.2.2.3. olm.bundle schema

## Example 2.3. 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 & !=""
}
```

## 2.2.2.3. 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.

#### 2.2.2.3.1. 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 2.4. olm.package property

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

#### 2.2.2.3.2. olm.qvk 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 2.5. olm.gvk property

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

```
version: string & !=""
kind: string & !=""
}
}
```

## 2.2.2.3.3. 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 2.6. olm.package.required property

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

#### 2.2.2.3.4. 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 2.7. olm.gvk.required property

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

## 2.2.2.4. 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@sha256:5891b5b522d5df086d0ff0b110fbd9d21bb4fc7163af34d08286a2e846f
6be03
- name: prometheus-operator
image: quay.io/prometheus-
operator/index@sha256:e258d248fda94c63753607f7c4494ee0fcbe92f1a76bfdac795c9d84101
eb317
```

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 I in $(yq e '.name as $catalog | .references[] | .image + "|" + $catalog + "/" + .name +

"/index.yaml" catalog.yaml); do
    image=$(echo $I | cut -d'|' -f1)
    file=$(echo $I | cut -d'|' -f2)
    opm render "$image" > "$file"

done

opm alpha generate dockerfile "$name"
indexImage=$(yq eval '.repo + ":" + .tag' catalog.yaml)
docker build -t "$indexImage" -f "$name.Dockerfile" .

docker push "$indexImage"
```

#### 2.2.2.5. Guidelines

Consider the following guidelines when maintaining file-based catalogs.

#### 2.2.2.5.1. 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**.

#### 2.2.2.5.2. 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.

## 2.2.2.6. 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.

## 2.2.2.7. 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.

## 2.2.3. RukPak (Technology Preview)



#### **IMPORTANT**

RukPak 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.

OpenShift Container Platform 4.12 introduces the *platform Operator* type as a Technology Preview feature. The platform Operator mechanism relies on the RukPak component, also introduced in OpenShift Container Platform 4.12, and its resources to manage content.

RukPak consists of a series of controllers, known as *provisioners*, that install and manage content on a Kubernetes cluster. RukPak also provides two primary APIs: **Bundle** and **BundleDeployment**. These components work together to bring content onto the cluster and install it, generating resources within the cluster.

A provisioner places a watch on both **Bundle** and **BundleDeployment** resources that refer to the provisioner explicitly. For a given bundle, the provisioner unpacks the contents of the **Bundle** resource onto the cluster. Then, given a **BundleDeployment** resource referring to that bundle, the provisioner installs the bundle contents and is responsible for managing the lifecycle of those resources.

Two provisioners are currently implemented and bundled with RukPak: the *plain provisioner* that sources and unpacks **plain+v0** bundles, and the *registry provisioner* that sources and unpacks Operator Lifecycle Manager (OLM) **registry+v1** bundles.

#### Additional resources

- Managing platform Operators
- Technology Preview restrictions for platform Operators

#### 2.2.3.1. Bundle

A RukPak **Bundle** object represents content to make available to other consumers in the cluster. Much like the contents of a container image must be pulled and unpacked in order for pod to start using them, **Bundle** objects are used to reference content that might need to be pulled and unpacked. In this sense, a bundle is a generalization of the image concept and can be used to represent any type of content.

Bundles cannot do anything on their own; they require a provisioner to unpack and make their content available in the cluster. They can be unpacked to any arbitrary storage medium, such as a **tar.gz** file in a directory mounted into the provisioner pods. Each **Bundle** object has an associated **spec.provisionerClassName** field that indicates the **Provisioner** object that watches and unpacks that particular bundle type.

#### Example Bundle object configured to work with the plain provisioner

api Version: core.rukpak.io/v1alpha1

kind: Bundle metadata:

name: my-bundle

spec: source: type: image image:

ref: my-bundle@sha256:xyz123

provisionerClassName: core-rukpak-io-plain



#### NOTE

Bundles are considered immutable after they are created.

#### 2.2.3.1.1. Bundle immutability

After a **Bundle** object is accepted by the API server, the bundle is considered an immutable artifact by the rest of the RukPak system. This behavior enforces the notion that a bundle represents some unique, static piece of content to source onto the cluster. A user can have confidence that a particular bundle is pointing to a specific set of manifests and cannot be updated without creating a new bundle. This property is true for both standalone bundles and dynamic bundles created by an embedded **BundleTemplate** object.

Bundle immutability is enforced by the core RukPak webhook. This webhook watches **Bundle** object events and, for any update to a bundle, checks whether the **spec** field of the existing bundle is semantically equal to that in the proposed updated bundle. If they are not equal, the update is rejected by the webhook. Other **Bundle** object fields, such as **metadata** or **status**, are updated during the bundle's lifecycle; it is only the **spec** field that is considered immutable.

Applying a **Bundle** object and then attempting to update its spec should fail. For example, the following example creates a bundle:

```
$ oc apply -f -<<EOF
apiVersion: core.rukpak.io/v1alpha1
kind: Bundle
metadata:
name: combo-tag-ref
spec:
source:
type: git
git:
ref:
tag: v0.0.2
repository: https://github.com/operator-framework/combo
provisionerClassName: core-rukpak-io-plain
EOF
```

## Example output

bundle.core.rukpak.io/combo-tag-ref created

Then, patching the bundle to point to a newer tag returns an error:

\$ oc patch bundle combo-tag-ref --type='merge' -p '{"spec":{"source":{"git":{"ref":{"tag":"v0.0.3"}}}}}'

## **Example output**

Error from server (bundle.spec is immutable): admission webhook "vbundles.core.rukpak.io" denied the request: bundle.spec is immutable

The core RukPak admission webhook rejected the patch because the spec of the bundle is immutable. The recommended method to change the content of a bundle is by creating a new **Bundle** object instead of updating it in-place.

## Further immutability considerations

While the **spec** field of the **Bundle** object is immutable, it is still possible for a **BundleDeployment** object to pivot to a newer version of bundle content without changing the underlying **spec** field. This unintentional pivoting could occur in the following scenario:

- A user sets an image tag, a Git branch, or a Git tag in the spec.source field of the Bundle object.
- 2. The image tag moves to a new digest, a user pushes changes to a Git branch, or a user deletes and re-pushes a Git tag on a different commit.
- 3. A user does something to cause the bundle unpack pod to be re-created, such as deleting the unpack pod.

If this scenario occurs, the new content from step 2 is unpacked as a result of step 3. The bundle deployment detects the changes and pivots to the newer version of the content.

This is similar to pod behavior, where one of the pod's container images uses a tag, the tag is moved to a different digest, and then at some point in the future the existing pod is rescheduled on a different node. At that point, the node pulls the new image at the new digest and runs something different without the user explicitly asking for it.

To be confident that the underlying **Bundle** spec content does not change, use a digest-based image or a Git commit reference when creating the bundle.

## 2.2.3.1.2. Plain bundle spec

A plain bundle in RukPak is a collection of static, arbitrary, Kubernetes YAML manifests in a given directory.

The currently implemented plain bundle format is the **plain+v0** format. The name of the bundle format, **plain+v0**, combines the type of bundle (**plain**) with the current schema version ( $\mathbf{v0}$ ).



#### NOTE

The **plain+v0** bundle format is at schema version **v0**, which means it is an experimental format that is subject to change.

For example, the following shows the file tree in a **plain+v0** bundle. It must have a **manifests**/ directory containing the Kubernetes resources required to deploy an application.

## Example plain+v0 bundle file tree



The static manifests must be located in the **manifests**/ directory with at least one resource in it for the bundle to be a valid **plain+v0** bundle that the provisioner can unpack. The **manifests**/ directory must also be flat; all manifests must be at the top-level with no subdirectories.



## **IMPORTANT**

Do not include any content in the **manifests**/ directory of a plain bundle that are not static manifests. Otherwise, a failure will occur when creating content on-cluster from that bundle. Any file that would not successfully apply with the **oc apply** command will result in an error. Multi-object YAML or JSON files are valid, as well.

### 2.2.3.1.3. Registry bundle spec

A registry bundle, or **registry+v1** bundle, contains a set of static Kubernetes YAML manifests organized in the legacy Operator Lifecycle Manager (OLM) bundle format.

#### Additional resources

Legacy OLM bundle format

# 2.2.3.2. BundleDeployment



#### **WARNING**

A **BundleDeployment** object changes the state of a Kubernetes cluster by installing and removing objects. It is important to verify and trust the content that is being installed and limit access, by using RBAC, to the **BundleDeployment** API to only those who require those permissions.

The RukPak **BundleDeployment** API points to a **Bundle** object and indicates that it should be active. This includes pivoting from older versions of an active bundle. A **BundleDeployment** object might also include an embedded spec for a desired bundle.

Much like pods generate instances of container images, a bundle deployment generates a deployed version of a bundle. A bundle deployment can be seen as a generalization of the pod concept.

The specifics of how a bundle deployment makes changes to a cluster based on a referenced bundle is defined by the provisioner that is configured to watch that bundle deployment.

# Example BundleDeployment object configured to work with the plain provisioner

apiVersion: core.rukpak.io/v1alpha1
kind: BundleDeployment
metadata:
name: my-bundle-deployment
spec:
provisionerClassName: core-rukpak-io-plain
template:
metadata:
labels:
app: my-bundle
spec:
source:
type: image

image:

ref: my-bundle@sha256:xyz123

provisionerClassName: core-rukpak-io-plain

#### 2.2.3.3. Provisioner

A RukPak provisioner is a controller that understands the **BundleDeployment** and **Bundle** APIs and can take action. Each provisioner is assigned a unique ID and is responsible for reconciling **Bundle** and **BundleDeployment** objects with a **spec.provisionerClassName** field that matches that particular ID.

For example, the plain provisioner is able to unpack a given **plain+v0** bundle onto a cluster and then instantiate it, making the content of the bundle available in the cluster.

# 2.3. OPERATOR FRAMEWORK GLOSSARY OF COMMON TERMS

This topic provides a glossary of common terms related to the Operator Framework, including Operator Lifecycle Manager (OLM) and the Operator SDK.

# 2.3.1. Common Operator Framework terms

### 2.3.1.1. Bundle

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

# 2.3.1.2. Bundle image

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

### 2.3.1.3. Catalog source

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

# 2.3.1.4. 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.

### 2.3.1.5. Channel head

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

## 2.3.1.6. 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.

# 2.3.1.7. 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.

## 2.3.1.8. 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.

# 2.3.1.9. Install plan

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

## 2.3.1.10. 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.

# 2.3.1.11. Operator group

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

### 2.3.1.12. Package

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

## 2.3.1.13. Registry

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

# 2.3.1.14. Subscription

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

# 2.3.1.15. Update graph

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

# 2.4. OPERATOR LIFECYCLE MANAGER (OLM)

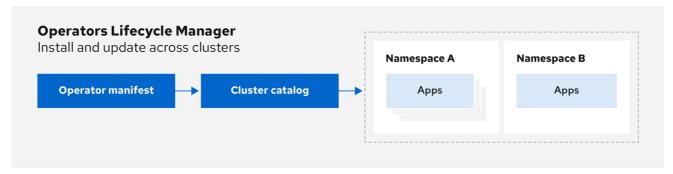
# 2.4.1. Operator Lifecycle Manager concepts and resources

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

# 2.4.1.1. What is Operator Lifecycle Manager?

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

Figure 2.2. Operator Lifecycle Manager workflow



OpenShift 43 1019

OLM runs by default in OpenShift Container Platform 4.13, which aids cluster administrators in installing, upgrading, and granting access to Operators running on their cluster. The OpenShift Container Platform web console provides management screens for cluster administrators to install Operators, as well as grant specific projects access to use the catalog of Operators available on the cluster.

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

### 2.4.1.2. OLM resources

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

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

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

#### 2.4.1.2.1. Cluster service version

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

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

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

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

### 2.4.1.2.2. Catalog source

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

#### TIP

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

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

## Example 2.8. Example CatalogSource object

```
apiVersion: operators.coreos.com/v1alpha1
kind: CatalogSource
metadata:
 generation: 1
 name: example-catalog 1
 namespace: openshift-marketplace 2
 annotations:
  olm.catalogImageTemplate: 3
   "quay.io/example-org/example-catalog:v{kube_major_version}.{kube_minor_version}.
{kube_patch_version}"
spec:
 displayName: Example Catalog 4
 image: quay.io/example-org/example-catalog:v1 5
 priority: -400 6
 publisher: Example Org
 sourceType: grpc 7
 grpcPodConfig:
  securityContextConfig: <security_mode> 8
  nodeSelector: 9
   custom_label: <label>
  priorityClassName: system-cluster-critical 10
  tolerations: 111
   - key: "key1"
    operator: "Equal"
    value: "value1"
    effect: "NoSchedule"
 updateStrategy:
  registryPoll: 12
   interval: 30m0s
status:
 connectionState:
  address: example-catalog.openshift-marketplace.svc:50051
  lastConnect: 2021-08-26T18:14:31Z
  lastObservedState: READY 13
 latestImageRegistryPoll: 2021-08-26T18:46:25Z 14
 registryService: 15
  createdAt: 2021-08-26T16:16:37Z
  port: 50051
  protocol: grpc
  serviceName: example-catalog
  serviceNamespace: openshift-marketplace
```

- Name for the **CatalogSource** object. This value is also used as part of the name for the related pod that is created in the requested namespace.
- Namespace to create the catalog in. To make the catalog available cluster-wide in all namespaces, set this value to **openshift-marketplace**. The default Red Hat-provided catalog sources also use the **openshift-marketplace** namespace. Otherwise, set the value to a specific

namespace to make the Operator only available in that namespace.

Optional: To avoid cluster upgrades potentially leaving Operator installations in an unsupported state or without a continued update path, you can enable automatically changing your Operator catalog's index image version as part of cluster upgrades.

Set the **olm.catalogImageTemplate** annotation to your index image name and use one or more of the Kubernetes cluster version variables as shown when constructing the template for the image tag. The annotation overwrites the **spec.image** field at run time. See the "Image template for custom catalog sources" section for more details.

- Display name for the catalog in the web console and CLI.
- Index image for the catalog. Optionally, can be omitted when using the olm.catalogImageTemplate annotation, which sets the pull spec at run time.
- Weight for the catalog source. OLM uses the weight for prioritization during dependency resolution. A higher weight indicates the catalog is preferred over lower-weighted catalogs.
- Source types include the following:
  - **grpc** with an **image** reference: OLM pulls the image and runs the pod, which is expected to serve a compliant API.
  - **grpc** with an **address** field: OLM attempts to contact the gRPC API at the given address. This should not be used in most cases.
  - **configmap**: OLM parses config map data and runs a pod that can serve the gRPC API over it.
- Specify the value of **legacy** or **restricted**. If the field is not set, the default value is **legacy**. In a future OpenShift Container Platform release, it is planned that the default value will be **restricted**. If your catalog cannot run with **restricted** permissions, it is recommended that you manually set this field to **legacy**.
- Optional: For **grpc** type catalog sources, overrides the default node selector for the pod serving the content in **spec.image**, if defined.
- Optional: For **grpc** type catalog sources, overrides the default priority class name for the pod serving the content in **spec.image**, if defined. Kubernetes provides **system-cluster-critical** and **system-node-critical** priority classes by default. Setting the field to empty ("") assigns the pod the default priority. Other priority classes can be defined manually.
- Optional: For **grpc** type catalog sources, overrides the default tolerations for the pod serving the content in **spec.image**, if defined.
- Automatically check for new versions at a given interval to stay up-to-date.
- Last observed state of the catalog connection. For example:
  - **READY**: A connection is successfully established.
  - **CONNECTING**: A connection is attempting to establish.
  - **TRANSIENT\_FAILURE**: A temporary problem has occurred while attempting to establish a connection, such as a timeout. The state will eventually switch back to **CONNECTING** and try again.

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



Status information for the catalog's Operator Registry service.

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

# Example 2.9. Example Subscription object referencing a catalog source

apiVersion: operators.coreos.com/v1alpha1

kind: Subscription

metadata:

name: example-operator

namespace: example-namespace

spec:

channel: stable

name: example-operator source: example-catalog

sourceNamespace: openshift-marketplace

#### Additional resources

- Understanding OperatorHub
- Red Hat-provided Operator catalogs
- Adding a catalog source to a cluster
- Catalog priority
- Viewing Operator catalog source status by using the CLI
- Understanding and managing pod security admission
- Catalog source pod scheduling

### 2.4.1.2.2.1. Image template for custom catalog sources

Operator compatibility with the underlying cluster can be expressed by a catalog source in various ways. One way, which is used for the default Red Hat-provided catalog sources, is to identify image tags for index images that are specifically created for a particular platform release, for example OpenShift Container Platform 4.13.

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 4.12 to 4.13, the **spec.image** field in the **CatalogSource** object for the **redhatoperators** catalog is updated from:

registry.redhat.io/redhat/redhat-operator-index:v4.12

to:

registry.redhat.io/redhat/redhat-operator-index:v4.13

However, the CVO does not automatically update image tags for custom catalogs. To ensure users are left with a compatible and supported Operator installation after a cluster upgrade, custom catalogs should also be kept updated to reference an updated index image.

Starting in OpenShift Container Platform 4.9, cluster administrators can add the **olm.catalogImageTemplate** annotation in the **CatalogSource** object for custom catalogs to an image reference that includes a template. The following Kubernetes version variables are supported for use in the template:

- kube\_major\_version
- kube\_minor\_version
- kube\_patch\_version



#### NOTE

You must specify the Kubernetes cluster version and not an OpenShift Container Platform cluster version, as the latter is not currently available for templating.

Provided that you have created and pushed an index image with a tag specifying the updated Kubernetes version, setting this annotation enables the index image versions in custom catalogs to be automatically changed after a cluster upgrade. The annotation value is used to set or update the image reference in the **spec.image** field of the **CatalogSource** object. This helps avoid cluster upgrades leaving Operator installations in unsupported states or without a continued update path.



### **IMPORTANT**

You must ensure that the index image with the updated tag, in whichever registry it is stored in, is accessible by the cluster at the time of the cluster upgrade.

## Example 2.10. Example catalog source with an image template

apiVersion: operators.coreos.com/v1alpha1 kind: CatalogSource

metadata: generation: 1

name: example-catalog

namespace: openshift-marketplace

annotations:

olm.catalogImageTemplate:

"quay.io/example-org/example-catalog:v{kube\_major\_version}.{kube\_minor\_version}"

spec:

displayName: Example Catalog

image: quay.io/example-org/example-catalog:v1.26

priority: -400

publisher: Example Org



#### NOTE

If the **spec.image** field and the **olm.catalogImageTemplate** annotation are both set, the **spec.image** field is overwritten by the resolved value from the annotation. If the annotation does not resolve to a usable pull spec, the catalog source falls back to the set **spec.image** value.

If the **spec.image** field is not set and the annotation does not resolve to a usable pull spec, OLM stops reconciliation of the catalog source and sets it into a human-readable error condition.

For an OpenShift Container Platform 4.13 cluster, which uses Kubernetes 1.26, the **olm.catalogImageTemplate** annotation in the preceding example resolves to the following image reference:

н

quay.io/example-org/example-catalog:v1.26

For future releases of OpenShift Container Platform, you can create updated index images for your custom catalogs that target the later Kubernetes version that is used by the later OpenShift Container Platform version. With the **olm.catalogImageTemplate** annotation set before the upgrade, upgrading the cluster to the later OpenShift Container Platform version would then automatically update the catalog's index image as well.

# 2.4.1.2.2.2. Catalog health requirements

Operator catalogs on a cluster are interchangeable from the perspective of installation resolution; a **Subscription** object might reference a specific catalog, but dependencies are resolved using all catalogs on the cluster.

For example, if Catalog A is unhealthy, a subscription referencing Catalog A could resolve a dependency in Catalog B, which the cluster administrator might not have been expecting, because B normally had a lower catalog priority than A.

As a result, OLM requires that all catalogs with a given global namespace (for example, the default **openshift-marketplace** namespace or a custom global namespace) are healthy. When a catalog is unhealthy, all Operator installation or update operations within its shared global namespace will fail with a **CatalogSourcesUnhealthy** condition. If these operations were permitted in an unhealthy state, OLM might make resolution and installation decisions that were unexpected to the cluster administrator.

As a cluster administrator, if you observe an unhealthy catalog and want to consider the catalog as invalid and resume Operator installations, see the "Removing custom catalogs" or "Disabling the default OperatorHub catalog sources" sections for information about removing the unhealthy catalog.

#### Additional resources

- Removing custom catalogs
- Disabling the default OperatorHub catalog sources

### 2.4.1.2.3. Subscription

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

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

# **Example Subscription object**

apiVersion: operators.coreos.com/v1alpha1

kind: Subscription

metadata:

name: example-operator

namespace: example-namespace

spec:

channel: stable

name: example-operator source: example-catalog

sourceNamespace: openshift-marketplace

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

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

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

#### Additional resources

- Multitenancy and Operator colocation
- Viewing Operator subscription status by using the CLI

#### 2.4.1.2.4. Install plan

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

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

The install plan must then be approved according to one of the following approval strategies:

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

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

### Example 2.11. Example InstallPlan object

```
apiVersion: operators.coreos.com/v1alpha1
kind: InstallPlan
metadata:
 name: install-abcde
 namespace: operators
 approval: Automatic
 approved: true
 clusterServiceVersionNames:
  - my-operator.v1.0.1
 generation: 1
status:
 catalogSources: []
 conditions:
  - lastTransitionTime: '2021-01-01T20:17:27Z'
   lastUpdateTime: '2021-01-01T20:17:27Z'
   status: 'True'
   type: Installed
 phase: Complete
 plan:
  - resolving: my-operator.v1.0.1
   resource:
    group: operators.coreos.com
    kind: ClusterServiceVersion
    manifest: >-
    name: my-operator.v1.0.1
    sourceName: redhat-operators
    sourceNamespace: openshift-marketplace
    version: v1alpha1
   status: Created
  - resolving: my-operator.v1.0.1
   resource:
    group: apiextensions.k8s.io
    kind: CustomResourceDefinition
    manifest: >-
    name: webservers.web.servers.org
    sourceName: redhat-operators
    sourceNamespace: openshift-marketplace
    version: v1beta1
   status: Created
  - resolving: my-operator.v1.0.1
   resource:
    group: "
    kind: ServiceAccount
    manifest: >-
    name: my-operator
    sourceName: redhat-operators
```

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

## Additional resources

- Multitenancy and Operator colocation
- Allowing non-cluster administrators to install Operators

# 2.4.1.2.5. Operator groups

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

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

#### Additional resources

Operator groups

### 2.4.1.2.6. Operator conditions

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

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



#### NOTE

By default, the **Spec.Conditions** array is not present in an **OperatorCondition** object until it is either added by a user or as a result of custom Operator logic.

### Additional resources

Operator conditions

# 2.4.2. Operator Lifecycle Manager architecture

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

# 2.4.2.1. Component responsibilities

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

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

Table 2.2. CRDs managed by OLM and Catalog Operators

Resource	Shor t nam e	Own er	Description
ClusterServic eVersion (CSV)	csv	OLM	Application metadata: name, version, icon, required resources, installation, and so on.
InstallPlan	ip	Catal og	Calculated list of resources to be created to automatically install or upgrade a CSV.
CatalogSour ce	cats	Catal og	A repository of CSVs, CRDs, and packages that define an application.
Subscription	sub	Catal og	Used to keep CSVs up to date by tracking a channel in a package.
OperatorGro up	og	OLM	Configures all Operators deployed in the same namespace as the <b>OperatorGroup</b> object to watch for their custom resource (CR) in a list of namespaces or cluster-wide.

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

Table 2.3. Resources created by OLM and Catalog Operators

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

# 2.4.2.2. OLM Operator

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

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

The OLM Operator uses the following workflow:

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



#### NOTE

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

# 2.4.2.3. Catalog Operator

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

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

The Catalog Operator uses the following workflow:

- 1. Connect to each catalog source in the cluster.
- 2. Watch for unresolved install plans created by a user, and if found:
  - Find the CCV metablic the management and add the CCV as a mesh ad a second

- a. Find the CSV matching the name requested and add the CSV as a resolved resource.
- b. For each managed or required CRD, add the CRD as a resolved resource.
- c. For each required CRD, find the CSV that manages it.
- 3. Watch for resolved install plans and create all of the discovered resources for it, if approved by a user or automatically.
- 4. Watch for catalog sources and subscriptions and create install plans based on them.

# 2.4.2.4. Catalog Registry

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

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

# 2.4.3. Operator Lifecycle Manager workflow

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

# 2.4.3.1. Operator installation and upgrade workflow in OLM

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

- ClusterServiceVersion (CSV)
- CatalogSource
- Subscription

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

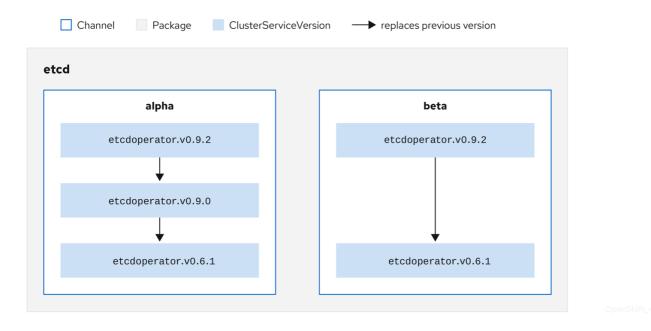
Figure 2.3. Catalog source overview



OpenShift\_43\_1019

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

Figure 2.4. Packages and channels in a Catalog source



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

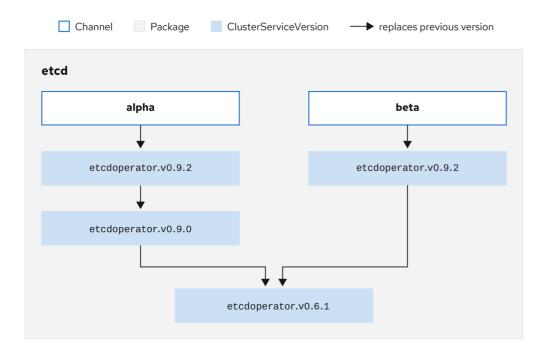


#### NOTE

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

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

Figure 2.5. OLM graph of available channel updates



Example channels in a package

packageName: example

channels:

- name: alpha

currentCSV: example.v0.1.2

- name: beta

currentCSV: example.v0.1.3

defaultChannel: alpha

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

### 2.4.3.1.1. Example upgrade path

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

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

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

### 2.4.3.1.2. Skipping upgrades

The basic path for upgrades in OLM is:

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

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

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

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

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

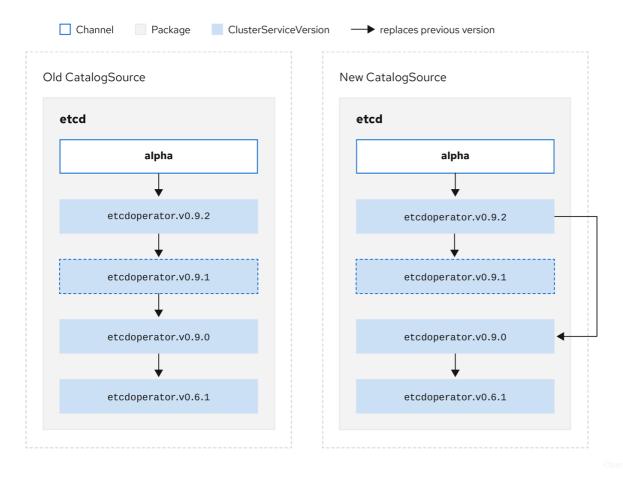
# Example CSV with skipped release

- etcdoperator.v0.9.1

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

Consider the following example of **Old CatalogSource** and **New CatalogSource**.

Figure 2.6. Skipping updates



This graph maintains that:

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

## 2.4.3.1.3. Replacing multiple Operators

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

olm.skipRange: <semver\_range>

where **<semver\_range>** has the version range format supported by the semver library.

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

The order of precedence is:

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

- 3. Channel head in another source that is visible to the subscription, if the other criteria for skipping are met.
- 4. The next Operator that replaces the current one in any source visible to the subscription.

# Example CSV with skipRange

apiVersion: operators.coreos.com/v1alpha1

kind: ClusterServiceVersion

metadata:

name: elasticsearch-operator.v4.1.2

namespace: <namespace>

annotations:

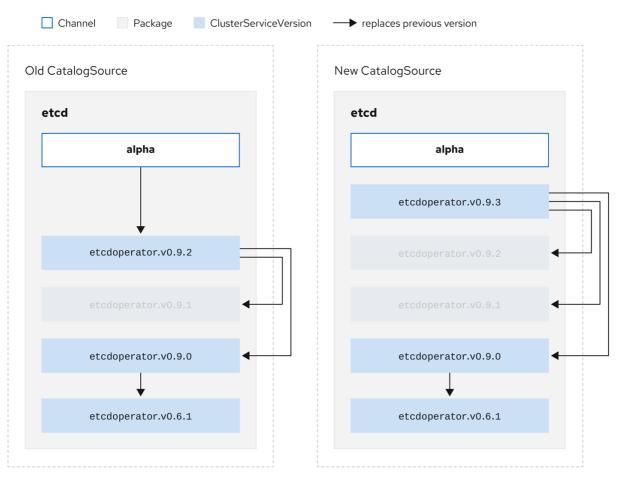
olm.skipRange: '>=4.1.0 <4.1.2'

## 2.4.3.1.4. Z-stream support

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

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

Figure 2.7. Replacing several Operators



OpenShift\_43\_1019

This graph maintains that:

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

# 2.4.4. Operator Lifecycle Manager dependency resolution

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

# 2.4.4.1. About dependency resolution

Operator Lifecycle Manager (OLM) manages the dependency resolution and upgrade lifecycle of running Operators. In many ways, the problems OLM faces are similar to other system or language package managers, such as **yum** and **rpm**.

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

As a result, OLM must never create the following scenarios:

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

This is made possible with two types of data:

Properties	Typed metadata about the Operator that constitutes the public interface for it in the dependency resolver. Examples include the group/version/kind (GVK) of the APIs provided by the Operator and the semantic version (semver) of the Operator.
Constraints or dependencie s	An Operator's requirements that should be satisfied by other Operators that might or might not have already been installed on the target cluster. These act as queries or filters over all available Operators and constrain the selection during dependency resolution and installation. Examples include requiring a specific API to be available on the cluster or expecting a particular Operator with a particular version to be installed.

OLM converts these properties and constraints into a system of Boolean formulas and passes them to a SAT solver, a program that establishes Boolean satisfiability, which does the work of determining what Operators should be installed.

## 2.4.4.2. Operator properties

All Operators in a catalog have the following properties:

### olm.package

Includes the name of the package and the version of the Operator

## olm.gvk

A single property for each provided API from the cluster service version (CSV)

Additional properties can also be directly declared by an Operator author by including a **properties.yaml** file in the **metadata**/ directory of the Operator bundle.

# **Example arbitrary property**

properties:

- type: olm.kubeversion

value:

version: "1.16.0"

## 2.4.4.2.1. Arbitrary properties

Operator authors can declare arbitrary properties in a **properties.yaml** file in the **metadata**/ directory of the Operator bundle. These properties are translated into a map data structure that is used as an input to the Operator Lifecycle Manager (OLM) resolver at runtime.

These properties are opaque to the resolver as it does not understand the properties, but it can evaluate the generic constraints against those properties to determine if the constraints can be satisfied given the properties list.

# **Example arbitrary properties**

properties:

- property:

type: color value: red

property: type: shape value: square

- property:

type: olm.gvk

value:

group: olm.coreos.io version: v1alpha1 kind: myresource

This structure can be used to construct a Common Expression Language (CEL) expression for generic constraints.

# Additional resources

Common Expression Language (CEL) constraints

## 2.4.4.3. Operator dependencies

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

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

## olm.package

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

### olm.gvk

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

## olm.constraint

This type declares generic constraints on arbitrary Operator properties.

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

# Example dependencies.yaml file

### dependencies:

- type: olm.package

value:

packageName: prometheus

version: ">0.27.0" - type: olm.gvk

value:

group: etcd.database.coreos.com

kind: EtcdCluster version: v1beta2

### 2.4.4.4. Generic constraints

An **olm.constraint** property declares a dependency constraint of a particular type, differentiating non-constraint and constraint properties. Its **value** field is an object containing a **failureMessage** field holding a string-representation of the constraint message. This message is surfaced as an informative comment to users if the constraint is not satisfiable at runtime.

The following keys denote the available constraint types:

#### gvk

Type whose value and interpretation is identical to the olm.gvk type

#### package

Type whose value and interpretation is identical to the **olm.package** type

#### cel

A Common Expression Language (CEL) expression evaluated at runtime by the Operator Lifecycle Manager (OLM) resolver over arbitrary bundle properties and cluster information

### all, any, not

Conjunction, disjunction, and negation constraints, respectively, containing one or more concrete constraints, such as **gvk** or a nested compound constraint

#### 2.4.4.4.1. Common Expression Language (CEL) constraints

The **cel** constraint type supports Common Expression Language (CEL) as the expression language. The **cel** struct has a **rule** field which contains the CEL expression string that is evaluated against Operator properties at runtime to determine if the Operator satisfies the constraint.

# Example cel constraint

```
type: olm.constraint
value:
  failureMessage: 'require to have "certified"'
  cel:
    rule: 'properties.exists(p, p.type == "certified")'
```

The CEL syntax supports a wide range of logical operators, such as **AND** and **OR**. As a result, a single CEL expression can have multiple rules for multiple conditions that are linked together by these logical operators. These rules are evaluated against a dataset of multiple different properties from a bundle or any given source, and the output is solved into a single bundle or Operator that satisfies all of those rules within a single constraint.

# Example cel constraint with multiple rules

```
type: olm.constraint
value:
failureMessage: 'require to have "certified" and "stable" properties'
cel:
rule: 'properties.exists(p, p.type == "certified") && properties.exists(p, p.type == "stable")'
```

# 2.4.4.4.2. Compound constraints (all, any, not)

Compound constraint types are evaluated following their logical definitions.

The following is an example of a conjunctive constraint (**all**) of two packages and one GVK. That is, they must all be satisfied by installed bundles:

# Example all constraint

```
schema: olm.bundle
name: red.v1.0.0
properties:
- type: olm.constraint
value:
    failureMessage: All are required for Red because...
    all:
        constraints:
        - failureMessage: Package blue is needed for...
        package:
        name: blue
        versionRange: '>=1.0.0'
        - failureMessage: GVK Green/v1 is needed for...
        gvk:
        group: greens.example.com
        version: v1
        kind: Green
```

The following is an example of a disjunctive constraint (**any**) of three versions of the same GVK. That is, at least one must be satisfied by installed bundles:

### Example any constraint

```
schema: olm.bundle
name: red.v1.0.0
properties:
- type: olm.constraint
 value:
  failureMessage: Any are required for Red because...
  any:
   constraints:
   - gvk:
      group: blues.example.com
      version: v1beta1
      kind: Blue
   - gvk:
      group: blues.example.com
      version: v1beta2
      kind: Blue
   - gvk:
      group: blues.example.com
      version: v1
      kind: Blue
```

The following is an example of a negation constraint (**not**) of one version of a GVK. That is, this GVK cannot be provided by any bundle in the result set:

## Example not constraint

```
schema: olm.bundle
name: red.v1.0.0
properties:
- type: olm.constraint
 value:
 all:
  constraints:
  - failureMessage: Package blue is needed for...
   package:
    name: blue
     versionRange: '>=1.0.0'
  - failureMessage: Cannot be required for Red because...
    constraints:
     - avk:
       group: greens.example.com
       version: v1alpha1
       kind: greens
```

The negation semantics might appear unclear in the **not** constraint context. To clarify, the negation is really instructing the resolver to remove any possible solution that includes a particular GVK, package at a version, or satisfies some child compound constraint from the result set.

As a corollary, the **not** compound constraint should only be used within **all** or **any** constraints, because negating without first selecting a possible set of dependencies does not make sense.

# 2.4.4.4.3. Nested compound constraints

A nested compound constraint, one that contains at least one child compound constraint along with zero or more simple constraints, is evaluated from the bottom up following the procedures for each previously described constraint type.

The following is an example of a disjunction of conjunctions, where one, the other, or both can satisfy the constraint:

# Example nested compound constraint

```
schema: olm.bundle
name: red.v1.0.0
properties:
- type: olm.constraint
  failureMessage: Required for Red because...
  any:
   constraints:
   - all:
      constraints:
      - package:
        name: blue
        versionRange: '>=1.0.0'
      - gvk:
        group: blues.example.com
        version: v1
        kind: Blue
   - all:
      constraints:
      - package:
        name: blue
        versionRange: '<1.0.0'
        group: blues.example.com
        version: v1beta1
        kind: Blue
```



## **NOTE**

The maximum raw size of an **olm.constraint** type is 64KB to limit resource exhaustion attacks.

# 2.4.4.5. Dependency preferences

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

### 2.4.4.5.1. Catalog priority

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

# **Example CatalogSource object**

apiVersion: "operators.coreos.com/v1alpha1"

kind: "CatalogSource"

metadata:

name: "my-operators" namespace: "operators"

spec:

sourceType: grpc grpcPodConfig:

securityContextConfig: <security\_mode> 1 image: example.com/my/operator-index:v1

displayName: "My Operators"

priority: 100

Specify the value of **legacy** or **restricted**. If the field is not set, the default value is **legacy**. In a future OpenShift Container Platform release, it is planned that the default value will be **restricted**. If your catalog cannot run with **restricted** permissions, it is recommended that you manually set this field to **legacy**.

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

There are two rules that govern catalog preference:

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

# 2.4.4.5.2. Channel ordering

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

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

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

## 2.4.4.5.3. Order within a channel

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

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

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

### 2.4.4.5.4. Other constraints

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

## 2.4.4.5.4.1. Subscription constraint

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

## 2.4.4.5.4.2. Package constraint

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

#### 2.4.4.5.5. Additional resources

Catalog health requirements

# 2.4.4.6. CRD upgrades

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

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

## Additional resources

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

## 2.4.4.7. Dependency best practices

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

# Depend on APIs or a specific version range of Operators

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

# Set a minimum version

The Kubernetes documentation on API changes describes what changes are allowed for Kubernetesstyle Operators. These versioning conventions allow an Operator to update an API without bumping the API version, as long as the API is backwards-compatible.

For Operator dependencies, this means that knowing the API version of a dependency might not be enough to ensure the dependent Operator works as intended.

#### For example:

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

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

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

# Omit a maximum version or allow a very wide range

Because Operators provide cluster-scoped resources such as API services and CRDs, an Operator that specifies a small window for a dependency might unnecessarily constrain updates for other consumers of that dependency.

Whenever possible, do not set a maximum version. Alternatively, set a very wide semantic range to prevent conflicts with other Operators. For example, >1.0.0 <2.0.0.

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



#### NOTE

Cluster administrators cannot override dependencies set by an Operator author.

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

#### Additional resources

Kubernetes documentation: Changing the API

### 2.4.4.8. Dependency caveats

When specifying dependencies, there are caveats you should consider.

## No compound constraints (AND)

There is currently no method for specifying an AND relationship between constraints. In other words, there is no way to specify that one Operator depends on another Operator that both provides a given API and has version >1.1.0.

This means that when specifying a dependency such as:

#### dependencies:

- type: olm.package

value:

packageName: etcd version: ">3.1.0"

- type: olm.gvk

value:

group: etcd.database.coreos.com

kind: EtcdCluster version: v1beta2

It would be possible for OLM to satisfy this with two Operators: one that provides EtcdCluster and one that has version >3.1.0. Whether that happens, or whether an Operator is selected that satisfies both constraints, depends on the ordering that potential options are visited. Dependency preferences and ordering options are well-defined and can be reasoned about, but to exercise caution, Operators should stick to one mechanism or the other.

## Cross-namespace compatibility

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

## 2.4.4.9. Example dependency resolution scenarios

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

# **Example: Deprecating dependent APIs**

A and B are APIs (CRDs):

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

This results in:

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

This is a case OLM prevents with its upgrade strategy.

#### **Example: Version deadlock**

A and B are APIs:

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

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

This is another case OLM prevents with its upgrade strategy.

# 2.4.5. Operator groups

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

# 2.4.5.1. About Operator groups

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

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

# 2.4.5.2. Operator group membership

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

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

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

Table 2.4. Install modes and supported Operator groups

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



### NOTE

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

## 2.4.5.3. Target namespace selection

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

apiVersion: operators.coreos.com/v1

kind: OperatorGroup

metadata:

name: my-group

namespace: my-namespace

spec:

targetNamespaces:

- my-namespace

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

apiVersion: operators.coreos.com/v1

kind: OperatorGroup

metadata:

name: my-group

namespace: my-namespace

spec: selector:

cool.io/prod: "true"



#### **IMPORTANT**

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

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

apiVersion: operators.coreos.com/v1

kind: OperatorGroup

metadata:

name: my-group

namespace: my-namespace

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

# 2.4.5.4. Operator group CSV annotations

Member CSVs of an Operator group have the following annotations:

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



### **NOTE**

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

#### 2.4.5.5. Provided APIs annotation

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

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

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

#### 2.4.5.6. Role-based access control

When an Operator group is created, three cluster roles are generated. Each contains a single aggregation rule with a cluster role selector set to match a label, as shown below:

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

The following RBAC resources are generated when a CSV becomes an active member of an Operator group, as long as the CSV is watching all namespaces with the **AllNamespaces** install mode and is not in a failed state with reason **InterOperatorGroupOwnerConflict**:

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

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

Cluster role	Settings
<kind>.<group>-<version>-admin</version></group></kind>	Verbs on <kind>:      *  Aggregation labels:      rbac.authorization.k8s.io/aggregate-to-admin: true      olm.opgroup.permissions/aggregate-to-admin: <operatorgroup_name></operatorgroup_name></kind>
<kind>.<group>-<version>-edit</version></group></kind>	<ul> <li>Verbs on <kind>:</kind></li> <li>create</li> <li>update</li> <li>patch</li> <li>delete</li> </ul> Aggregation labels: <ul> <li>rbac.authorization.k8s.io/aggregate-to-edit: true</li> <li>olm.opgroup.permissions/aggregate-to-edit: <operatorgroup_name></operatorgroup_name></li> </ul>

Cluster role	Settings
<kind>.<group>-<version>-view</version></group></kind>	<ul> <li>Verbs on <kind>:</kind></li> <li>get</li> <li>list</li> <li>watch</li> </ul> Aggregation labels: <ul> <li>rbac.authorization.k8s.io/aggregate-to-view: true</li> <li>olm.opgroup.permissions/aggregate-to-view: <operatorgroup_name></operatorgroup_name></li> </ul>
<kind>.<group>-<version>-view-crdview</version></group></kind>	Verbs on apiextensions.k8s.io customresourcedefinitions <crd-name>:  • get  Aggregation labels:  • rbac.authorization.k8s.io/aggregate-to-view: true  • olm.opgroup.permissions/aggregate-to-view: <operatorgroup_name></operatorgroup_name></crd-name>

Table 2.6. Cluster roles generated for each API resource from an API service

Cluster role	Settings
<kind>.<group>-<version>-admin</version></group></kind>	Verbs on <kind>:</kind>

Cluster role	Settings
<kind>.<group>-<version>-edit</version></group></kind>	Verbs on <kind>:</kind>
<kind>.<group>-<version>-view</version></group></kind>	<ul> <li>Verbs on <kind>:</kind></li> <li>get</li> <li>list</li> <li>watch</li> </ul> Aggregation labels: <ul> <li>rbac.authorization.k8s.io/aggregate-to-view: true</li> <li>olm.opgroup.permissions/aggregate-to-view: <operatorgroup_name></operatorgroup_name></li> </ul>

## Additional roles and role bindings

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

## 2.4.5.7. Copied CSVs

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

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

tenants.

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



#### **NOTE**

By default, the **disableCopiedCSVs** field is disabled. After enabling a **disableCopiedCSVs** field, the OLM deletes existing copied CSVs on a cluster. When a **disableCopiedCSVs** field is disabled, the OLM adds copied CSVs again.

• Disable the disableCopiedCSVs field:

```
$ cat << EOF | oc apply -f -
apiVersion: operators.coreos.com/v1
kind: OLMConfig
metadata:
name: cluster
spec:
features:
disableCopiedCSVs: false
EOF
```

• Enable the disableCopiedCSVs field:

```
$ cat << EOF | oc apply -f -
apiVersion: operators.coreos.com/v1
kind: OLMConfig
metadata:
name: cluster
spec:
features:
disableCopiedCSVs: true
EOF
```

### 2.4.5.8. Static Operator groups

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

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

```
apiVersion: operators.coreos.com/v1 kind: OperatorGroup metadata: name: cluster-monitoring namespace: cluster-monitoring annotations: olm.providedAPIs:
```

Alertmanager.v1.monitoring.coreos.com,Prometheus.v1.monitoring.coreos.com,PrometheusRule.v1.monitoring.coreos.com,ServiceMonitor.v1.monitoring.coreos.com

spec:
 staticProvidedAPIs: true
 selector:
 matchLabels:
 something.cool.io/cluster-monitoring: "true"

# 2.4.5.9. Operator group intersection

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

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



#### NOTE

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

### Rules for intersection

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

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

- If the Operator group is *not* static:
  - Replace the Operator group's olm.providedAPIs annotation with the difference between itself and the CSV's provided APIs.



#### NOTE

Failure states caused by Operator groups are non-terminal.

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

- The set of provided APIs from active member CSVs is calculated from the cluster. Note that copied CSVs are ignored.
- The cluster set is compared to **olm.providedAPIs**, and if **olm.providedAPIs** contains any extra APIs, then those APIs are pruned.
- All CSVs that provide the same APIs across all namespaces are requeued. This notifies
  conflicting CSVs in intersecting groups that their conflict has possibly been resolved, either
  through resizing or through deletion of the conflicting CSV.

# 2.4.5.10. Limitations for multitenant Operator management

OpenShift Container Platform provides limited support for simultaneously installing different versions of an Operator on the same cluster. Operator Lifecycle Manager (OLM) installs Operators multiple times in different namespaces. One constraint of this is that the Operator's API versions must be the same.

Operators are control plane extensions due to their usage of **CustomResourceDefinition** objects (CRDs), which are global resources in Kubernetes. Different major versions of an Operator often have incompatible CRDs. This makes them incompatible to install simultaneously in different namespaces on a cluster.

All tenants, or namespaces, share the same control plane of a cluster. Therefore, tenants in a multitenant cluster also share global CRDs, which limits the scenarios in which different instances of the same Operator can be used in parallel on the same cluster.

The supported scenarios include the following:

- Operators of different versions that ship the exact same CRD definition (in case of versioned CRDs, the exact same set of versions)
- Operators of different versions that do not ship a CRD, and instead have their CRD available in a separate bundle on the OperatorHub

All other scenarios are not supported, because the integrity of the cluster data cannot be guaranteed if there are multiple competing or overlapping CRDs from different Operator versions to be reconciled on the same cluster.

#### Additional resources

- Operator Lifecycle Manager (OLM) → Multitenancy and Operator colocation
- Operators in multitenant clusters
- Allowing non-cluster administrators to install Operators

# 2.4.5.11. Troubleshooting Operator groups

### Membership

- An install plan's namespace must contain only one Operator group. When attempting to generate a cluster service version (CSV) in a namespace, an install plan considers an Operator group invalid in the following scenarios:
  - No Operator groups exist in the install plan's namespace.
  - Multiple Operator groups exist in the install plan's namespace.
  - An incorrect or non-existent service account name is specified in the Operator group.

If an install plan encounters an invalid Operator group, the CSV is not generated and the **InstallPlan** resource continues to install with a relevant message. For example, the following message is provided if more than one Operator group exists in the same namespace:

attenuated service account query failed - more than one operator group(s) are managing this namespace count=2

where **count=** specifies the number of Operator groups in the namespace.

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

# 2.4.6. Multitenancy and Operator colocation

This guide outlines multitenancy and Operator colocation in Operator Lifecycle Manager (OLM).

### 2.4.6.1. Colocation of Operators in a namespace

Operator Lifecycle Manager (OLM) handles OLM-managed Operators that are installed in the same namespace, meaning their **Subscription** resources are colocated in the same namespace, as related Operators. Even if they are not actually related, OLM considers their states, such as their version and update policy, when any one of them is updated.

This default behavior manifests in two ways:

- **InstallPlan** resources of pending updates include **ClusterServiceVersion** (CSV) resources of all other Operators that are in the same namespace.
- All Operators in the same namespace share the same update policy. For example, if one Operator is set to manual updates, all other Operators' update policies are also set to manual.

These scenarios can lead to the following issues:

- It becomes hard to reason about install plans for Operator updates, because there are many more resources defined in them than just the updated Operator.
- It becomes impossible to have some Operators in a namespace update automatically while other are updated manually, which is a common desire for cluster administrators.

These issues usually surface because, when installing Operators with the OpenShift Container Platform web console, the default behavior installs Operators that support the **All namespaces** install mode into the default **openshift-operators** global namespace.

As a cluster administrator, you can bypass this default behavior manually by using the following workflow:

- 1. Create a namespace for the installation of the Operator.
- 2. Create a custom *global Operator group*, which is an Operator group that watches all namespaces. By associating this Operator group with the namespace you just created, it makes the installation namespace a global namespace, which makes Operators installed there available in all namespaces.
- 3. Install the desired Operator in the installation namespace.

If the Operator has dependencies, the dependencies are automatically installed in the pre-created namespace. As a result, it is then valid for the dependency Operators to have the same update policy and shared install plans. For a detailed procedure, see "Installing global Operators in custom namespaces".

#### Additional resources

- Installing global Operators in custom namespaces
- Operators in multitenant clusters

# 2.4.7. Operator conditions

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

## 2.4.7.1. About Operator conditions

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

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



# **NOTE**

By default, the **Spec.Conditions** array is not present in an **OperatorCondition** object until it is either added by a user or as a result of custom Operator logic.

## 2.4.7.2. Supported conditions

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

#### 2.4.7.2.1. Upgradeable condition

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

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



#### **IMPORTANT**

Setting the **Upgradeable** Operator condition to the **False** value does not avoid pod disruption. If you must ensure your pods are not disrupted, see "Using pod disruption budgets to specify the number of pods that must be up" and "Graceful termination" in the "Additional resources" section.

# **Example Upgradeable Operator condition**

apiVersion: operators.coreos.com/v1

kind: OperatorCondition

metadata:

name: my-operator namespace: operators

spec:

conditions:

- type: Upgradeable 1 status: "False" 2 reason: "migration"

message: "The Operator is performing a migration."

lastTransitionTime: "2020-08-24T23:15:55Z"

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

#### 2.4.7.3. Additional resources

- Managing Operator conditions
- Enabling Operator conditions
- Using pod disruption budgets to specify the number of pods that must be up
- Graceful termination

# 2.4.8. Operator Lifecycle Manager metrics

### 2.4.8.1. Exposed metrics

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

Table 2.7. Metrics exposed by OLM

Name	Description
catalog_source _count	Number of catalog sources.
catalogsource_r eady	State of a catalog source. The value <b>1</b> indicates that the catalog source is in a <b>READY</b> state. The value of <b>0</b> indicates that the catalog source is not in a <b>READY</b> state.
csv_abnormal	When reconciling a cluster service version (CSV), present whenever a CSV version is in any state other than <b>Succeeded</b> , for example when it is not installed. Includes the <b>name</b> , <b>namespace</b> , <b>phase</b> , <b>reason</b> , and <b>version</b> labels. A Prometheus alert is created when this metric is present.
csv_count	Number of CSVs successfully registered.
csv_succeeded	When reconciling a CSV, represents whether a CSV version is in a <b>Succeeded</b> state (value <b>1</b> ) or not (value <b>0</b> ). Includes the <b>name</b> , <b>namespace</b> , and <b>version</b> labels.
csv_upgrade_c ount	Monotonic count of CSV upgrades.
install_plan_co unt	Number of install plans.
installplan_war nings_total	Monotonic count of warnings generated by resources, such as deprecated resources, included in an install plan.
olm_resolution_ duration_secon ds	The duration of a dependency resolution attempt.
subscription_co unt	Number of subscriptions.
subscription_sy nc_total	Monotonic count of subscription syncs. Includes the <b>channel</b> , <b>installed</b> CSV, and subscription <b>name</b> labels.

# 2.4.9. Webhook management in Operator Lifecycle Manager

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

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

## 2.4.9.1. Additional resources

• Types of webhook admission plugins

- Kubernetes documentation:
  - Validating admission webhooks
  - Mutating admission webhooks
  - Conversion webhooks

## 2.5. UNDERSTANDING OPERATORHUB

# 2.5.1. About Operator Hub

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

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

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

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

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

# 2.5.2. Operator Hub architecture

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

# 2.5.2.1. Operator Hub custom resource

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

# Example OperatorHub custom resource

- disableAllDefaultSources is an override that controls availability of all default catalogs that are configured by default during an OpenShift Container Platform installation.
- Disable default catalogs individually by changing the **disabled** parameter value per source.

# 2.5.3. Additional resources

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

## 2.6. RED HAT-PROVIDED OPERATOR CATALOGS

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



## **IMPORTANT**

As of OpenShift Container Platform 4.11, the default Red Hat-provided Operator catalog releases in the file-based catalog format. The default Red Hat-provided Operator catalogs for OpenShift Container Platform 4.6 through 4.10 released in the deprecated SQLite database format.

The **opm** subcommands, flags, and functionality related to the SQLite database format are also deprecated and will be removed in a future release. The features are still supported and must be used for catalogs that use the deprecated SQLite database format.

Many of the **opm** subcommands and flags for working with the SQLite database format, such as **opm index prune**, do not work with the file-based catalog format. For more information about working with file-based catalogs, see Managing custom catalogs, Operator Framework packaging format, and Mirroring images for a disconnected installation using the oc-mirror plugin.

# 2.6.1. About Operator catalogs

An Operator catalog is a repository of metadata that Operator Lifecycle Manager (OLM) can query to discover and install Operators and their dependencies on a cluster. OLM always installs Operators from the latest version of a catalog.

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

As catalogs are updated, the latest versions of Operators change, and older versions may be removed or altered. In addition, when OLM runs on an OpenShift Container Platform cluster in a restricted network environment, it is unable to access the catalogs directly from the internet to pull the latest content.

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



#### **IMPORTANT**

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.



#### NOTE

Support for the legacy *package manifest format* for Operators, including custom catalogs that were using the legacy format, is removed in OpenShift Container Platform 4.8 and later.

When creating custom catalog images, previous versions of OpenShift Container Platform 4 required using the **oc adm catalog build** command, which was deprecated for several releases and is now removed. With the availability of Red Hat-provided index images starting in OpenShift Container Platform 4.6, catalog builders must use the **opm index** command to manage index images.

#### Additional resources

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

# 2.6.2. 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:v4.13	Red Hat products packaged and shipped by Red Hat. Supported by Red Hat.
certified- operators	registry.redhat.io/redhat/certified-operator-index:v4.13	Products from leading independent software vendors (ISVs). Red Hat partners with ISVs to package and ship. Supported by the ISV.
redhat- marketplace	registry.redhat.io/redhat/redhat-marketplace-index:v4.13	Certified software that can be purchased from Red Hat Marketplace.
community- operators	registry.redhat.io/redhat/community-operator-index:v4.13	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

## 2.7. OPERATORS IN MULTITENANT CLUSTERS

The default behavior for Operator Lifecycle Manager (OLM) aims to provide simplicity during Operator installation. However, this behavior can lack flexibility, especially in multitenant clusters. In order for multiple tenants on a OpenShift Container Platform cluster to use an Operator, the default behavior of OLM requires that administrators install the Operator in **All namespaces** mode, which can be considered to violate the principle of least privilege.

Consider the following scenarios to determine which Operator installation workflow works best for your environment and requirements.

#### Additional resources

- Common terms: Multitenant
- Limitations for multitenant Operator management

## 2.7.1. Default Operator install modes and behavior

When installing Operators with the web console as an administrator, you typically have two choices for the install mode, depending on the Operator's capabilities:

# Single namespace

Installs the Operator in the chosen single namespace, and makes all permissions that the Operator requests available in that namespace.

#### All namespaces

Installs the Operator in the default **openshift-operators** namespace to watch and be made available to all namespaces in the cluster. Makes all permissions that the Operator requests available in all namespaces. In some cases, an Operator author can define metadata to give the user a second option for that Operator's suggested namespace.

This choice also means that users in the affected namespaces get access to the Operators APIs, which can leverage the custom resources (CRs) they own, depending on their role in the namespace:

- The **namespace-admin** and **namespace-edit** roles can read/write to the Operator APIs, meaning they can use them.
- The **namespace-view** role can read CR objects of that Operator.

For **Single namespace** mode, because the Operator itself installs in the chosen namespace, its pod and service account are also located there. For **All namespaces** mode, the Operator's privileges are all automatically elevated to cluster roles, meaning the Operator has those permissions in all namespaces.

#### Additional resources

- Adding Operators to a cluster
- Install modes types
- Setting a suggested namespace

## 2.7.2. Recommended solution for multitenant clusters

While a **Multinamespace** install mode does exist, it is supported by very few Operators. As a middle ground solution between the standard **All namespaces** and **Single namespace** install modes, you can install multiple instances of the same Operator, one for each tenant, by using the following workflow:

- 1. Create a namespace for the tenant Operator that is separate from the tenant's namespace.
- 2. Create an Operator group for the tenant Operator scoped only to the tenant's namespace.
- 3. Install the Operator in the tenant Operator namespace.

As a result, the Operator resides in the tenant Operator namespace and watches the tenant namespace, but neither the Operator's pod nor its service account are visible or usable by the tenant.

This solution provides better tenant separation, least privilege principle at the cost of resource usage, and additional orchestration to ensure the constraints are met. For a detailed procedure, see "Preparing for multiple instances of an Operator for multitenant clusters".

### Limitations and considerations

This solution only works when the following constraints are met:

- All instances of the same Operator must be the same version.
- The Operator cannot have dependencies on other Operators.
- The Operator cannot ship a CRD conversion webhook.



### **IMPORTANT**

You cannot use different versions of the same Operator on the same cluster. Eventually, the installation of another instance of the Operator would be blocked when it meets the following conditions:

- The instance is not the newest version of the Operator.
- The instance ships an older revision of the CRDs that lack information or versions that newer revisions have that are already in use on the cluster.



## **WARNING**

As an administrator, use caution when allowing non-cluster administrators to install Operators self-sufficiently, as explained in "Allowing non-cluster administrators to install Operators". These tenants should only have access to a curated catalog of Operators that are known to not have dependencies. These tenants must also be forced to use the same version line of an Operator, to ensure the CRDs do not change. This requires the use of namespace-scoped catalogs and likely disabling the global default catalogs.

#### Additional resources

- Preparing for multiple instances of an Operator for multitenant clusters
- Allowing non-cluster administrators to install Operators
- Disabling the default OperatorHub catalog sources

# 2.7.3. Operator colocation and Operator groups

Operator Lifecycle Manager (OLM) handles OLM-managed Operators that are installed in the same namespace, meaning their **Subscription** resources are colocated in the same namespace, as related Operators. Even if they are not actually related, OLM considers their states, such as their version and update policy, when any one of them is updated.

For more information on Operator colocation and using Operator groups effectively, see Operator Lifecycle Manager (OLM) → Multitenancy and Operator colocation.

# 2.8. CRDS

# 2.8.1. Extending the Kubernetes API with custom resource definitions

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

#### 2.8.1.1. Custom resource definitions

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

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

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

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

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

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



#### NOTE

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

# 2.8.1.2. Creating a custom resource definition

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

### **Prerequisites**

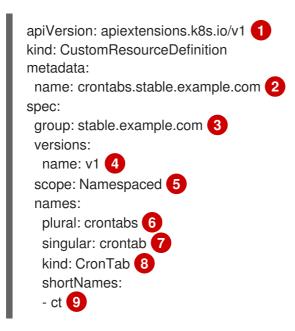
• Access to an OpenShift Container Platform cluster with **cluster-admin** user privileges.

#### Procedure

To create a CRD:

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

# Example YAML file for a CRD



Use the apiextensions.k8s.io/v1 API.



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

- Specify a group name for the API. An API group is a collection of objects that are logically related. For example, all batch objects like **Job** or **ScheduledJob** could be in the batch API group (such as **batch.api.example.com**). A good practice is to use a fully-qualified-domain name (FQDN) of your organization.
- Specify a version name to be used in the URL. Each API group can exist in multiple versions, for example **v1alpha**, **v1beta**, **v1**.
- Specify whether the custom objects are available to a project (**Namespaced**) or all projects in the cluster (**Cluster**).
- 6 Specify the plural name to use in the URL. The **plural** field is the same as a resource in an API URL.
- Specify a singular name to use as an alias on the CLI and for display.
- Specify the kind of objects that can be created. The type can be in CamelCase.
- Specify a shorter string to match your resource on the CLI.



### **NOTE**

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

### 2. Create the CRD object:

\$ oc create -f <file\_name>.yaml

A new RESTful API endpoint is created at:

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

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

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

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

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

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



### **IMPORTANT**

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

### **Prerequisites**

• Create a CRD.

#### **Procedure**

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

# Example YAML file for a cluster role definition

```
kind: ClusterRole
apiVersion: rbac.authorization.k8s.io/v1 1
metadata:
 name: aggregate-cron-tabs-admin-edit 2
  rbac.authorization.k8s.io/aggregate-to-admin: "true" (3)
  rbac.authorization.k8s.io/aggregate-to-edit: "true" 4
- apiGroups: ["stable.example.com"] 5
 resources: ["crontabs"] 6
 verbs: ["get", "list", "watch", "create", "update", "patch", "delete", "deletecollection"]
kind: ClusterRole
apiVersion: rbac.authorization.k8s.io/v1
metadata:
 name: aggregate-cron-tabs-view 8
 labels:
  # Add these permissions to the "view" default role.
  rbac.authorization.k8s.io/aggregate-to-view: "true" 9
  rbac.authorization.k8s.io/aggregate-to-cluster-reader: "true" 10
rules:
- apiGroups: ["stable.example.com"] 111
 resources: ["crontabs"] 12
 verbs: ["get", "list", "watch"] 13
```

- Use the rbac.authorization.k8s.io/v1 API.
- 28 Specify a name for the definition.
- 3 Specify this label to grant permissions to the admin default role.
- A Specify this label to grant permissions to the edit default role.

- 5 11 Specify the group name of the CRD.
- 6.12 Specify the plural name of the CRD that these rules apply to.
- 7 13 Specify the verbs that represent the permissions that are granted to the role. For example, apply read and write permissions to the **admin** and **edit** roles and only read permission to the **view** role.
- Specify this label to grant permissions to the **view** default role.
- Specify this label to grant permissions to the **cluster-reader** default role.
- 2. Create the cluster role:

\$ oc create -f <file\_name>.yaml

# 2.8.1.4. Creating custom resources from a file

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

### **Prerequisites**

• CRD added to the cluster by a cluster administrator.

#### **Procedure**

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

### Example YAML file for a CR

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

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

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

\$ oc create -f <file\_name>.yaml

# 2.8.1.5. Inspecting custom resources

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

# **Prerequisites**

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

#### **Procedure**

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

\$ oc get <kind>

For example:

\$ oc get crontab

# Example output

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

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

\$ oc get crontabs

\$ oc get crontab

\$ oc get ct

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

\$ oc get <kind> -o yaml

For example:

\$ oc get ct -o yaml

# Example output

apiVersion: v1

items:

- apiVersion: stable.example.com/v1

kind: CronTab metadata:

clusterName: ""

creationTimestamp: 2017-05-31T12:56:35Z

deletionGracePeriodSeconds: null

deletionTimestamp: null name: my-new-cron-object

namespace: default resourceVersion: "285"

selfLink: /apis/stable.example.com/v1/namespaces/default/crontabs/my-new-cron-object

uid: 9423255b-4600-11e7-af6a-28d2447dc82b

spec:

cronSpec: '\* \* \* \* /5' 1

image: my-awesome-cron-image 2

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

# 2.8.2. Managing resources from custom resource definitions

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

#### 2.8.2.1. Custom resource definitions

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

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

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

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



#### NOTE

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

## 2.8.2.2. Creating custom resources from a file

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

### **Prerequisites**

• CRD added to the cluster by a cluster administrator.

#### Procedure

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

# Example YAML file for a CR

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

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

\$ oc create -f <file\_name>.yaml

# 2.8.2.3. Inspecting custom resources

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

## **Prerequisites**

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

## Procedure

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

\$ oc get <kind>

For example:

\$ oc get crontab

# **Example output**

NAME KIND

my-new-cron-object CronTab.v1.stable.example.com

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

\$ oc get crontabs

\$ oc get crontab

\$ oc get ct

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

\$ oc get <kind> -o yaml

For example:

\$ oc get ct -o yaml

# **Example output**

apiVersion: v1

items:

- apiVersion: stable.example.com/v1

kind: CronTab metadata: clusterName: ""

creationTimestamp: 2017-05-31T12:56:35Z

deletionGracePeriodSeconds: null

deletionTimestamp: null name: my-new-cron-object namespace: default resourceVersion: "285"

selfLink: /apis/stable.example.com/v1/namespaces/default/crontabs/my-new-cron-object

uid: 9423255b-4600-11e7-af6a-28d2447dc82b

spec:

cronSpec: '\* \* \* \* /5'

image: my-awesome-cron-image 2

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

# **CHAPTER 3. USER TASKS**

### 3.1. CREATING APPLICATIONS FROM INSTALLED OPERATORS

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

# 3.1.1. Creating an etcd cluster using an Operator

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

# **Prerequisites**

- Access to an OpenShift Container Platform 4.13 cluster.
- The etcd Operator already installed cluster-wide by an administrator.

#### **Procedure**

- 1. Create a new project in the OpenShift Container Platform web console for this procedure. This example uses a project called **my-etcd**.
- Navigate to the Operators → Installed Operators page. The Operators that have been installed
  to the cluster by the cluster administrator and are available for use are shown here as a list of
  cluster service versions (CSVs). CSVs are used to launch and manage the software provided by
  the Operator.

## TIP

You can get this list from the CLI using:

- \$ oc get csv
- 3. On the **Installed Operators** page, click the etcd Operator to view more details and available actions.

As shown under **Provided APIs**, this Operator makes available three new resource types, including one for an **etcd Cluster** (the **EtcdCluster** resource). These objects work similar to the built-in native Kubernetes ones, such as **Deployment** or **ReplicaSet**, but contain logic specific to managing etcd.

- 4. Create a new etcd cluster:
  - a. In the etcd Cluster API box, click Create instance.
  - b. The next page allows you to make any modifications to the minimal starting template of an **EtcdCluster** object, such as the size of the cluster. For now, click **Create** to finalize. This triggers the Operator to start up the pods, services, and other components of the new etcd cluster.
- 5. Click the **example** etcd cluster, then click the **Resources** tab to see that your project now contains a number of resources created and configured automatically by the Operator.

Verify that a Kubernetes service has been created that allows you to access the database from other pods in your project.

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

\$ oc policy add-role-to-user edit <user> -n <target\_project>

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

# 3.2. INSTALLING OPERATORS IN YOUR NAMESPACE

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

# 3.2.1. Prerequisites

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

# 3.2.2. About Operator installation with Operator Hub

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

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

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

### Installation Mode

Choose a specific namespace in which to install the Operator.

### **Update Channel**

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

### **Approval Strategy**

You can choose automatic or manual updates.

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

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

Understanding OperatorHub

# 3.2.3. Installing from OperatorHub using the web console

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

### **Prerequisites**

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

#### **Procedure**

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



#### NOTE

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

- 4. Read the information about the Operator and click Install.
- 5. On the **Install Operator** page:
  - a. Choose a specific, single namespace in which to install the Operator. The Operator will only watch and be made available for use in this single namespace.
  - b. Select an **Update channel** (if more than one is available).
  - c. Select **Automatic** or **Manual** approval strategy, as described earlier.
- 6. Click **Install** to make the Operator available to the selected namespaces on this OpenShift Container Platform cluster.
  - a. If you selected a Manual approval strategy, the upgrade status of the subscription remains
     Upgrading until you review and approve the install plan.

     After approving on the Install Plan page, the subscription upgrade status moves to Up to
     date.
  - b. If you selected an **Automatic** approval strategy, the upgrade status should resolve to **Up to** date without intervention.
- 7. After the upgrade status of the subscription is Up to date, select Operators → Installed Operators to verify that the cluster service version (CSV) of the installed Operator eventually shows up. The Status should ultimately resolve to InstallSucceeded in the relevant namespace.



### **NOTE**

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

#### If it does not:

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

# 3.2.4. Installing from OperatorHub using the CLI

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

## **Prerequisites**

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

#### **Procedure**

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

\$ oc get packagemanifests -n openshift-marketplace

## **Example output**

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

Note the catalog for your desired Operator.

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

\$ oc describe packagemanifests < operator\_name > -n openshift-marketplace

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

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

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



### **NOTE**

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

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

## **Example Operator Group object**

apiVersion: operators.coreos.com/v1

kind: OperatorGroup

metadata:

name: <operatorgroup\_name>
namespace: <namespace>

spec:

targetNamespaces:
- <namespace>

b. Create the **OperatorGroup** object:

\$ oc apply -f operatorgroup.yaml

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

### **Example Subscription object**

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

```
envFrom: 7
- secretRef:
  name: license-secret
volumes: 8
- name: <volume_name>
 configMap:
  name: <configmap_name>
volumeMounts: 9
- mountPath: <directory_name>
 name: <volume_name>
tolerations: 10
- operator: "Exists"
resources: 111
 requests:
  memory: "64Mi"
  cpu: "250m"
 limits:
  memory: "128Mi"
  cpu: "500m"
nodeSelector: 12
 foo: bar
```

- For default **AllNamespaces** install mode usage, specify the **openshift-operators** namespace. Alternatively, you can specify a custom global namespace, if you have created one. Otherwise, specify the relevant single namespace for **SingleNamespace** install mode usage.
- Name of the channel to subscribe to.
- 3 Name of the Operator to subscribe to.
- Name of the catalog source that provides the Operator.
- Namespace of the catalog source. Use **openshift-marketplace** for the default OperatorHub catalog sources.
- The **env** parameter defines a list of Environment Variables that must exist in all containers in the pod created by OLM.
- 7 The **envFrom** parameter defines a list of sources to populate Environment Variables in the container.
- The **volumes** parameter defines a list of Volumes that must exist on the pod created by OLM.
- The **volumeMounts** parameter defines a list of VolumeMounts that must exist in all containers in the pod created by OLM. If a **volumeMount** references a **volume** that does not exist, OLM fails to deploy the Operator.
- The **tolerations** parameter defines a list of Tolerations for the pod created by OLM.
- The **resources** parameter defines resource constraints for all the containers in the pod created by OLM.
- The **nodeSelector** parameter defines a **NodeSelector** for the pod created by OLM.

5. Create the **Subscription** object:

\$ oc apply -f sub.yaml

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

#### Additional resources

- Operator groups
- Channel names

# 3.2.5. Installing a specific version of an Operator

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

## **Prerequisites**

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

#### **Procedure**

1. Look up the available versions and channels of the Operator you want to install by running the following command:

### **Command syntax**

\$ oc describe packagemanifests < operator\_name > -n < catalog\_namespace >

For example, the following command prints the available channels and versions of the Red Hat Quay Operator from OperatorHub:

## Example command

\$ oc describe packagemanifests quay-operator -n openshift-marketplace

## Example 3.1. Example output

Name: quay-operator

Namespace: operator-marketplace Labels: catalog=redhat-operators

catalog-namespace=openshift-marketplace

hypershift.openshift.io/managed=true

operatorframework.io/arch.amd64=supported operatorframework.io/os.linux=supported

provider=Red Hat

provider-url=

Annotations: <none>

API Version: packages.operators.coreos.com/v1

Kind: PackageManifest

. . .

Current CSV: quay-operator.v3.7.11

• • •

Entries:

Name: quay-operator.v3.7.11

Version: 3.7.11

Name: quay-operator.v3.7.10

Version: 3.7.10

Name: quay-operator.v3.7.9

Version: 3.7.9

Name: quay-operator.v3.7.8

Version: 3.7.8

Name: quay-operator.v3.7.7

Version: 3.7.7

Name: quay-operator.v3.7.6

Version: 3.7.6

Name: quay-operator.v3.7.5

Version: 3.7.5

Name: quay-operator.v3.7.4

Version: 3.7.4

Name: quay-operator.v3.7.3

Version: 3.7.3

Name: quay-operator.v3.7.2

Version: 3.7.2

Name: quay-operator.v3.7.1

Version: 3.7.1

Name: quay-operator.v3.7.0

Version: 3.7.0 Name: stable-3.7

. .

Current CSV: quay-operator.v3.8.5

٠.

Entries:

Name: quay-operator.v3.8.5

Version: 3.8.5

Name: quay-operator.v3.8.4

Version: 3.8.4

Name: quay-operator.v3.8.3

Version: 3.8.3

Name: quay-operator.v3.8.2

Version: 3.8.2

Name: quay-operator.v3.8.1

Version: 3.8.1

Name: quay-operator.v3.8.0

Version: 3.8.0
Name: stable-3.8
Default Channel: stable-3.8
Package Name: quay-operator

### TIP

You can print an Operator's version and channel information in the YAML format by running the following command:

\$ oc get packagemanifests < operator\_name > -n < catalog\_namespace > -o yaml

• If more than one catalog is installed in a namespace, run the following command to look up the available versions and channels of an Operator from a specific catalog:

\$ oc get packagemanifest \

- --selector=catalog=<catalogsource name> \
- --field-selector metadata.name=<operator\_name> \
- -n <catalog namespace> -o yaml



#### **IMPORTANT**

If you do not specify the Operator's catalog, running the **oc get packagemanifest** and **oc describe packagemanifest** commands might return a package from an unexpected catalog if the following conditions are met:

- Multiple catalogs are installed in the same namespace.
- The catalogs contain the same Operators or Operators with the same name.
- 2. An Operator group, defined by an **OperatorGroup** object, selects target namespaces in which to generate required role-based access control (RBAC) access for all Operators in the same namespace as the Operator group.

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

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

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

## **Example Operator Group object**

apiVersion: operators.coreos.com/v1

kind: OperatorGroup

metadata:

name: <operatorgroup\_name> namespace: <namespace>

spec:

targetNamespaces:

- <namespace>

b. Create the **OperatorGroup** object:

\$ oc apply -f operatorgroup.yaml

3. Create a **Subscription** object YAML file that subscribes a namespace to an Operator with a specific version by setting the **startingCSV** field. Set the **installPlanApproval** field to **Manual** to prevent the Operator from automatically upgrading if a later version exists in the catalog. For example, the following **sub.yaml** file can be used to install the Red Hat Quay Operator specifically to version 3.7.10:

# Subscription with a specific starting Operator version

apiVersion: operators.coreos.com/v1alpha1

kind: Subscription

metadata:

name: quay-operator namespace: quay

spec:

channel: quay-operator.v3.7.10 installPlanApproval: Manual 1

name: quay-operator source: redhat-operators

sourceNamespace: openshift-marketplace startingCSV: quay-operator.v3.7.10 2

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

\$ oc apply -f sub.yaml

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

#### Additional resources

• Manually approving a pending Operator update

# **CHAPTER 4. ADMINISTRATOR TASKS**

# 4.1. ADDING OPERATORS TO A CLUSTER

Using Operator Lifecycle Manager (OLM), cluster administrators can install OLM-based Operators to an OpenShift Container Platform cluster.



#### NOTE

For information on how OLM handles updates for installed Operators colocated in the same namespace, as well as an alternative method for installing Operators with custom global Operator groups, see Multitenancy and Operator colocation.

# 4.1.1. About Operator installation with Operator Hub

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

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

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

#### Installation Mode

Choose a specific namespace in which to install the Operator.

#### **Update Channel**

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

# **Approval Strategy**

You can choose automatic or manual updates.

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

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

#### Additional resources

Understanding OperatorHub

# 4.1.2. Installing from OperatorHub using the web console

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

### Prerequisites

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

#### **Procedure**

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



#### **NOTE**

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

- 4. Read the information about the Operator and click Install.
- 5. On the **Install Operator** page:
  - a. Select one of the following:
    - All namespaces on the cluster (default)installs the Operator in the default openshiftoperators namespace to watch and be made available to all namespaces in the cluster. This option is not always available.
    - A specific namespace on the clusterallows you to choose a specific, single namespace in which to install the Operator. The Operator will only watch and be made available for use in this single namespace.
  - b. Choose a specific, single namespace in which to install the Operator. The Operator will only watch and be made available for use in this single namespace.
  - c. Select an **Update channel** (if more than one is available).
  - d. Select **Automatic** or **Manual** approval strategy, as described earlier.
- 6. Click **Install** to make the Operator available to the selected namespaces on this OpenShift Container Platform cluster.
  - a. If you selected a Manual approval strategy, the upgrade status of the subscription remains
     Upgrading until you review and approve the install plan.

     After approving on the Install Plan page, the subscription upgrade status moves to Up to
     date.
  - b. If you selected an **Automatic** approval strategy, the upgrade status should resolve to **Up to date** without intervention.
- 7. After the upgrade status of the subscription is Up to date, select Operators → Installed Operators to verify that the cluster service version (CSV) of the installed Operator eventually shows up. The Status should ultimately resolve to InstallSucceeded in the relevant namespace.



### **NOTE**

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

#### If it does not:

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

# 4.1.3. Installing from Operator Hub using the CLI

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

## **Prerequisites**

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

#### **Procedure**

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

\$ oc get packagemanifests -n openshift-marketplace

## **Example output**

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

Note the catalog for your desired Operator.

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

\$ oc describe packagemanifests < operator\_name > -n openshift-marketplace

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

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

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



### **NOTE**

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

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

## **Example Operator Group object**

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

metadata:

name: <operatorgroup\_name>
namespace: <namespace>

spec:

targetNamespaces:
- <namespace>

b. Create the **OperatorGroup** object:

\$ oc apply -f operator group.yaml

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

### **Example Subscription object**

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

```
envFrom: 7
- secretRef:
  name: license-secret
volumes: 8
- name: <volume_name>
 configMap:
  name: <configmap_name>
volumeMounts: 9
- mountPath: <directory_name>
 name: <volume name>
tolerations: 10
- operator: "Exists"
resources: 111
 requests:
  memory: "64Mi"
  cpu: "250m"
 limits:
  memory: "128Mi"
  cpu: "500m"
nodeSelector: 12
 foo: bar
```

- For default **AllNamespaces** install mode usage, specify the **openshift-operators** namespace. Alternatively, you can specify a custom global namespace, if you have created one. Otherwise, specify the relevant single namespace for **SingleNamespace** install mode usage.
- 2 Name of the channel to subscribe to.
- 3 Name of the Operator to subscribe to.
- Name of the catalog source that provides the Operator.
- Namespace of the catalog source. Use **openshift-marketplace** for the default OperatorHub catalog sources.
- The **env** parameter defines a list of Environment Variables that must exist in all containers in the pod created by OLM.
- 7 The **envFrom** parameter defines a list of sources to populate Environment Variables in the container.
- The **volumes** parameter defines a list of Volumes that must exist on the pod created by OLM.
- The **volumeMounts** parameter defines a list of VolumeMounts that must exist in all containers in the pod created by OLM. If a **volumeMount** references a **volume** that does not exist, OLM fails to deploy the Operator.
- The **tolerations** parameter defines a list of Tolerations for the pod created by OLM.
- The **resources** parameter defines resource constraints for all the containers in the pod created by OLM.
- The **nodeSelector** parameter defines a **NodeSelector** for the pod created by OLM.

# 5. Create the **Subscription** object:

\$ oc apply -f sub.yaml

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

### Additional resources

About Operator groups

# 4.1.4. Installing a specific version of an Operator

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

## **Prerequisites**

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

### **Procedure**

1. Look up the available versions and channels of the Operator you want to install by running the following command:

## **Command syntax**

\$ oc describe packagemanifests < operator\_name > -n < catalog\_namespace >

For example, the following command prints the available channels and versions of the Red Hat Quay Operator from OperatorHub:

### **Example command**

\$ oc describe packagemanifests quay-operator -n openshift-marketplace

## Example 4.1. Example output

Name: quay-operator

Namespace: operator-marketplace Labels: catalog=redhat-operators

catalog-namespace=openshift-marketplace

hypershift.openshift.io/managed=true operatorframework.io/arch.amd64=supported

operatorframework.io/arch.amd64=supported operatorframework.io/os.linux=supported

provider=Red Hat provider-url= Annotations: <none>

API Version: packages.operators.coreos.com/v1

```
Kind:
          PackageManifest
  Current CSV: quay-operator.v3.7.11
  Entries:
   Name:
             quay-operator.v3.7.11
   Version:
             3.7.11
   Name:
             quay-operator.v3.7.10
   Version:
             3.7.10
   Name:
             quay-operator.v3.7.9
   Version:
             3.7.9
   Name:
             quay-operator.v3.7.8
   Version: 3.7.8
   Name:
             quay-operator.v3.7.7
   Version: 3.7.7
   Name:
             quay-operator.v3.7.6
   Version:
             3.7.6
   Name:
             quay-operator.v3.7.5
   Version:
             3.7.5
   Name:
             quay-operator.v3.7.4
   Version:
             3.7.4
   Name:
             quay-operator.v3.7.3
   Version:
             3.7.3
   Name:
             quay-operator.v3.7.2
   Version:
             3.7.2
   Name:
             quay-operator.v3.7.1
   Version: 3.7.1
   Name:
             quay-operator.v3.7.0
   Version:
             3.7.0
  Name:
             stable-3.7
 Current CSV: quay-operator.v3.8.5
 Entries:
   Name:
              quay-operator.v3.8.5
   Version:
              3.8.5
   Name:
              quay-operator.v3.8.4
   Version:
              3.8.4
   Name:
              quay-operator.v3.8.3
   Version:
              3.8.3
   Name:
              quay-operator.v3.8.2
   Version:
              3.8.2
   Name:
              quay-operator.v3.8.1
   Version:
              3.8.1
   Name:
              quay-operator.v3.8.0
   Version:
              3.8.0
  Name:
              stable-3.8
 Default Channel: stable-3.8
 Package Name:
                  quay-operator
```

### TIP

You can print an Operator's version and channel information in the YAML format by running the following command:

\$ oc get packagemanifests < operator\_name > -n < catalog\_namespace > -o yaml

• If more than one catalog is installed in a namespace, run the following command to look up the available versions and channels of an Operator from a specific catalog:

\$ oc get packagemanifest \

- --selector=catalog=<catalogsource name> \
- --field-selector metadata.name=<operator\_name> \
- -n <catalog namespace> -o yaml



### **IMPORTANT**

If you do not specify the Operator's catalog, running the **oc get packagemanifest** and **oc describe packagemanifest** commands might return a package from an unexpected catalog if the following conditions are met:

- Multiple catalogs are installed in the same namespace.
- The catalogs contain the same Operators or Operators with the same name.
- 2. An Operator group, defined by an **OperatorGroup** object, selects target namespaces in which to generate required role-based access control (RBAC) access for all Operators in the same namespace as the Operator group.

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

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

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

### Example Operator Group object

apiVersion: operators.coreos.com/v1

kind: OperatorGroup

metadata:

name: <operatorgroup\_name> namespace: <namespace>

spec:

targetNamespaces:

- <namespace>

b. Create the **OperatorGroup** object:

\$ oc apply -f operatorgroup.yaml

3. Create a **Subscription** object YAML file that subscribes a namespace to an Operator with a specific version by setting the **startingCSV** field. Set the **installPlanApproval** field to **Manual** to prevent the Operator from automatically upgrading if a later version exists in the catalog. For example, the following **sub.yaml** file can be used to install the Red Hat Quay Operator specifically to version 3.7.10:

# Subscription with a specific starting Operator version

apiVersion: operators.coreos.com/v1alpha1

kind: Subscription

metadata:

name: quay-operator namespace: quay

spec:

channel: quay-operator.v3.7.10 installPlanApproval: Manual 1

name: quay-operator source: redhat-operators

sourceNamespace: openshift-marketplace startingCSV: quay-operator.v3.7.10 2

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

\$ oc apply -f sub.yaml

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

### Additional resources

- Manually approving a pending Operator update
- Installing global Operators in custom namespaces

# 4.1.5. Preparing for multiple instances of an Operator for multitenant clusters

As a cluster administrator, you can add multiple instances of an Operator for use in multitenant clusters. This is an alternative solution to either using the standard **All namespaces** install mode, which can be considered to violate the principle of least privilege, or the **Multinamespace** mode, which is not widely adopted. For more information, see "Operators in multitenant clusters".

In the following procedure, the *tenant* is a user or group of users that share common access and privileges for a set of deployed workloads. The *tenant Operator* is the instance of an Operator that is intended for use by only that tenant.

# **Prerequisites**

• All instances of the Operator you want to install must be the same version across a given cluster.



### **IMPORTANT**

For more information on this and other limitations, see "Operators in multitenant clusters".

### Procedure

- 1. Before installing the Operator, create a namespace for the tenant Operator that is separate from the tenant's namespace. For example, if the tenant's namespace is **team1**, you might create a **team1-operator** namespace:
  - a. Define a Namespace resource and save the YAML file, for example, team1-operator.yaml:

apiVersion: v1 kind: Namespace metadata:

name: team1-operator

- b. Create the namespace by running the following command:
  - \$ oc create -f team1-operator.yaml
- 2. Create an Operator group for the tenant Operator scoped to the tenant's namespace, with only that one namespace entry in the **spec.targetNamespaces** list:
  - a. Define an OperatorGroup resource and save the YAML file, for example, team1operatorgroup.yaml:

apiVersion: operators.coreos.com/v1

kind: OperatorGroup

metadata:

name: team1-operatorgroup namespace: team1-operator

spec:

targetNamespaces:

- team1 1
- Define only the tenant's namespace in the **spec.targetNamespaces** list.
- b. Create the Operator group by running the following command:

\$ oc create -f team1-operatorgroup.yaml

## **Next steps**

• Install the Operator in the tenant Operator namespace. This task is more easily performed by using the OperatorHub in the web console instead of the CLI; for a detailed procedure, see Installing from OperatorHub using the web console.



### **NOTE**

After completing the Operator installation, the Operator resides in the tenant Operator namespace and watches the tenant namespace, but neither the Operator's pod nor its service account are visible or usable by the tenant.

### Additional resources

• Operators in multitenant clusters

## 4.1.6. Installing global Operators in custom namespaces

When installing Operators with the OpenShift Container Platform web console, the default behavior installs Operators that support the **All namespaces** install mode into the default **openshift-operators** global namespace. This can cause issues related to shared install plans and update policies between all Operators in the namespace. For more details on these limitations, see "Multitenancy and Operator colocation".

As a cluster administrator, you can bypass this default behavior manually by creating a custom global namespace and using that namespace to install your individual or scoped set of Operators and their dependencies.

### **Prerequisites**

• You have access to the cluster as a user with the **cluster-admin** role.

### Procedure

- 1. Before installing the Operator, create a namespace for the installation of your desired Operator. This installation namespace will become the custom global namespace:
  - a. Define a Namespace resource and save the YAML file, for example, globaloperators.yaml:

apiVersion: v1 kind: Namespace metadata:

name: global-operators

- b. Create the namespace by running the following command:
  - \$ oc create -f global-operators.yaml
- 2. Create a custom *global Operator group*, which is an Operator group that watches all namespaces:
  - a. Define an **OperatorGroup** resource and save the YAML file, for example, **global-operatorgroup.yaml**. Omit both the **spec.selector** and **spec.targetNamespaces** fields to make it a *global Operator group*, which selects all namespaces:

apiVersion: operators.coreos.com/v1

kind: OperatorGroup

metadata:

name: global-operatorgroup namespace: global-operators



### NOTE

The **status.namespaces** of a created global Operator group contains the empty string (""), which signals to a consuming Operator that it should watch all namespaces.

b. Create the Operator group by running the following command:

\$ oc create -f global-operatorgroup.yaml

### Next steps

Install the desired Operator in your custom global namespace. Because the web console does
not populate the Installed Namespace menu during Operator installation with custom global
namespaces, this task can only be performed with the OpenShift CLI (oc). For a detailed
procedure, see Installing from OperatorHub using the CLI.



### NOTE

When you initiate the Operator installation, if the Operator has dependencies, the dependencies are also automatically installed in the custom global namespace. As a result, it is then valid for the dependency Operators to have the same update policy and shared install plans.

### Additional resources

Multitenancy and Operator colocation

# 4.1.7. Pod placement of Operator workloads

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

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

- Determine a node or set of nodes to target for the pods per your requirements. If available, note
  an existing label, such as **node-role.kubernetes.io/app**, that identifies the node or nodes.
   Otherwise, add a label, such as **myoperator**, by using a compute machine set or editing the node
  directly. You will use this label in a later step as the node selector on your project.
- 2. If you want to ensure that only pods with a certain label are allowed to run on the nodes, while steering unrelated workloads to other nodes, add a taint to the node or nodes by using a compute machine set or editing the node directly. Use an effect that ensures that new pods that do not match the taint cannot be scheduled on the nodes. For example, a myoperator:NoSchedule taint ensures that new pods that do not match the taint are not scheduled onto that node, but existing pods on the node are allowed to remain.
- 3. Create a project that is configured with a default node selector and, if you added a taint, a matching toleration.

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

### For Operator pods

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

### For Operand pods

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

### Additional resources

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

# 4.1.8. Controlling where an Operator is installed

By default, when you install an Operator, OpenShift Container Platform installs the Operator pod to one of your worker nodes randomly. However, there might be situations where you want that pod scheduled on a specific node or set of nodes.

The following examples describe situations where you might want to schedule an Operator pod to a specific node or set of nodes:

- If an Operator requires a particular platform, such as **amd64** or **arm64**
- If an Operator requires a particular operating system, such as Linux or Windows
- If you want Operators that work together scheduled on the same host or on hosts located on the same rack
- If you want Operators dispersed throughout the infrastructure to avoid downtime due to network or hardware issues

You can control where an Operator pod is installed by adding node affinity, pod affinity, or pod antiaffinity constraints to the Operator's **Subscription** object. Node affinity is a set of rules used by the scheduler to determine where a pod can be placed. Pod affinity enables you to ensure that related pods are scheduled to the same node. Pod anti-affinity allows you to prevent a pod from being scheduled on a node.

The following examples show how to use node affinity or pod anti-affinity to install an instance of the Custom Metrics Autoscaler Operator to a specific node in the cluster:

# Node affinity example that places the Operator pod on a specific node

apiVersion: operators.coreos.com/v1alpha1

kind: Subscription

metadata:

name: openshift-custom-metrics-autoscaler-operator

namespace: openshift-keda

```
spec:
 name: my-package
 source: my-operators
 sourceNamespace: operator-registries
  affinity:
   nodeAffinity: 1
     required During Scheduling Ignored During Execution: \\
      nodeSelectorTerms:
      - matchExpressions:
       - key: kubernetes.io/hostname
        operator: In
        values:
        - ip-10-0-163-94.us-west-2.compute.internal
```

A node affinity that requires the Operator's pod to be scheduled on a node named ip-10-0-163-94.us-west-2.compute.internal.

# Node affinity example that places the Operator pod on a node with a specific platform

```
apiVersion: operators.coreos.com/v1alpha1
kind: Subscription
metadata:
 name: openshift-custom-metrics-autoscaler-operator
 namespace: openshift-keda
spec:
 name: my-package
 source: my-operators
 sourceNamespace: operator-registries
 config:
  affinity:
   nodeAffinity: 1
     requiredDuringSchedulingIgnoredDuringExecution:
      nodeSelectorTerms:
      - matchExpressions:
       - key: kubernetes.io/arch
        operator: In
        values:
        - arm64
       - key: kubernetes.io/os
        operator: In
        values:
        - linux
```

A node affinity that requires the Operator's pod to be scheduled on a node with the kubernetes.io/arch=arm64 and kubernetes.io/os=linux labels.

# Pod affinity example that places the Operator pod on one or more specific nodes

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

115

```
metadata:
 name: openshift-custom-metrics-autoscaler-operator
 namespace: openshift-keda
spec:
 name: my-package
 source: my-operators
 sourceNamespace: operator-registries
 config:
  affinity:
   podAffinity: 1
    requiredDuringSchedulingIgnoredDuringExecution:
    - labelSelector:
       matchExpressions:
       - key: app
        operator: In
        values:
        - test
      topologyKey: kubernetes.io/hostname
```

A pod affinity that places the Operator's pod on a node that has pods with the **app=test** label.

# Pod anti-affinity example that prevents the Operator pod from one or more specific nodes

```
apiVersion: operators.coreos.com/v1alpha1
kind: Subscription
metadata:
 name: openshift-custom-metrics-autoscaler-operator
 namespace: openshift-keda
spec:
 name: my-package
 source: my-operators
 sourceNamespace: operator-registries
 config:
  affinity:
   podAntiAffinity: 1
    requiredDuringSchedulingIgnoredDuringExecution:
    - labelSelector:
       matchExpressions:
       - key: cpu
        operator: In
        values:
        - high
      topologyKey: kubernetes.io/hostname
```

A pod anti-affinity that prevents the Operator's pod from being scheduled on a node that has pods with the **cpu=high** label.

### **Procedure**

To control the placement of an Operator pod, complete the following steps:

1. Install the Operator as usual.

- 2. If needed, ensure that your nodes are labeled to properly respond to the affinity.
- 3. Edit the Operator **Subscription** object to add an affinity:

apiVersion: operators.coreos.com/v1alpha1 kind: Subscription metadata: name: openshift-custom-metrics-autoscaler-operator namespace: openshift-keda spec: name: my-package source: my-operators sourceNamespace: operator-registries config: affinity: 1 nodeAffinity: requiredDuringSchedulingIgnoredDuringExecution: nodeSelectorTerms: - matchExpressions: - key: kubernetes.io/hostname operator: In values: - ip-10-0-185-229.ec2.internal

Add a **nodeAffinity**, **podAffinity**, or **podAntiAffinity**. See the Additional resources section that follows for information about creating the affinity.

## Verification

• To ensure that the pod is deployed on the specific node, run the following command:

\$ oc get pods -o wide

# Example output

NAME READY STATUS RESTARTS AGE IP

NODE NOMINATED NODE READINESS GATES

custom-metrics-autoscaler-operator-5dcc45d656-bhshg 1/1 Running 0 50s

10.131.0.20 ip-10-0-185-229.ec2.internal <none> <none>

### Additional resources

- Understanding pod affinity
- Understanding node affinity
- Understanding how to update labels on nodes

## 4.2. UPDATING INSTALLED OPERATORS

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



### NOTE

For information on how OLM handles updates for installed Operators colocated in the same namespace, as well as an alternative method for installing Operators with custom global Operator groups, see Multitenancy and Operator colocation.

# 4.2.1. Preparing for an Operator update

The subscription of an installed Operator specifies an update channel that tracks and receives updates for the Operator. You can change the update channel to start tracking and receiving updates from a newer channel.

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



### NOTE

You cannot change installed Operators to a channel that is older than the current channel

Red Hat Customer Portal Labs include the following application that helps administrators prepare to update their Operators:

• Red Hat OpenShift Container Platform Operator Update Information Checker

You can use the application to search for Operator Lifecycle Manager-based Operators and verify the available Operator version per update channel across different versions of OpenShift Container Platform. Cluster Version Operator-based Operators are not included.

## 4.2.2. Changing the update channel for an Operator

You can change the update channel for an Operator by using the OpenShift Container Platform web console.

# TIP

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

### **Prerequisites**

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

### Procedure

- In the Administrator perspective of the web console, navigate to Operators → Installed Operators.
- 2. Click the name of the Operator you want to change the update channel for.
- 3. Click the **Subscription** tab.

- 4. Click the name of the update channel under **Update channel**.
- 5. Click the newer update channel that you want to change to, then click **Save**.
- 6. For subscriptions with an Automatic approval strategy, the update begins automatically. Navigate back to the Operators → Installed Operators page to monitor the progress of the update. When complete, the status changes to Succeeded and Up to date. For subscriptions with a Manual approval strategy, you can manually approve the update from the Subscription tab.

# 4.2.3. Manually approving a pending Operator update

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

## **Prerequisites**

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

### **Procedure**

- In the Administrator perspective of the OpenShift Container Platform web console, navigate to Operators → Installed Operators.
- 2. Operators that have a pending update display a status with **Upgrade available**. Click the name of the Operator you want to update.
- 3. Click the **Subscription** tab. Any updates requiring approval are displayed next to **Upgrade status**. For example, it might display **1 requires approval**.
- 4. Click 1 requires approval, then click Preview Install Plan.
- 5. Review the resources that are listed as available for update. When satisfied, click **Approve**.
- 6. Navigate back to the **Operators** → **Installed Operators** page to monitor the progress of the update. When complete, the status changes to **Succeeded** and **Up to date**.

# 4.3. DELETING OPERATORS FROM A CLUSTER

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



## **IMPORTANT**

You must successfully and completely uninstall an Operator prior to attempting to reinstall the same Operator. Failure to fully uninstall the Operator properly can leave resources, such as a project or namespace, stuck in a "Terminating" state and cause "error resolving resource" messages to be observed when trying to reinstall the Operator. For more information, see Reinstalling Operators after failed uninstallation.

# 4.3.1. Deleting Operators from a cluster using the web console

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

### **Prerequisites**

• You have access to an OpenShift Container Platform cluster web console using an account with **cluster-admin** permissions.

### **Procedure**

- 1. Navigate to the **Operators** → **Installed Operators** page.
- 2. Scroll or enter a keyword into the **Filter by name** field to find the Operator that you want to remove. Then, click on it.
- 3. On the right side of the **Operator Details** page, select **Uninstall Operator** from the **Actions** list. An **Uninstall Operator?** dialog box is displayed.
- 4. Select **Uninstall** to remove the Operator, Operator deployments, and pods. Following this action, the Operator stops running and no longer receives updates.



### NOTE

This action does not remove resources managed by the Operator, including custom resource definitions (CRDs) and custom resources (CRs). Dashboards and navigation items enabled by the web console and off-cluster resources that continue to run might need manual clean up. To remove these after uninstalling the Operator, you might need to manually delete the Operator CRDs.

# 4.3.2. Deleting Operators from a cluster using the CLI

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

# **Prerequisites**

- You have access to an OpenShift Container Platform cluster using an account with clusteradmin permissions.
- The OpenShift CLI (**oc**) is installed on your workstation.

### **Procedure**

- Check the current version of the subscribed Operator (for example, jaeger) in the currentCSV field:
  - \$ oc get subscription jaeger -n openshift-operators -o yaml | grep currentCSV

# **Example output**

- currentCSV: jaeger-operator.v1.8.2
- 2. Delete the subscription (for example, jaeger):
  - \$ oc delete subscription jaeger -n openshift-operators

**Example output** 

subscription.operators.coreos.com "jaeger" deleted

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

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

# **Example output**

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

# 4.3.3. Refreshing failing subscriptions

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

## Example output

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

### Example output

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

As a result, the subscription is stuck in this failing state and the Operator is unable to install or upgrade.

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

### **Prerequisites**

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

## **Procedure**

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

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

# **Example output**

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

NAME DISPLAY VERSION

REPLACES PHASE

clusterserviceversion.operators.coreos.com/elasticsearch-operator.5.0.0-65 OpenShift Elasticsearch Operator 5.0.0-65 Succeeded

2. Delete the subscription:

\$ oc delete subscription <subscription\_name> -n <namespace>

3. Delete the cluster service version:

\$ oc delete csv <csv\_name> -n <namespace>

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

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

## **Example output**

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

NAME DATA AGE configmap/1de9443b6324e629ddf31fed0a853a121275806170e34c926d69e53a7fcbccb 3 9m30s

5. Delete the job:

\$ oc delete job <job\_name> -n openshift-marketplace

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

6. Delete the config map:

\$ oc delete configmap < configmap\_name > -n openshift-marketplace

7. Reinstall the Operator using Operator Hub in the web console.

## Verification

Check that the Operator has been reinstalled successfully:

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

## 4.4. CONFIGURING OPERATOR LIFECYCLE MANAGER FEATURES

The Operator Lifecycle Manager (OLM) controller is configured by an **OLMConfig** custom resource (CR) named **cluster**. Cluster administrators can modify this resource to enable or disable certain features.

This document outlines the features currently supported by OLM that are configured by the **OLMConfig** resource.

# 4.4.1. Disabling copied CSVs

When an Operator is installed by Operator Lifecycle Manager (OLM), a simplified copy of its cluster service version (CSV) is created by default in every namespace that the Operator is configured to watch. These CSVs are known as *copied CSVs* and communicate to users which controllers are actively reconciling resource events in a given namespace.

When an Operator is configured to use the **AllNamespaces** install mode, versus targeting a single or specified set of namespaces, a copied CSV for the Operator is created in every namespace on the cluster. On especially large clusters, with namespaces and installed Operators potentially in the hundreds or thousands, copied CSVs consume an untenable amount of resources, such as OLM's memory usage, cluster etcd limits, and networking.

To support these larger clusters, cluster administrators can disable copied CSVs for Operators globally installed with the **AllNamespaces** mode.



## NOTE

If you disable copied CSVs, an Operator installed in **AllNamespaces** mode has their CSV copied only to the **openshift** namespace, instead of every namespace on the cluster. In disabled copied CSVs mode, the behavior differs between the web console and CLI:

- In the web console, the default behavior is modified to show copied CSVs from the **openshift** namespace in every namespace, even though the CSVs are not actually copied to every namespace. This allows regular users to still be able to view the details of these Operators in their namespaces and create related custom resources (CRs).
- In the OpenShift CLI (oc), regular users can view Operators installed directly in their namespaces by using the oc get csvs command, but the copied CSVs from the openshift namespace are not visible in their namespaces. Operators affected by this limitation are still available and continue to reconcile events in the user's namespace.

To view a full list of installed global Operators, similar to the web console behavior, all authenticated users can run the following command:

\$ oc get csvs -n openshift

### Procedure

• Edit the **OLMConfig** object named **cluster** and set the **spec.features.disableCopiedCSVs** field to **true**:

\$ oc apply -f - <<EOF apiVersion: operators.coreos.com/v1 kind: OLMConfig metadata: name: cluster spec:
features:
disableCopiedCSVs: true

EOF

1 Dis

Disabled copied CSVs for **AllNamespaces** install mode Operators

### Verification

 When copied CSVs are disabled, OLM captures this information in an event in the Operator's namespace:

\$ oc get events

## **Example output**

LAST SEEN TYPE REASON OBJECT MESSAGE 85s Warning DisabledCopiedCSVs clusterserviceversion/my-csv.v1.0.0 CSV copying disabled for operators/my-csv.v1.0.0

When the **spec.features.disableCopiedCSVs** field is missing or set to **false**, OLM recreates the copied CSVs for all Operators installed with the **AllNamespaces** mode and deletes the previously mentioned events.

### Additional resources

Install modes

# 4.5. CONFIGURING PROXY SUPPORT IN OPERATOR LIFECYCLE MANAGER

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

## Additional resources

- Configuring the cluster-wide proxy
- Configuring a custom PKI (custom CA certificate)
- Developing Operators that support proxy settings for Go, Ansible, and Helm

# 4.5.1. Overriding proxy settings of an Operator

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



### **IMPORTANT**

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

## **Prerequisites**

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

### **Procedure**

- 1. Navigate in the web console to the **Operators** → **OperatorHub** page.
- 2. Select the Operator and click Install.
- 3. On the **Install Operator** page, modify the **Subscription** object to include one or more of the following environment variables in the **spec** section:
  - HTTP PROXY
  - HTTPS\_PROXY
  - NO\_PROXY

For example:

## Subscription object with proxy setting overrides

apiVersion: operators.coreos.com/v1alpha1

kind: Subscription

metadata:

name: etcd-config-test

namespace: openshift-operators

spec: config: env:

name: HTTP\_PROXY value: test\_http

- name: HTTPS\_PROXY

value: test\_https
- name: NO\_PROXY

value: test

channel: clusterwide-alpha installPlanApproval: Automatic

name: etcd

source: community-operators

sourceNamespace: openshift-marketplace startingCSV: etcdoperator.v0.9.4-clusterwide



### **NOTE**

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

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

- 4. Click Install to make the Operator available to the selected namespaces.
- 5. After the CSV for the Operator appears in the relevant namespace, you can verify that custom proxy environment variables are set in the deployment. For example, using the CLI:

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

## **Example output**

name: HTTP\_PROXY value: test\_http

name: HTTPS\_PROXY value: test\_https

- name: NO PROXY

value: test

image: quay.io/coreos/etcd-

operator@sha256:66a37fd61a06a43969854ee6d3e21088a98b93838e284a6086b13917f96b0

d9c

# 4.5.2. Injecting a custom CA certificate

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

### Prerequisites

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

## Procedure

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

apiVersion: v1 kind: ConfigMap metadata:

name: trusted-ca

labels:

config.openshift.io/inject-trusted-cabundle: "true" 2

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

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

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

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

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



## **NOTE**

Deployments of an Operator can fail to validate the authority and display a **x509 certificate signed by unknown authority** error. This error can occur even after injecting a custom CA when using the subscription of an Operator. In this case, you can set the **mountPath** as /etc/ssl/certs for trusted-ca by using the subscription of an Operator.

## 4.6. VIEWING OPERATOR STATUS

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

# 4.6.1. Operator subscription condition types

Subscriptions can report the following condition types:

Table 4.1. Subscription condition types

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



### **NOTE**

Default OpenShift Container Platform cluster Operators are managed by the Cluster Version Operator (CVO) and they do not have a **Subscription** object. Application Operators are managed by Operator Lifecycle Manager (OLM) and they have a **Subscription** object.

### Additional resources

• Refreshing failing subscriptions

# 4.6.2. Viewing Operator subscription status by using the CLI

You can view Operator subscription status by using the CLI.

# **Prerequisites**

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

### **Procedure**

1. List Operator subscriptions:

\$ oc get subs -n <operator\_namespace>

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

\$ oc describe sub <subscription\_name> -n <operator\_namespace>

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

## Example output

Conditions:

Last Transition Time: 2019-07-29T13:42:57Z

Message: all available catalogsources are healthy

Reason: AllCatalogSourcesHealthy

Status: False

Type: CatalogSourcesUnhealthy



### **NOTE**

Default OpenShift Container Platform cluster Operators are managed by the Cluster Version Operator (CVO) and they do not have a **Subscription** object. Application Operators are managed by Operator Lifecycle Manager (OLM) and they have a **Subscription** object.

# 4.6.3. Viewing Operator catalog source status by using the CLI

You can view the status of an Operator catalog source by using the CLI.

## **Prerequisites**

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

## Procedure

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

\$ oc get catalogsources -n openshift-marketplace

# Example output

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

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

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

## **Example output**

Name: example-catalog

Namespace: openshift-marketplace

• • •

Status:

Connection State:

Address: example-catalog.openshift-marketplace.svc:50051

Last Connect: 2021-09-09T17:07:35Z Last Observed State: TRANSIENT\_FAILURE

Registry Service:

Created At: 2021-09-09T17:05:45Z

Port: 50051 Protocol: grpc

Service Name: example-catalog

Service Namespace: openshift-marketplace

In the preceding example output, the last observed state is **TRANSIENT\_FAILURE**. This state indicates that there is a problem establishing a connection for the catalog source.

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

\$ oc get pods -n openshift-marketplace

# **Example output**

NAME	READY S	TATUS	RES <sup>-</sup>	TARTS	AGE
certified-operators-cv9nn	1/1	Running	0	36m	
community-operators-6v8lp	1/1	Running	0	36	3m
marketplace-operator-86bfc	75f9b-jkgbc	1/1 Runr	ning	0	42m
example-catalog-bwt8z	0/1	ImagePullE	BackOff	0	3m55s
redhat-marketplace-57p8c	1/1	Running	0	36	m
redhat-operators-smxx8	1/1	Running	0	36n	n

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

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

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

# **Example output**

Name: example-catalog-bwt8z Namespace: openshift-marketplace

Priority: 0

Node: ci-ln-jyryyg2-f76d1-ggdbq-worker-b-vsxjd/10.0.128.2

... Events:

Type	Reason	Age	From	Message				
	l Scheduled blace/example			eduler Successfully assigned openshift- -f76d1-fgdbq-worker-b-vsxjd				
Norma	l AddedInter	face 47s	multus	Add eth0 [10.131.0.40/23] from				
openshi	ft-sdn							
Norma	l BackOff	20s (x2 over	46s) kubelet	Back-off pulling image				
"quay.io/example-org/example-catalog:v1"								
Warnir	ng Failed	20s (x2 over 4	46s) kubelet	Error: ImagePullBackOff				
Norma	l Pulling	8s (x3 over 47	s) kubelet	Pulling image "quay.io/example-				
org/example-catalog:v1"								
Warnir	ng Failed	8s (x3 over 47	7s) kubelet	Failed to pull image				
"quay.io/example-org/example-catalog:v1": rpc error: code = Unknown desc = reading manifest v1 in quay.io/example-org/example-catalog: unauthorized: access to the requested resource is not authorized								
	ng Failed	8s (x3 over 47	7s) kubelet	Error: ErrlmagePull				

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

### Additional resources

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

## 4.7. MANAGING OPERATOR CONDITIONS

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

# 4.7.1. Overriding Operator conditions

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



## **NOTE**

By default, the **Spec.Overrides** array is not present in an **OperatorCondition** object until it is added by a cluster administrator. The **Spec.Conditions** array is also not present until it is either added by a user or as a result of custom Operator logic.

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

## **Prerequisites**

- You have access to the cluster as a user with the **cluster-admin** role.
- An Operator with an **OperatorCondition** object, installed using OLM.

### **Procedure**

1. Edit the **OperatorCondition** object for the Operator:

\$ oc edit operatorcondition <name>

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

## **Example Operator condition override**

apiVersion: operators.coreos.com/v1

kind: OperatorCondition

metadata:

name: my-operator namespace: operators

spec:

overrides:

- type: Upgradeable 1

status: "True"

reason: "upgradelsSafe"

message: "This is a known issue with the Operator where it always reports that it cannot

be upgraded." conditions:

type: Upgradeable status: "False" reason: "migration"

message: "The operator is performing a migration." lastTransitionTime: "2020-08-24T23:15:55Z"

Allows the cluster administrator to change the upgrade readiness to **True**.

# 4.7.2. Updating your Operator to use Operator conditions

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

An Operator author can develop their Operator to use the **operator-lib** library such that, after the Operator has been deployed by OLM, it can set its own conditions. For more resources about setting Operator conditions as an Operator author, see the Enabling Operator conditions page.

# 4.7.2.1. Setting defaults

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

## 4.7.3. Additional resources

Operator conditions

# 4.8. ALLOWING NON-CLUSTER ADMINISTRATORS TO INSTALL OPERATORS

Cluster administrators can use Operator groups to allow regular users to install Operators.

### Additional resources

Operator groups

# 4.8.1. Understanding Operator installation policy

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

To ensure that an Operator cannot achieve cluster-scoped privileges and that users cannot escalate privileges using OLM, Cluster administrators can manually audit Operators before they are added to the cluster. Cluster administrators are also provided tools for determining and constraining which actions are allowed during an Operator installation or upgrade using service accounts.

Cluster administrators can associate an Operator group with a service account that has a set of privileges granted to it. The service account sets policy on Operators to ensure they only run within predetermined boundaries by using role-based access control (RBAC) rules. As a result, the Operator is unable to do anything that is not explicitly permitted by those rules.

By employing Operator groups, users with enough privileges can install Operators with a limited scope. As a result, more of the Operator Framework tools can safely be made available to more users, providing a richer experience for building applications with Operators.



### NOTE

Role-based access control (RBAC) for **Subscription** objects is automatically granted to every user with the **edit** or **admin** role in a namespace. However, RBAC does not exist on **OperatorGroup** objects; this absence is what prevents regular users from installing Operators. Pre-installing Operator groups is effectively what gives installation privileges.

Keep the following points in mind when associating an Operator group with a service account:

- The **APIService** and **CustomResourceDefinition** resources are always created by OLM using the **cluster-admin** role. A service account associated with an Operator group should never be granted privileges to write these resources.
- Any Operator tied to this Operator group is now confined to the permissions granted to the specified service account. If the Operator asks for permissions that are outside the scope of the service account, the install fails with appropriate errors so the cluster administrator can troubleshoot and resolve the issue.

### 4.8.1.1. Installation scenarios

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

- A cluster administrator creates a new Operator group and specifies a service account. All
  Operator(s) associated with this Operator group are installed and run against the privileges
  granted to the service account.
- A cluster administrator creates a new Operator group and does not specify any service account. OpenShift Container Platform maintains backward compatibility, so the default behavior remains and Operator installs and upgrades are permitted.
- For existing Operator groups that do not specify a service account, the default behavior remains and Operator installs and upgrades are permitted.
- A cluster administrator updates an existing Operator group and specifies a service account. OLM allows the existing Operator to continue to run with their current privileges. When such an existing Operator is going through an upgrade, it is reinstalled and run against the privileges granted to the service account like any new Operator.
- A service account specified by an Operator group changes by adding or removing permissions, or the existing service account is swapped with a new one. When existing Operators go through an upgrade, it is reinstalled and run against the privileges granted to the updated service account like any new Operator.
- A cluster administrator removes the service account from an Operator group. The default behavior remains and Operator installs and upgrades are permitted.

## 4.8.1.2. Installation workflow

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

- 1. The given **Subscription** object is picked up by OLM.
- 2. OLM fetches the Operator group tied to this subscription.
- 3. OLM determines that the Operator group has a service account specified.
- 4. OLM creates a client scoped to the service account and uses the scoped client to install the Operator. This ensures that any permission requested by the Operator is always confined to that of the service account in the Operator group.
- 5. OLM creates a new service account with the set of permissions specified in the CSV and assigns it to the Operator. The Operator runs as the assigned service account.

# 4.8.2. Scoping Operator installations

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

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

## **Prerequisites**

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

### Procedure

1. Create a new namespace:

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

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

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

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

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

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

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

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

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

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

- Specify a catalog source that already exists in the designated namespace or one that is in the global catalog namespace.
- 2 Specify a namespace where the catalog source was created.

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

### 4.8.2.1. Fine-grained permissions

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

- ClusterServiceVersion
- Subscription
- Secret
- ServiceAccount
- Service
- ClusterRole and ClusterRoleBinding

# • Role and RoleBinding

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



### **NOTE**

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

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

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

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

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

Required to get the secret from the OLM namespace.

# 4.8.3. Operator catalog access control

When an Operator catalog is created in the global catalog namespace **openshift-marketplace**, the catalog's Operators are made available cluster-wide to all namespaces. A catalog created in other namespaces only makes its Operators available in that same namespace of the catalog.

On clusters where non-cluster administrator users have been delegated Operator installation privileges, cluster administrators might want to further control or restrict the set of Operators those users are allowed to install. This can be achieved with the following actions:

- 1. Disable all of the default global catalogs.
- 2. Enable custom, curated catalogs in the same namespace where the relevant Operator groups have been pre-installed.

### Additional resources

- Disabling the default OperatorHub catalog sources
- Adding a catalog source to a cluster

# 4.8.4. Troubleshooting permission failures

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

### **Procedure**

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

apiVersion: operators.coreos.com/v1 kind: Subscription

metadata: name: etcd

namespace: scoped

status:

installPlanRef:

apiVersion: operators.coreos.com/v1

kind: InstallPlan name: install-4plp8 namespace: scoped resourceVersion: "117359"

uid: 2c1df80e-afea-11e9-bce3-5254009c9c23

2. Check the status of the **InstallPlan** object for any errors:

apiVersion: operators.coreos.com/v1

kind: InstallPlan

status:

conditions:

lastTransitionTime: "2019-07-26T21:13:10Z" lastUpdateTime: "2019-07-26T21:13:10Z"

message: 'error creating clusterrole etcdoperator.v0.9.4-clusterwide-dsfx4:

clusterroles.rbac.authorization.k8s.io

is forbidden: User "system:serviceaccount:scoped:scoped" cannot create resource

"clusterroles" in API group "rbac.authorization.k8s.io" at the cluster scope'

reason: InstallComponentFailed

status: "False" type: Installed phase: Failed

The error message tells you:

- The type of resource it failed to create, including the API group of the resource. In this case, it was **clusterroles** in the **rbac.authorization.k8s.io** group.
- The name of the resource.
- The type of error: **is forbidden** tells you that the user does not have enough permission to do the operation.
- The name of the user who attempted to create or update the resource. In this case, it refers to the service account specified in the Operator group.
- The scope of the operation: cluster scope or not.
   The user can add the missing permission to the service account and then iterate.



### NOTE

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

# 4.9. MANAGING CUSTOM CATALOGS

Cluster administrators and Operator catalog maintainers can create and manage custom catalogs packaged using the bundle format on Operator Lifecycle Manager (OLM) in OpenShift Container Platform.



## **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.

## Additional resources

Red Hat-provided Operator catalogs

## 4.9.1. Prerequisites

• You have installed the **opm** CLI.

# 4.9.2. File-based catalogs

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**

As of OpenShift Container Platform 4.11, the default Red Hat-provided Operator catalog releases in the file-based catalog format. The default Red Hat-provided Operator catalogs for OpenShift Container Platform 4.6 through 4.10 released in the deprecated SQLite database format.

The **opm** subcommands, flags, and functionality related to the SQLite database format are also deprecated and will be removed in a future release. The features are still supported and must be used for catalogs that use the deprecated SQLite database format.

Many of the **opm** subcommands and flags for working with the SQLite database format, such as **opm index prune**, do not work with the file-based catalog format. For more information about working with file-based catalogs, see Operator Framework packaging format and Mirroring images for a disconnected installation using the oc-mirror plugin .

# 4.9.2.1. 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**

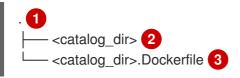
- You have installed the **opm** CLI.
- You have **podman** version 1.9.3+.
- A bundle image is built and pushed to a registry that supports Docker v2-2.

### **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:v4.13 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



- Parent directory
- 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
  - Operator, or package, name
  - Channel that subscriptions default to if unspecified
  - Path to the Operator's **README.md** or other documentation
  - 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
- 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

- 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

# 4.9.2.2. Updating or filtering a file-based catalog image

You can use the **opm** CLI to update or filter (also known as prune) a catalog image that uses the file-based catalog format. By extracting and modifying the contents of an existing catalog image, you can update, add, or remove one or more Operator package entries from the catalog. 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 ocmirror 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 by updating, adding, or removing one or more Operator package entries.



# **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.

For example, if you wanted to remove an Operator package, the following example lists a set of **olm.package**, **olm.channel**, and **olm.bundle** blobs which must be deleted to remove the package from the catalog:

Example 4.2. 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
  data: <base64 string>
relatedImages:
- image: example.com/example-inc/example-related-image@sha256:<digest>
 name: example-related-image
schema: olm.bundle
```

- 3. Save your changes to the index.yaml file.
- 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

- 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

- Mirroring images for a disconnected installation using the oc-mirror plugin → Keeping your mirror registry content updated
- Adding a catalog source to a cluster

# 4.9.3. SQLite-based catalogs



### **IMPORTANT**

The SQLite database format for Operator catalogs is a deprecated feature. Deprecated functionality is still included in OpenShift Container Platform and continues to be supported; however, it will be removed in a future release of this product and is not recommended for new deployments.

For the most recent list of major functionality that has been deprecated or removed within OpenShift Container Platform, refer to the *Deprecated and removed features* section of the OpenShift Container Platform release notes.

### 4.9.3.1. Creating a SQLite-based index image

You can create an index image based on the SQLite database format by using the opm CLI.

### **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. Start a new index:

\$ opm index add \
--bundles <registry>/<namespace>/<bundle\_image\_name>:<tag> \1
--tag <registry>/<namespace>/<index\_image\_name>:<tag> \2
[--binary-image <registry\_base\_image>] 3

- Comma-separated list of bundle images to add to the index.
- The image tag that you want the index image to have.
- Optional: An alternative registry base image to use for serving the catalog.
- 2. Push the index image to a registry.
  - a. If required, authenticate with your target registry:
    - \$ podman login <registry>
  - b. Push the index image:
    - \$ podman push <registry>/<namespace>/<index\_image\_name>:<tag>

# 4.9.3.2. Updating a SQLite-based index image

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

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

### **Prerequisites**

- You have installed the **opm** CLI.
- You have **podman** version 1.9.3+.
- An index image is built and pushed to a registry.
- You have an existing catalog source referencing the index image.

#### Procedure

1. Update the existing index by adding bundle images:

\$ opm index add \

- --bundles <registry>/<namespace>/<new\_bundle\_image>@sha256:<digest> \1
- --from-index <registry>/<namespace>/<existing\_index\_image>:<existing\_tag> \2
- --tag <registry>/<namespace>/<existing\_index\_image>:<updated\_tag> \3
- --pull-tool podman 4
- The **--bundles** flag specifies a comma-separated list of additional bundle images to add to the index.
- The **--from-index** flag specifies the previously pushed index.
- The **--tag** flag specifies the image tag to apply to the updated index image.
- The **--pull-tool** flag specifies the tool used to pull container images.

where:

# <registry>

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

### <namespace>

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

# <new\_bundle\_image>

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

# <digest>

Specifies the SHA image ID, or digest, of the bundle image, such as **c7f11097a628f092d8bad148406aa0e0951094a03445fd4bc0775431ef683a41**.

#### <existing index image>

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

### <existing\_tag>

Specifies a previously pushed image tag, such as **4.13**.

#### <updated tag>

Specifies the image tag to apply to the updated index image, such as **4.13.1**.

# **Example command**

\$ opm index add \

--bundles quay.io/ocs-dev/ocs-

operator@sha256:c7f11097a628f092d8bad148406aa0e0951094a03445fd4bc0775431ef683a41  $\$ 

- --from-index mirror.example.com/abc/abc-redhat-operator-index:4.13 \
- --tag mirror.example.com/abc/abc-redhat-operator-index:4.13.1 \
- --pull-tool podman

### 2. Push the updated index image:

\$ podman push <registry>/<namespace>/<existing\_index\_image>:<updated\_tag>

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

\$ oc get packagemanifests -n openshift-marketplace

# 4.9.3.3. Filtering a SQLite-based index image

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

### **Prerequisites**

- You have **podman** version 1.9.3+.
- You have **grpcurl** (third-party command-line tool).
- You have installed the **opm** CLI.
- You have access to a registry that supports Docker v2-2.

#### **Procedure**

- 1. Authenticate with your target registry:
  - \$ podman login <target\_registry>
- 2. Determine the list of packages you want to include in your pruned index.
  - a. Run the source index image that you want to prune in a container. For example:

```
$ podman run -p50051:50051 \
  -it registry.redhat.io/redhat/redhat-operator-index:v4.13
```

# **Example output**

Trying to pull registry.redhat.io/redhat/redhat-operator-index:v4.13... Getting image source signatures Copying blob ae8a0c23f5b1 done

INFO[0000] serving registry

database=/database/index.db port=50051

- b. In a separate terminal session, use the **grpcurl** command to get a list of the packages provided by the index:
  - \$ grpcurl -plaintext localhost:50051 api.Registry/ListPackages > packages.out
- c. Inspect the **packages.out** file and identify which package names from this list you want to keep in your pruned index. For example:

# Example snippets of packages list

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

- d. In the terminal session where you executed the **podman run** command, press **Ctrl** and **C** to stop the container process.
- 3. Run the following command to prune the source index of all but the specified packages:

\$ opm index prune \

- -f registry.redhat.io/redhat/redhat-operator-index:v4.13 \1
- -p advanced-cluster-management, jaeger-product, quay-operator \2
- [-i registry.redhat.io/openshift4/ose-operator-registry:v4.9] \3
- -t <target\_registry>:<port>/<namespace>/redhat-operator-index:v4.13 4
- 1 Index to prune.
- Comma-separated list of packages to keep.
- Required only for IBM Power and IBM Z images: Operator Registry base image with the tag that matches the target OpenShift Container Platform cluster major and minor version.
- 4 Custom tag for new index image being built.
- 4. Run the following command to push the new index image to your target registry:
  - \$ podman push <target\_registry>:<port>/<namespace>/redhat-operator-index:v4.13

where **<namespace>** is any existing namespace on the registry.

# 4.9.4. Catalog sources and pod security admission

Pod security admission was introduced in OpenShift Container Platform 4.11 to ensure pod security standards. Catalog sources built using the SQLite-based catalog format and a version of the **opm** CLI tool released before OpenShift Container Platform 4.11 cannot run under restricted pod security enforcement.

In OpenShift Container Platform 4.13, namespaces do not have restricted pod security enforcement by default and the default catalog source security mode is set to **legacy**.

Default restricted enforcement for all namespaces is planned for inclusion in a future OpenShift Container Platform release. When restricted enforcement occurs, the security context of the pod specification for catalog source pods must match the restricted pod security standard. If your catalog source image requires a different pod security standard, the pod security admissions label for the namespace must be explicitly set.



### NOTE

If you do not want to run your SQLite-based catalog source pods as restricted, you do not need to update your catalog source in OpenShift Container Platform 4.13.

However, it is recommended that you take action now to ensure your catalog sources run under restricted pod security enforcement. If you do not take action to ensure your catalog sources run under restricted pod security enforcement, your catalog sources might not run in future OpenShift Container Platform releases.

As a catalog author, you can enable compatibility with restricted pod security enforcement by completing either of the following actions:

- Migrate your catalog to the file-based catalog format.
- Update your catalog image with a version of the opm CLI tool released with OpenShift Container Platform 4.11 or later.



#### NOTE

The SQLite database catalog format is deprecated, but still supported by Red Hat. In a future release, the SQLite database format will not be supported, and catalogs will need to migrate to the file-based catalog format. As of OpenShift Container Platform 4.11, the default Red Hat-provided Operator catalog is released in the file-based catalog format. File-based catalogs are compatible with restricted pod security enforcement.

If you do not want to update your SQLite database catalog image or migrate your catalog to the file-based catalog format, you can configure your catalog to run with elevated permissions.

#### Additional resources

• Understanding and managing pod security admission

# 4.9.4.1. Migrating SQLite database catalogs to the file-based catalog format

You can update your deprecated SQLite database format catalogs to the file-based catalog format.

# Prerequisites

- You have a SQLite database catalog source.
- You have access to the cluster as a user with the **cluster-admin** role.
- You have the latest version of the **opm** CLI tool released with OpenShift Container Platform 4.13 on your workstation.

#### Procedure

1. Migrate your SQLite database catalog to a file-based catalog by running the following command:

\$ opm migrate <registry\_image> <fbc\_directory>

2. Generate a Dockerfile for your file-based catalog by running the following command:

```
$ opm generate dockerfile <fbc_directory> \
--binary-image \
registry.redhat.io/openshift4/ose-operator-registry:v4.13
```

### **Next steps**

• The generated Dockerfile can be built, tagged, and pushed to your registry.

#### Additional resources

• Adding a catalog source to a cluster

# 4.9.4.2. Rebuilding SQLite database catalog images

You can rebuild your SQLite database catalog image with the latest version of the **opm** CLI tool that is released with your version of OpenShift Container Platform.

# **Prerequisites**

- You have a SQLite database catalog source.
- You have access to the cluster as a user with the **cluster-admin** role.
- You have the latest version of the **opm** CLI tool released with OpenShift Container Platform 4.13 on your workstation.

#### Procedure

• Run the following command to rebuild your catalog with a more recent version of the **opm** CLI tool:

```
$ opm index add --binary-image \
  registry.redhat.io/openshift4/ose-operator-registry:v4.13 \
  --from-index <your_registry_image> \
  --bundles "" -t \<your_registry_image>
```

# 4.9.4.3. Configuring catalogs to run with elevated permissions

If you do not want to update your SQLite database catalog image or migrate your catalog to the file-based catalog format, you can perform the following actions to ensure your catalog source runs when the default pod security enforcement changes to restricted:

- Manually set the catalog security mode to legacy in your catalog source definition. This action
  ensures your catalog runs with legacy permissions even if the default catalog security mode
  changes to restricted.
- Label the catalog source namespace for baseline or privileged pod security enforcement.



#### NOTE

The SQLite database catalog format is deprecated, but still supported by Red Hat. In a future release, the SQLite database format will not be supported, and catalogs will need to migrate to the file-based catalog format. File-based catalogs are compatible with restricted pod security enforcement.

# **Prerequisites**

- You have a SQLite database catalog source.
- You have access to the cluster as a user with the **cluster-admin** role.
- You have a target namespace that supports running pods with the elevated pod security admission standard of baseline or privileged.

#### **Procedure**

1. Edit the **CatalogSource** definition by setting the spec.grpcPodConfig.securityContextConfig label to legacy, as shown in the following example:

# **Example CatalogSource definition**

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

kind: CatalogSource

metadata:

name: my-catsrc namespace: my-ns

spec:

sourceType: grpc grpcPodConfig:

securityContextConfig: legacy

image: my-image:latest

### TIP

In OpenShift Container Platform 4.13, the spec.grpcPodConfig.securityContextConfig field is set to legacy by default. In a future release of OpenShift Container Platform, it is planned that the default setting will change to restricted. If your catalog cannot run under restricted enforcement, it is recommended that you manually set this field to legacy.

2. Edit your <namespace>.yaml file to add elevated pod security admission standards to your catalog source namespace, as shown in the following example:

# Example < namespace > . yaml file

apiVersion: v1 kind: Namespace metadata:

labels:

security.openshift.io/scc.podSecurityLabelSync: "false" 1

openshift.io/cluster-monitoring: "true" pod-security.kubernetes.io/enforce: baseline 2 name: "<namespace name>"

- Turn off pod security label synchronization by adding the **security.openshift.io/scc.podSecurityLabelSync=false** label to the namespace.
- Apply the pod security admission **pod-security.kubernetes.io/enforce** label. Set the label to **baseline** or **privileged**. Use the **baseline** pod security profile unless other workloads in the namespace require a **privileged** profile.

# 4.9.5. Adding a catalog source to a cluster

Adding a catalog source to an OpenShift Container Platform cluster enables the discovery and installation of Operators for users. Cluster administrators can create a **CatalogSource** object that references an index image. OperatorHub uses catalog sources to populate the user interface.

#### TIP

Alternatively, you can use the web console to manage catalog sources. From the **Administration** → **Cluster Settings** → **Configuration** → **OperatorHub** page, click the **Sources** tab, where you can create, update, delete, disable, and enable individual sources.

# **Prerequisites**

- You built and pushed an index image to a registry.
- You have access to the cluster as a user with the **cluster-admin** role.

#### **Procedure**

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

```
apiVersion: operators.coreos.com/v1alpha1
kind: CatalogSource
metadata:
 name: my-operator-catalog
 namespace: openshift-marketplace 1
 annotations:
  olm.catalogImageTemplate: 2
   "<registry>/<namespace>/<index_image_name>:v{kube_major_version}.
{kube_minor_version}.{kube_patch_version}"
spec:
 sourceType: grpc
 grpcPodConfig:
  securityContextConfig: <security_mode> 3
 image: <registry>/<namespace>/<index image name>:<tag> 4
 displayName: My Operator Catalog
 publisher: <publisher_name> 5
```

updateStrategy: registryPoll: 6 interval: 30m

- If you want the catalog source to be available globally to users in all namespaces, specify the **openshift-marketplace** namespace. Otherwise, you can specify a different namespace for the catalog to be scoped and available only for that namespace.
- Optional: Set the **olm.catalogImageTemplate** annotation to your index image name and use one or more of the Kubernetes cluster version variables as shown when constructing the template for the image tag.
- Specify the value of **legacy** or **restricted**. If the field is not set, the default value is **legacy**. In a future OpenShift Container Platform release, it is planned that the default value will be **restricted**. If your catalog cannot run with **restricted** permissions, it is recommended that you manually set this field to **legacy**.
- Specify your index image. If you specify a tag after the image name, for example :v4.13, the catalog source pod uses an image pull policy of Always, meaning the pod always pulls the image prior to starting the container. If you specify a digest, for example @sha256:<id>, the image pull policy is IfNotPresent, meaning the pod pulls the image only if it does not already exist on the node.
- Specify your name or an organization name publishing the catalog.
- 6 Catalog sources can automatically check for new versions to keep up to date.
- b. Use the file to create the **CatalogSource** object:
  - \$ oc apply -f catalogSource.yaml
- 2. Verify the following resources are created successfully.
  - a. Check the pods:
    - \$ oc get pods -n openshift-marketplace

### Example output

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

b. Check the catalog source:

\$ oc get catalogsource -n openshift-marketplace

# Example output

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

c. Check the package manifest:

\$ oc get packagemanifest -n openshift-marketplace

# Example output

NAME CATALOG AGE jaeger-product My Operator Catalog 93s

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

#### Additional resources

- Operator Lifecycle Manager concepts and resources → Catalog source
- Accessing images for Operators from private registries
- Image pull policy

# 4.9.6. Accessing images for Operators from private registries

If certain images relevant to Operators managed by Operator Lifecycle Manager (OLM) are hosted in an authenticated container image registry, also known as a private registry, OLM and OperatorHub are unable to pull the images by default. To enable access, you can create a pull secret that contains the authentication credentials for the registry. By referencing one or more pull secrets in a catalog source, OLM can handle placing the secrets in the Operator and catalog namespace to allow installation.

Other images required by an Operator or its Operands might require access to private registries as well. OLM does not handle placing the secrets in target tenant namespaces for this scenario, but authentication credentials can be added to the global cluster pull secret or individual namespace service accounts to enable the required access.

The following types of images should be considered when determining whether Operators managed by OLM have appropriate pull access:

#### Index images

A **CatalogSource** object can reference an index image, which use the Operator bundle format and are catalog sources packaged as container images hosted in images registries. If an index image is hosted in a private registry, a secret can be used to enable pull access.

# **Bundle images**

Operator bundle images are metadata and manifests packaged as container images that represent a unique version of an Operator. If any bundle images referenced in a catalog source are hosted in one or more private registries, a secret can be used to enable pull access.

# **Operator and Operand images**

If an Operator installed from a catalog source uses a private image, either for the Operator image itself or one of the Operand images it watches, the Operator will fail to install because the deployment will not have access to the required registry authentication. Referencing secrets in a catalog source does not enable OLM to place the secrets in target tenant namespaces in which Operands are installed.

Instead, the authentication details can be added to the global cluster pull secret in the **openshift-config** namespace, which provides access to all namespaces on the cluster. Alternatively, if providing access to the entire cluster is not permissible, the pull secret can be added to the **default** service accounts of the target tenant namespaces.

### **Prerequisites**

- You have at least one of the following hosted in a private registry:
  - An index image or catalog image.
  - An Operator bundle image.
  - An Operator or Operand image.
- You have access to the cluster as a user with the **cluster-admin** role.

#### **Procedure**

- 1. Create a secret for each required private registry.
  - a. Log in to the private registry to create or update your registry credentials file:
    - \$ podman login <registry>:<port>



#### NOTE

The file path of your registry credentials can be different depending on the container tool used to log in to the registry. For the **podman** CLI, the default location is **\${XDG\_RUNTIME\_DIR}/containers/auth.json**. For the **docker** CLI, the default location is **/root/.docker/config.json**.

b. It is recommended to include credentials for only one registry per secret, and manage credentials for multiple registries in separate secrets. Multiple secrets can be included in a CatalogSource object in later steps, and OpenShift Container Platform will merge the secrets into a single virtual credentials file for use during an image pull.
 A registry credentials file can, by default, store details for more than one registry or for multiple repositories in one registry. Verify the current contents of your file. For example:

# File storing credentials for multiple registries

```
{
    "auths": {
        "registry.redhat.io": {
            "auth": "FrNHNydQXdzclNqdg=="
        },
        "quay.io": {
            "auth": "fegdsRib21iMQ=="
        },
        "https://quay.io/my-namespace/my-user/my-image": {
            "auth": "eWfjwsDdfsa221=="
        },
        "https://quay.io/my-namespace/my-user": {
            "auth": "feFweDdscw34rR=="
        },
        "https://quay.io/my-namespace": {
            "auth": "frwEews4fescyq=="
        }
    }
}
```

Because this file is used to create secrets in later steps, ensure that you are storing details for only one registry per file. This can be accomplished by using either of the following methods:

- Use the podman logout <registry> command to remove credentials for additional registries until only the one registry you want remains.
- Edit your registry credentials file and separate the registry details to be stored in multiple files. For example:

# File storing credentials for one registry

# File storing credentials for another registry

c. Create a secret in the **openshift-marketplace** namespace that contains the authentication credentials for a private registry:

```
$ oc create secret generic <secret_name> \
    -n openshift-marketplace \
    --from-file=.dockerconfigjson=<path/to/registry/credentials> \
    --type=kubernetes.io/dockerconfigjson
```

Repeat this step to create additional secrets for any other required private registries, updating the **--from-file** flag to specify another registry credentials file path.

2. Create or update an existing **CatalogSource** object to reference one or more secrets:

```
apiVersion: operators.coreos.com/v1alpha1
kind: CatalogSource
metadata:
name: my-operator-catalog
namespace: openshift-marketplace
spec:
sourceType: grpc
secrets: 1
- "<secret_name_1>"
- "<secret_name_2>"
grpcPodConfig:
```

securityContextConfig: <security\_mode> 2
image: <registry>:<port>/<namespace>/<image>:<tag>
displayName: My Operator Catalog
publisher: <publisher\_name>
updateStrategy:
registryPoll:
interval: 30m

- Add a **spec.secrets** section and specify any required secrets.
- Specify the value of **legacy** or **restricted**. If the field is not set, the default value is **legacy**. In a future OpenShift Container Platform release, it is planned that the default value will be **restricted**. If your catalog cannot run with **restricted** permissions, it is recommended that you manually set this field to **legacy**.
- 3. If any Operator or Operand images that are referenced by a subscribed Operator require access to a private registry, you can either provide access to all namespaces in the cluster, or individual target tenant namespaces.
  - To provide access to all namespaces in the cluster, add authentication details to the global cluster pull secret in the **openshift-config** namespace.



#### **WARNING**

Cluster resources must adjust to the new global pull secret, which can temporarily limit the usability of the cluster.

- a. Extract the **.dockerconfigjson** file from the global pull secret:
  - \$ oc extract secret/pull-secret -n openshift-config --confirm
- b. Update the **.dockerconfigjson** file with your authentication credentials for the required private registry or registries and save it as a new file:

- Replace < registry >: < port > / < namespace > with the private registry details and < token > with your authentication credentials.
- c. Update the global pull secret with the new file:
  - \$ oc set data secret/pull-secret -n openshift-config \
    --from-file=.dockerconfigjson=new\_dockerconfigjson

- To update an individual namespace, add a pull secret to the service account for the Operator that requires access in the target tenant namespace.
  - a. Recreate the secret that you created for the **openshift-marketplace** in the tenant namespace:

```
$ oc create secret generic <secret_name> \
    -n <tenant_namespace> \
    --from-file=.dockerconfigjson=<path/to/registry/credentials> \
    --type=kubernetes.io/dockerconfigjson
```

- b. Verify the name of the service account for the Operator by searching the tenant namespace:
  - \$ oc get sa -n <tenant\_namespace> 1
  - If the Operator was installed in an individual namespace, search that namespace. If the Operator was installed for all namespaces, search the **openshift-operators** namespace.

# Example output

```
NAME SECRETS AGE
builder 2 6m1s
default 2 6m1s
deployer 2 6m1s
etcd-operator 2 5m18s 1
```

- 1 Service account for an installed etcd Operator.
- c. Link the secret to the service account for the Operator:

```
$ oc secrets link <operator_sa> \
    -n <tenant_namespace> \
    <secret_name> \
    --for=pull
```

#### Additional resources

- See What is a secret? for more information on the types of secrets, including those used for registry credentials.
- See Updating the global cluster pull secret for more details on the impact of changing this secret.
- See Allowing pods to reference images from other secured registries for more details on linking pull secrets to service accounts per namespace.

# 4.9.7. Disabling the default OperatorHub catalog sources

Operator catalogs that source content provided by Red Hat and community projects are configured for OperatorHub by default during an OpenShift Container Platform installation. As a cluster administrator, you can disable the set of default catalogs.

#### Procedure

Disable the sources for the default catalogs by adding disableAllDefaultSources: true to the OperatorHub object:

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

#### TIP

Alternatively, you can use the web console to manage catalog sources. From the **Administration** → **Cluster Settings** → **Configuration** → **OperatorHub** page, click the **Sources** tab, where you can create, update, delete, disable, and enable individual sources.

# 4.9.8. Removing custom catalogs

As a cluster administrator, you can remove custom Operator catalogs that have been previously added to your cluster by deleting the related catalog source.

### **Prerequisites**

• You have access to the cluster as a user with the **cluster-admin** role.

### Procedure

- In the Administrator perspective of the web console, navigate to Administration → Cluster Settings.
- 2. Click the **Configuration** tab, and then click **OperatorHub**.
- 3. Click the Sources tab.
- 4. Select the **Options** menu for the catalog that yo **CatalogSource**.

for the catalog that you want to remove, and then click Delete

# 4.10. USING OPERATOR LIFECYCLE MANAGER ON RESTRICTED NETWORKS

For OpenShift Container Platform clusters that are installed on restricted networks, also known as disconnected clusters, Operator Lifecycle Manager (OLM) by default cannot access the Red Hatprovided OperatorHub sources hosted on remote registries because those remote sources require full internet connectivity.

However, as a cluster administrator you can still enable your cluster to use OLM in a restricted network if you have a workstation that has full internet access. The workstation, which requires full internet access to pull the remote OperatorHub content, is used to prepare local mirrors of the remote sources, and push the content to a mirror registry.

The mirror registry can be located on a bastion host, which requires connectivity to both your workstation and the disconnected cluster, or a completely disconnected, or *airgapped*, host, which requires removable media to physically move the mirrored content to the disconnected environment.

This guide describes the following process that is required to enable OLM in restricted networks:

- Disable the default remote OperatorHub sources for OLM.
- Use a workstation with full internet access to create and push local mirrors of the OperatorHub content to a mirror registry.
- Configure OLM to install and manage Operators from local sources on the mirror registry instead of the default remote sources.

After enabling OLM in a restricted network, you can continue to use your unrestricted workstation to keep your local OperatorHub sources updated as newer versions of Operators are released.



### **IMPORTANT**

While OLM can manage Operators from local sources, the ability for a given Operator to run successfully in a restricted network still depends on the Operator itself meeting the following criteria:

- List any related images, or other container images that the Operator might require to perform their functions, in the **relatedImages** parameter of its **ClusterServiceVersion** (CSV) object.
- Reference all specified images by a digest (SHA) and not by a tag.

You can search software on the Red Hat Ecosystem Catalog for a list of Red Hat Operators that support running in disconnected mode by filtering with the following selections:

Туре	Containerized application
Deployme nt method	Operator
Infrastruct ure features	Disconnected

#### Additional resources

- Red Hat-provided Operator catalogs
- Enabling your Operator for restricted network environments

# 4.10.1. Prerequisites

• Log in to your OpenShift Container Platform cluster as a user with **cluster-admin** privileges.



#### NOTE

If you are using OLM in a restricted network on IBM Z, you must have at least 12 GB allocated to the directory where you place your registry.

# 4.10.2. Disabling the default OperatorHub catalog sources

Operator catalogs that source content provided by Red Hat and community projects are configured for OperatorHub by default during an OpenShift Container Platform installation. In a restricted network environment, you must disable the default catalogs as a cluster administrator. You can then configure OperatorHub to use local catalog sources.

#### Procedure

 Disable the sources for the default catalogs by adding disableAllDefaultSources: true to the OperatorHub object:

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

#### **TIP**

Alternatively, you can use the web console to manage catalog sources. From the **Administration** → **Cluster Settings** → **Configuration** → **OperatorHub** page, click the **Sources** tab, where you can create, update, delete, disable, and enable individual sources.

# 4.10.3. Mirroring an Operator catalog

For instructions about mirroring Operator catalogs for use with disconnected clusters, see Installing → Mirroring images for a disconnected installation.



# **IMPORTANT**

As of OpenShift Container Platform 4.11, the default Red Hat-provided Operator catalog releases in the file-based catalog format. The default Red Hat-provided Operator catalogs for OpenShift Container Platform 4.6 through 4.10 released in the deprecated SQLite database format.

The **opm** subcommands, flags, and functionality related to the SQLite database format are also deprecated and will be removed in a future release. The features are still supported and must be used for catalogs that use the deprecated SQLite database format.

Many of the **opm** subcommands and flags for working with the SQLite database format, such as **opm index prune**, do not work with the file-based catalog format. For more information about working with file-based catalogs, see Operator Framework packaging format, Managing custom catalogs, and Mirroring images for a disconnected installation using the oc-mirror plugin.

# 4.10.4. Adding a catalog source to a cluster

Adding a catalog source to an OpenShift Container Platform cluster enables the discovery and installation of Operators for users. Cluster administrators can create a **CatalogSource** object that references an index image. OperatorHub uses catalog sources to populate the user interface.

#### TIP

Alternatively, you can use the web console to manage catalog sources. From the **Administration** → **Cluster Settings** → **Configuration** → **OperatorHub** page, click the **Sources** tab, where you can create, update, delete, disable, and enable individual sources.

### **Prerequisites**

- You built and pushed an index image to a registry.
- You have access to the cluster as a user with the **cluster-admin** role.

#### **Procedure**

- Create a CatalogSource object that references your index image. If you used the oc adm catalog mirror command to mirror your catalog to a target registry, you can use the generated catalogSource.yaml file in your manifests directory as a starting point.
  - a. Modify the following to your specifications and save it as a catalogSource.yaml file:

```
apiVersion: operators.coreos.com/v1alpha1
kind: CatalogSource
metadata:
name: my-operator-catalog 1
namespace: openshift-marketplace 2
spec:
sourceType: grpc
grpcPodConfig:
securityContextConfig: <security_mode> 3
image: <registry>/<namespace>/redhat-operator-index:v4.13 4
displayName: My Operator Catalog
publisher: <publisher_name> 5
updateStrategy:
registryPoll: 6
interval: 30m
```

- If you mirrored content to local files before uploading to a registry, remove any backslash (/) characters from the **metadata.name** field to avoid an "invalid resource name" error when you create the object.
- If you want the catalog source to be available globally to users in all namespaces, specify the **openshift-marketplace** namespace. Otherwise, you can specify a different namespace for the catalog to be scoped and available only for that namespace.
- Specify the value of **legacy** or **restricted**. If the field is not set, the default value is **legacy**. In a future OpenShift Container Platform release, it is planned that the default value will be **restricted**. If your catalog cannot run with **restricted** permissions, it is recommended that you manually set this field to **legacy**.
- Specify your index image. If you specify a tag after the image name, for example :v4.13, the catalog source pod uses an image pull policy of Always, meaning the pod always pulls the image prior to starting the container. If you specify a digest, for example @sha256:<id>, the image pull policy is IfNotPresent, meaning the pod pulls the image only if it does not already exist on the node.

- 5 Specify your name or an organization name publishing the catalog.
- 6 Catalog sources can automatically check for new versions to keep up to date.
- b. Use the file to create the **CatalogSource** object:
  - \$ oc apply -f catalogSource.yaml
- 2. Verify the following resources are created successfully.
  - a. Check the pods:
    - \$ oc get pods -n openshift-marketplace

# Example output

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

- b. Check the catalog source:
  - \$ oc get catalogsource -n openshift-marketplace

# Example output

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

- c. Check the package manifest:
  - \$ oc get packagemanifest -n openshift-marketplace

# **Example output**

NAME CATALOG AGE jaeger-product My Operator Catalog 93s

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

### Additional resources

- Accessing images for Operators from private registries
- Image template for custom catalog sources
- Image pull policy

# 4.11. CATALOG SOURCE POD SCHEDULING

When an Operator Lifecycle Manager (OLM) catalog source of source type **grpc** defines a **spec.image**, the Catalog Operator creates a pod that serves the defined image content. By default, this pod defines the following in its specification:

- Only the **kubernetes.io/os=linux** node selector.
- The default priority class name: **system-cluster-critical**.
- No tolerations.

As an administrator, you can override these values by modifying fields in the **CatalogSource** object's optional **spec.grpcPodConfig** section.



#### **IMPORTANT**

The Marketplace Operator, **openshift-marketplace**, manages the default **OperatorHub** custom resource's (CR). This CR manages **CatalogSource** objects. If you attempt to modify fields in the **CatalogSource** object's **spec.grpcPodConfig** section, the Marketplace Operator automatically reverts these modifications. By default, if you modify fields in the **spec.grpcPodConfig** section of the **CatalogSource** object, the Marketplace Operator automatically reverts these changes.

To apply persistent changes to **CatalogSource** object, you must first disable a default **CatalogSource** object.

#### Additional resources

• OLM concepts and resources → Catalog source

# 4.11.1. Disabling default CatalogSource objects at a local level

You can apply persistent changes to a **CatalogSource** object, such as catalog source pods, at a local level, by disabling a default **CatalogSource** object. Consider the default configuration in situations where the default **CatalogSource** object's configuration does not meet your organization's needs. By default, if you modify fields in the **spec.grpcPodConfig** section of the **CatalogSource** object, the Marketplace Operator automatically reverts these changes.

The Marketplace Operator, **openshift-marketplace**, manages the default custom resources (CRs) of the **OperatorHub**. The **OperatorHub** manages **CatalogSource** objects.

To apply persistent changes to **CatalogSource** object, you must first disable a default **CatalogSource** object.

#### Procedure

• To disable all the default **CatalogSource** objects at a local level, enter the following command:





#### NOTE

You can also configure the default **OperatorHub** CR to either disable all **CatalogSource** objects or disable a specific object.

#### Additional resources

- OperatorHub custom resource
- Disabling the default OperatorHub catalog sources

# 4.11.2. Overriding the node selector for catalog source pods

### **Prerequisites**

• A CatalogSource object of source type grpc with spec.image is defined.

#### Procedure

• Edit the **CatalogSource** object and add or modify the **spec.grpcPodConfig** section to include the following:

```
grpcPodConfig:
  nodeSelector:
  custom_label: <label>
```

where **<label>** is the label for the node selector that you want catalog source pods to use for scheduling.

# Additional resources

• Placing pods on specific nodes using node selectors

# 4.11.3. Overriding the priority class name for catalog source pods

### **Prerequisites**

• A CatalogSource object of source type grpc with spec.image is defined.

#### Procedure

 Edit the CatalogSource object and add or modify the spec.grpcPodConfig section to include the following:

```
grpcPodConfig:
  priorityClassName: <pri>cpriority_class>
```

where **<pri>priority\_class>** is one of the following:

- One of the default priority classes provided by Kubernetes: system-cluster-critical or system-node-critical
- An empty set ("") to assign the default priority
- A pre-existing and custom defined priority class



#### **NOTE**

Previously, the only pod scheduling parameter that could be overriden was **priorityClassName**. This was done by adding the **operatorframework.io/priorityclass** annotation to the **CatalogSource** object. For example:

apiVersion: operators.coreos.com/v1alpha1

kind: CatalogSource

metadata:

name: example-catalog

namespace: openshift-marketplace

annotations:

operatorframework.io/priorityclass: system-cluster-critical

If a **CatalogSource** object defines both the annotation and **spec.grpcPodConfig.priorityClassName**, the annotation takes precedence over the configuration parameter.

#### Additional resources

Pod priority classes

# 4.11.4. Overriding tolerations for catalog source pods

### **Prerequisites**

• A CatalogSource object of source type grpc with spec.image is defined.

#### Procedure

• Edit the **CatalogSource** object and add or modify the **spec.grpcPodConfig** section to include the following:

```
grpcPodConfig:
  tolerations:
    - key: "<key_name>"
    operator: "<operator_type>"
    value: "<value>"
    effect: "<effect>"
```

### Additional resources

Understanding taints and tolerations

# 4.12. MANAGING PLATFORM OPERATORS (TECHNOLOGY PREVIEW)

A platform Operator is an OLM-based Operator that can be installed during or after an OpenShift Container Platform cluster's Day 0 operations and participates in the cluster's lifecycle. As a cluster administrator, you can manage platform Operators by using the **PlatformOperator** API.



#### **IMPORTANT**

The platform Operator type 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.

# 4.12.1. About platform Operators

Operator Lifecycle Manager (OLM) introduces a new type of Operator called *platform Operators*. A platform Operator is an OLM-based Operator that can be installed during or after an OpenShift Container Platform cluster's Day 0 operations and participates in the cluster's lifecycle. As a cluster administrator, you can use platform Operators to further customize your OpenShift Container Platform installation to meet your requirements and use cases.

Using the existing cluster capabilities feature in OpenShift Container Platform, cluster administrators can already disable a subset of Cluster Version Operator-based (CVO) components considered non-essential to the initial payload prior to cluster installation. Platform Operators iterate on this model by providing additional customization options. Through the platform Operator mechanism, which relies on resources from the RukPak component, OLM-based Operators can now be installed at cluster installation time and can block cluster rollout if the Operator fails to install successfully.

In OpenShift Container Platform 4.12, this Technology Preview release focuses on the basic platform Operator mechanism and builds a foundation for expanding the concept in upcoming releases. You can use the cluster-wide **PlatformOperator** API to configure Operators before or after cluster creation on clusters that have enabled the **TechPreviewNoUpgrades** feature set.

### Additional resources

- RukPak component and packaging format
- Cluster capabilities

# 4.12.1.1. Technology Preview restrictions for platform Operators

During the Technology Preview release of the platform Operators feature in OpenShift Container Platform 4.12, the following restrictions determine whether an Operator can be installed through the platform Operators mechanism:

- Kubernetes manifests must be packaged using the Operator Lifecycle Manager (OLM)
   registry+v1 bundle format.
- The Operator cannot declare package or group/version/kind (GVK) dependencies.
- The Operator cannot specify cluster service version (CSV) install modes other than **AllNamespaces**
- The Operator cannot specify any **Webhook** or **APIService** definitions.
- All package bundles must be in the **redhat-operators** catalog source.

After considering these restrictions, the following Operators can be successfully installed:

Table 4.2. OLM-based Operators installable as platform Operators

3scale-operator	amq-broker-rhel8
amq-online	amq-streams
ansible-cloud-addons-operator	apicast-operator
container-security-operator	eap
file-integrity-operator	gatekeeper-operator-product
integration-operator	jws-operator
kiali-ossm	node-healthcheck-operator
odf-csi-addons-operator	odr-hub-operator
openshift-custom-metrics-autoscaler-operator	openshift-gitops-operator
openshift-pipelines-operator-rh	quay-operator
red-hat-camel-k	rhpam-kogito-operator
service-registry-operator	servicemeshoperator
skupper-operator	



# **NOTE**

The following features are not available during this Technology Preview release:

- Automatically upgrading platform Operator packages after cluster rollout
- Extending the platform Operator mechanism to support any optional, CVObased components

# 4.12.2. Prerequisites

- Access to an OpenShift Container Platform cluster using an account with **cluster-admin** permissions.
- The **TechPreviewNoUpgrades** feature set enabled on the cluster.



### **WARNING**

Enabling the **TechPreviewNoUpgrade** feature set cannot be undone and prevents minor version updates. These feature sets are not recommended on production clusters.

- Only the **redhat-operators** catalog source enabled on the cluster. This is a restriction during the Technology Preview release.
- The **oc** command installed on your workstation.

#### Additional resources

- Enabling features using feature gates
- Disabling the default OperatorHub catalog sources

# 4.12.3. Installing platform Operators during cluster creation

As a cluster administrator, you can install platform Operators by providing **FeatureGate** and **PlatformOperator** manifests during cluster creation.

#### **Procedure**

- 1. Choose a platform Operator from the supported set of OLM-based Operators. For the list of this set and details on current limitations, see "Technology Preview restrictions for platform Operators".
- 2. Select a cluster installation method and follow the instructions through creating an **install-config.yaml** file. For more details on preparing for a cluster installation, see "Selecting a cluster installation method and preparing it for users".
- 3. After you have created the **install-config.yaml** file and completed any modifications to it, change to the directory that contains the installation program and create the manifests:
  - \$ ./openshift-install create manifests --dir <installation\_directory> 1
  - 1 For <installation\_directory>, specify the name of the directory that contains the install-config.yaml file for your cluster.
- 4. Create a **FeatureGate** object YAML file in the **<installation\_directory>/manifests**/ directory that enables the **TechPreviewNoUpgrade** feature set, for example a **feature-gate.yaml** file:

### Example feature-gate.yaml file

apiVersion: config.openshift.io/v1 kind: FeatureGate metadata: annotations:

include.release.openshift.io/self-managed-high-availability: "true" include.release.openshift.io/single-node-developer: "true" release.openshift.io/create-only: "true"

name: cluster

spec:

featureSet: TechPreviewNoUpgrade 1



- Enable the **TechPreviewNoUpgrade** feature set.
- 5. Create a PlatformOperator object YAML file for your chosen platform Operator in the <installation\_directory>/manifests/ directory, for example a service-mesh-po.yaml file for the Red Hat OpenShift Service Mesh Operator:

# Example service-mesh-po.yaml file

apiVersion: platform.openshift.io/v1alpha1

kind: PlatformOperator

metadata:

name: service-mesh-po

spec: package:

name: servicemeshoperator

6. When you are ready to complete the cluster install, refer to your chosen installation method and continue through running the openshift-install create cluster command. During cluster creation, your provided manifests are used to enable the

**TechPreviewNoUpgrade** feature set and install your chosen platform Operator.



#### **IMPORTANT**

Failure of the platform Operator to successfully install will block the cluster installation process.

#### Verification

1. Check the status of the **service-mesh-po** platform Operator by running the following command:

\$ oc get platformoperator service-mesh-po -o yaml

# **Example output**

status:

activeBundleDeployment:

name: service-mesh-po

- lastTransitionTime: "2022-10-24T17:24:40Z"

message: Successfully applied the service-mesh-po BundleDeployment resource

reason: InstallSuccessful

status: "True" 1 type: Installed



Wait until the **Installed** status condition reports **True**.

2. Verify that the **platform-operators-aggregated** cluster Operator is reporting an **Available=True** status:

\$ oc get clusteroperator platform-operators-aggregated -o yaml

# Example output

...

#### status:

### conditions:

- lastTransitionTime: "2022-10-24T17:43:26Z"

message: All platform operators are in a successful state

reason: AsExpected status: "False" type: Progressing

- lastTransitionTime: "2022-10-24T17:43:26Z"

status: "False" type: Degraded

- lastTransitionTime: "2022-10-24T17:43:26Z"

message: All platform operators are in a successful state

reason: AsExpected status: "True" type: Available

#### Additional resources

- Selecting a cluster installation method and preparing it for users
- Technology Preview restrictions for platform Operators

# 4.12.4. Installing platform Operators after cluster creation

As a cluster administrator, you can install platform Operators after cluster creation on clusters that have enabled the **TechPreviewNoUpgrades** feature set by using the cluster-wide **PlatformOperator** API.

#### Procedure

- Choose a platform Operator from the supported set of OLM-based Operators. For the list of this set and details on current limitations, see "Technology Preview restrictions for platform Operators".
- 2. Create a **PlatformOperator** object YAML file for your chosen platform Operator, for example a **service-mesh-po.yaml** file for the Red Hat OpenShift Service Mesh Operator:

### Example sevice-mesh-po.yaml file

apiVersion: platform.openshift.io/v1alpha1

kind: PlatformOperator

metadata:

name: service-mesh-po

spec: package:

name: servicemeshoperator

3. Create the **PlatformOperator** object by running the following command:

\$ oc apply -f service-mesh-po.yaml



#### **NOTE**

If your cluster does not have the **TechPreviewNoUpgrades** feature set enabled, the object creation fails with the following message:

error: resource mapping not found for name: "service-mesh-po" namespace: "" from "service-mesh-po.yaml": no matches for kind "PlatformOperator" in version "platform.openshift.io/v1alpha1" ensure CRDs are installed first

#### Verification

 Check the status of the **service-mesh-po** platform Operator by running the following command:

\$ oc get platformoperator service-mesh-po -o yaml

# **Example output**

---

status:

activeBundleDeployment:

name: service-mesh-po

conditions:

- lastTransitionTime: "2022-10-24T17:24:40Z"

message: Successfully applied the service-mesh-po BundleDeployment resource

reason: InstallSuccessful

status: "True" 1 type: Installed

- Wait until the **Installed** status condition reports **True**.
- 2. Verify that the **platform-operators-aggregated** cluster Operator is reporting an **Available=True** status:

\$ oc get clusteroperator platform-operators-aggregated -o yaml

# **Example output**

...

status:

conditions:

- lastTransitionTime: "2022-10-24T17:43:26Z"

message: All platform operators are in a successful state

reason: AsExpected status: "False" type: Progressing

- lastTransitionTime: "2022-10-24T17:43:26Z"

status: "False" type: Degraded

- lastTransitionTime: "2022-10-24T17:43:26Z"

message: All platform operators are in a successful state

reason: AsExpected

status: "True" type: Available

#### Additional resources

Technology Preview restrictions for platform Operators

# 4.12.5. Deleting platform Operators

As a cluster administrator, you can delete existing platform Operators. Operator Lifecycle Manager (OLM) performs a cascading deletion. First, OLM removes the bundle deployment for the platform Operator, which then deletes any objects referenced in the **registry+v1** type bundle.



#### NOTE

The platform Operator manager and bundle deployment provisioner only manage objects that are referenced in the bundle, but not objects subsequently deployed by any bundle workloads themselves. For example, if a bundle workload creates a namespace and the Operator is not configured to clean it up before the Operator is removed, it is outside of the scope of OLM to remove the namespace during platform Operator deletion.

# **Procedure**

- 1. Get a list of installed platform Operators and find the name for the Operator you want to delete:
  - \$ oc get platformoperator
- 2. Delete the **PlatformOperator** resource for the chosen Operator, for example, for the Quay Operator:
  - \$ oc delete platformoperator quay-operator

# **Example output**

platformoperator.platform.openshift.io "quay-operator" deleted

#### Verification

1. Verify the namespace for the platform Operator is eventually deleted, for example, for the Quay Operator:

\$ oc get ns quay-operator-system

# Example output

Error from server (NotFound): namespaces "quay-operator-system" not found

2. Verify the **platform-operators-aggregated** cluster Operator continues to report an **Available=True** status:

\$ oc get co platform-operators-aggregated

# Example output

NAME VERSION AVAILABLE PROGRESSING DEGRADED SINCE MESSAGE platform-operators-aggregated 4.13.0-0 True False False 70s

# **CHAPTER 5. DEVELOPING OPERATORS**

# 5.1. ABOUT THE OPERATOR SDK

The Operator Framework is an open source toolkit to manage Kubernetes native applications, called *Operators*, in an effective, automated, and scalable way. Operators take advantage of Kubernetes extensibility to deliver the automation advantages of cloud services, like provisioning, scaling, and backup and restore, while being able to run anywhere that Kubernetes can run.

Operators make it easy to manage complex, stateful applications on top of Kubernetes. However, writing an Operator today can be difficult because of challenges such as using low-level APIs, writing boilerplate, and a lack of modularity, which leads to duplication.

The Operator SDK, a component of the Operator Framework, provides a command-line interface (CLI) tool that Operator developers can use to build, test, and deploy an Operator.

### Why use the Operator SDK?

The Operator SDK simplifies this process of building Kubernetes-native applications, which can require deep, application-specific operational knowledge. The Operator SDK not only lowers that barrier, but it also helps reduce the amount of boilerplate code required for many common management capabilities, such as metering or monitoring.

The Operator SDK is a framework that uses the controller-runtime library to make writing Operators easier by providing the following features:

- High-level APIs and abstractions to write the operational logic more intuitively
- Tools for scaffolding and code generation to quickly bootstrap a new project
- Integration with Operator Lifecycle Manager (OLM) to streamline packaging, installing, and running Operators on a cluster
- Extensions to cover common Operator use cases
- Metrics set up automatically in any generated Go-based Operator for use on clusters where the Prometheus Operator is deployed

Operator authors with cluster administrator access to a Kubernetes-based cluster (such as OpenShift Container Platform) can use the Operator SDK CLI to develop their own Operators based on Go, Ansible, Java, or Helm. Kubebuilder is embedded into the Operator SDK as the scaffolding solution for Go-based Operators, which means existing Kubebuilder projects can be used as is with the Operator SDK and continue to work.



### **NOTE**

OpenShift Container Platform 4.13 supports Operator SDK 1.28.0.

# 5.1.1. What are Operators?

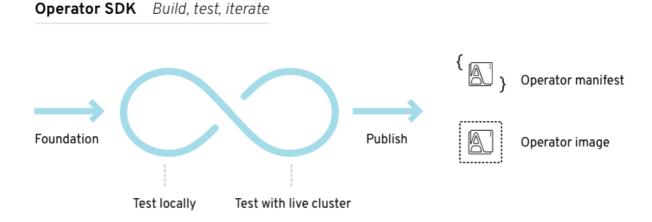
For an overview about basic Operator concepts and terminology, see Understanding Operators.

# 5.1.2. Development workflow

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

- 1. Create an Operator project by using the Operator SDK command-line interface (CLI).
- 2. Define new resource APIs by adding custom resource definitions (CRDs).
- 3. Specify resources to watch by using the Operator SDK API.
- 4. Define the Operator reconciling logic in a designated handler and use the Operator SDK API to interact with resources.
- 5. Use the Operator SDK CLI to build and generate the Operator deployment manifests.

Figure 5.1. Operator SDK workflow



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

# 5.1.3. Additional resources

• Certified Operator Build Guide

# 5.2. INSTALLING THE OPERATOR SDK CLI

The Operator SDK provides a command-line interface (CLI) tool that Operator developers can use to build, test, and deploy an Operator. You can install the Operator SDK CLI on your workstation so that you are prepared to start authoring your own Operators.

Operator authors with cluster administrator access to a Kubernetes-based cluster, such as OpenShift Container Platform, can use the Operator SDK CLI to develop their own Operators based on Go, Ansible, Java, or Helm. Kubebuilder is embedded into the Operator SDK as the scaffolding solution for Go-based Operators, which means existing Kubebuilder projects can be used as is with the Operator SDK and continue to work.



#### NOTE

OpenShift Container Platform 4.13 supports Operator SDK 1.28.0.

# 5.2.1. Installing the Operator SDK CLI on Linux

You can install the OpenShift SDK CLI tool on Linux.

## **Prerequisites**

- Go v1.19+
- docker v17.03+, podman v1.9.3+, or buildah v1.7+

#### **Procedure**

- 1. Navigate to the OpenShift mirror site.
- 2. From the latest 4.13 directory, download the latest version of the tarball for Linux.
- 3. Unpack the archive:
  - \$ tar xvf operator-sdk-v1.28.0-ocp-linux-x86\_64.tar.gz
- 4. Make the file executable:
  - \$ chmod +x operator-sdk
- 5. Move the extracted **operator-sdk** binary to a directory that is on your **PATH**.

### TIP

To check your **PATH**:

\$ echo \$PATH

\$ sudo mv ./operator-sdk /usr/local/bin/operator-sdk

#### Verification

- After you install the Operator SDK CLI, verify that it is available:
  - \$ operator-sdk version

# **Example output**

operator-sdk version: "v1.28.0-ocp", ...

# 5.2.2. Installing the Operator SDK CLI on macOS

You can install the OpenShift SDK CLI tool on macOS.

## **Prerequisites**

- Go v1.19+
- docker v17.03+, podman v1.9.3+, or buildah v1.7+

### Procedure

- For the amd64 and arm64 architectures, navigate to the OpenShift mirror site for the amd64 architecture and OpenShift mirror site for the arm64 architecture respectively.
- 2. From the latest 4.13 directory, download the latest version of the tarball for macOS.
- 3. Unpack the Operator SDK archive for **amd64** architecture by running the following command:
  - \$ tar xvf operator-sdk-v1.28.0-ocp-darwin-x86\_64.tar.gz
- 4. Unpack the Operator SDK archive for **arm64** architecture by running the following command:
  - \$ tar xvf operator-sdk-v1.28.0-ocp-darwin-aarch64.tar.gz
- 5. Make the file executable by running the following command:
  - \$ chmod +x operator-sdk
- 6. Move the extracted **operator-sdk** binary to a directory that is on your **PATH** by running the following command:

#### TIP

Check your **PATH** by running the following command:

- \$ echo \$PATH
- \$ sudo mv ./operator-sdk /usr/local/bin/operator-sdk

# Verification

- After you install the Operator SDK CLI, verify that it is available by running the following command::
  - \$ operator-sdk version

# **Example output**

operator-sdk version: "v1.28.0-ocp", ...

### 5.3. GO-BASED OPERATORS

# 5.3.1. Getting started with Operator SDK for Go-based Operators

To demonstrate the basics of setting up and running a Go-based Operator using tools and libraries provided by the Operator SDK, Operator developers can build an example Go-based Operator for Memcached, a distributed key-value store, and deploy it to a cluster.

### 5.3.1.1. Prerequisites

Operator SDK CLI installed

- OpenShift CLI (oc) v4.13+ installed
- Go v1.19+
- Logged into an OpenShift Container Platform 4.13 cluster with **oc** with an account that has **cluster-admin** permissions
- To allow the cluster to pull the image, the repository where you push your image must be set as public, or you must configure an image pull secret

#### Additional resources

- Installing the Operator SDK CLI
- Getting started with the OpenShift CLI

# 5.3.1.2. Creating and deploying Go-based Operators

You can build and deploy a simple Go-based Operator for Memcached by using the Operator SDK.

#### **Procedure**

- 1. Create a project.
  - a. Create your project directory:
    - \$ mkdir memcached-operator
  - b. Change into the project directory:
    - \$ cd memcached-operator
  - c. Run the **operator-sdk init** command to initialize the project:
    - \$ operator-sdk init \
      - --domain=example.com \
      - --repo=github.com/example-inc/memcached-operator

The command uses the Go plugin by default.

#### 2. Create an API.

Create a simple Memcached API:

\$ operator-sdk create api \

- --resource=true \
- --controller=true \
- --group cache  $\$
- --version v1 \
- --kind Memcached

# 3. Build and push the Operator image.

Use the default **Makefile** targets to build and push your Operator. Set **IMG** with a pull spec for your image that uses a registry you can push to:

\$ make docker-build docker-push IMG=<registry>/<user>/<image\_name>:<tag>

### 4. Run the Operator.

- a. Install the CRD:
  - \$ make install
- b. Deploy the project to the cluster. Set **IMG** to the image that you pushed:
  - \$ make deploy IMG=<registry>/<user>/<image\_name>:<tag>
- 5. Create a sample custom resource (CR).
  - a. Create a sample CR:
    - \$ oc apply -f config/samples/cache\_v1\_memcached.yaml \ -n memcached-operator-system
  - b. Watch for the CR to reconcile the Operator:
    - $\$  oc logs deployment.apps/memcached-operator-controller-manager  $\$ 
      - -c manager \
      - -n memcached-operator-system

### 6. Delete a CR.

Delete a CR by running the following command:

\$ oc delete -f config/samples/cache\_v1\_memcached -n memcached-operator-system

### 7. Clean up.

Run the following command to clean up the resources that have been created as part of this procedure:

\$ make undeploy

## 5.3.1.3. Next steps

 See Operator SDK tutorial for Go-based Operators for a more in-depth walkthrough on building a Go-based Operator.

# 5.3.2. Operator SDK tutorial for Go-based Operators

Operator developers can take advantage of Go programming language support in the Operator SDK to build an example Go-based Operator for Memcached, a distributed key-value store, and manage its lifecycle.

This process is accomplished using two centerpieces of the Operator Framework:

# **Operator SDK**

The operator-sdk CLI tool and controller-runtime library API

Operator Lifecycle Manager (OLM)

Installation, upgrade, and role-based access control (RBAC) of Operators on a cluster



#### NOTE

This tutorial goes into greater detail than Getting started with Operator SDK for Gobased Operators.

# 5.3.2.1. Prerequisites

- Operator SDK CLI installed
- OpenShift CLI (oc) v4.13+ installed
- Go v1.19+
- Logged into an OpenShift Container Platform 4.13 cluster with oc with an account that has cluster-admin permissions
- To allow the cluster to pull the image, the repository where you push your image must be set as public, or you must configure an image pull secret

#### Additional resources

- Installing the Operator SDK CLI
- Getting started with the OpenShift CLI

# 5.3.2.2. Creating a project

Use the Operator SDK CLI to create a project called **memcached-operator**.

### Procedure

- 1. Create a directory for the project:
  - \$ mkdir -p \$HOME/projects/memcached-operator
- 2. Change to the directory:
  - \$ cd \$HOME/projects/memcached-operator
- 3. Activate support for Go modules:
  - \$ export GO111MODULE=on
- 4. Run the **operator-sdk init** command to initialize the project:
  - \$ operator-sdk init \
    - --domain=example.com \
    - --repo=github.com/example-inc/memcached-operator



### **NOTE**

The operator-sdk init command uses the Go plugin by default.

The **operator-sdk init** command generates a **go.mod** file to be used with **Go modules**. The **-repo** flag is required when creating a project outside of **\$GOPATH/src/**, because generated files require a valid module path.

#### 5.3.2.2.1. PROJECT file

Among the files generated by the **operator-sdk init** command is a Kubebuilder **PROJECT** file. Subsequent **operator-sdk** commands, as well as **help** output, that are run from the project root read this file and are aware that the project type is Go. For example:

```
domain: example.com
layout:
- go.kubebuilder.io/v3
projectName: memcached-operator
repo: github.com/example-inc/memcached-operator
version: "3"
plugins:
    manifests.sdk.operatorframework.io/v2: {}
    scorecard.sdk.operatorframework.io/v2: {}
    sdk.x-openshift.io/v1: {}
```

### 5.3.2.2.2. About the Manager

The main program for the Operator is the **main.go** file, which initializes and runs the Manager. The Manager automatically registers the Scheme for all custom resource (CR) API definitions and sets up and runs controllers and webhooks.

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

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

By default, the Manager watches the namespace where the Operator runs. To watch all namespaces, you can leave the **namespace** option empty:

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

You can also use the MultiNamespacedCacheBuilder function to watch a specific set of namespaces:

```
var namespaces []string 1
mgr, err := ctrl.NewManager(cfg, manager.Options{ 2
    NewCache: cache.MultiNamespacedCacheBuilder(namespaces),
})
```

- List of namespaces.
- Creates a **Cmd** struct to provide shared dependencies and start components.

### 5.3.2.2.3. About multi-group APIs

Before you create an API and controller, consider whether your Operator requires multiple API groups. This tutorial covers the default case of a single group API, but to change the layout of your project to support multi-group APIs, you can run the following command:

\$ operator-sdk edit --multigroup=true

This command updates the **PROJECT** file, which should look like the following example:

```
domain: example.com
layout: go.kubebuilder.io/v3
multigroup: true
...
```

For multi-group projects, the API Go type files are created in the **apis**/**<group**>/**<version>**/ directory, and the controllers are created in the **controllers**/**<group>**/ directory. The Dockerfile is then updated accordingly.

#### Additional resource

• For more details on migrating to a multi-group project, see the Kubebuilder documentation.

# 5.3.2.3. Creating an API and controller

Use the Operator SDK CLI to create a custom resource definition (CRD) API and controller.

#### **Procedure**

 Run the following command to create an API with group cache, version, v1, and kind Memcached:

```
$ operator-sdk create api \
--group=cache \
--version=v1 \
--kind=Memcached
```

2. When prompted, enter **y** for creating both the resource and controller:

```
Create Resource [y/n]
y
Create Controller [y/n]
y
```

# Example output

```
Writing scaffold for you to edit... api/v1/memcached_types.go controllers/memcached_controller.go ...
```

This process generates the **Memcached** resource API at **api/v1/memcached\_types.go** and the controller at **controllers/memcached controller.go**.

#### 5.3.2.3.1. Defining the API

Define the API for the **Memcached** custom resource (CR).

#### **Procedure**

 Modify the Go type definitions at api/v1/memcached\_types.go to have the following spec and status:

```
// MemcachedSpec defines the desired state of Memcached
type MemcachedSpec struct {
    // +kubebuilder:validation:Minimum=0
    // Size is the size of the memcached deployment
    Size int32 `json:"size"`
}

// MemcachedStatus defines the observed state of Memcached
type MemcachedStatus struct {
    // Nodes are the names of the memcached pods
    Nodes []string `json:"nodes"`
}
```

2. Update the generated code for the resource type:

\$ make generate

### TIP

After you modify a \*\_types.go file, you must run the **make generate** command to update the generated code for that resource type.

The above Makefile target invokes the **controller-gen** utility to update the **api/v1/zz\_generated.deepcopy.go** file. This ensures your API Go type definitions implement the **runtime.Object** interface that all Kind types must implement.

## 5.3.2.3.2. Generating CRD manifests

After the API is defined with **spec** and **status** fields and custom resource definition (CRD) validation markers, you can generate CRD manifests.

#### **Procedure**

Run the following command to generate and update CRD manifests:

\$ make manifests

This Makefile target invokes the **controller-gen** utility to generate the CRD manifests in the **config/crd/bases/cache.example.com\_memcacheds.yaml** file.

### 5.3.2.3.2.1. About OpenAPI validation

OpenAPIv3 schemas are added to CRD manifests in the **spec.validation** block when the manifests are generated. This validation block allows Kubernetes to validate the properties in a Memcached custom resource (CR) when it is created or updated.

Markers, or annotations, are available to configure validations for your API. These markers always have a **+kubebuilder:validation** prefix.

#### Additional resources

- For more details on the usage of markers in API code, see the following Kubebuilder documentation:
  - CRD generation
  - Markers
  - List of OpenAPIv3 validation markers
- For more details about OpenAPIv3 validation schemas in CRDs, see the Kubernetes documentation.

## 5.3.2.4. Implementing the controller

After creating a new API and controller, you can implement the controller logic.

#### Procedure

• For this example, replace the generated controller file **controllers/memcached\_controller.go** with following example implementation:

# Example 5.1. Example memcached\_controller.go

You may obtain a copy of the License at

```
/*
Copyright 2020.

Licensed under the Apache License, Version 2.0 (the "License");
you may not use this file except in compliance with the License.
```

http://www.apache.org/licenses/LICENSE-2.0

Unless required by applicable law or agreed to in writing, software distributed under the License is distributed on an "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied. See the License for the specific language governing permissions and limitations under the License.

```
package controllers
```

```
import (
     appsv1 "k8s.io/api/apps/v1"
     corev1 "k8s.io/api/core/v1"
     "k8s.io/apimachinery/pkg/api/errors"
     metav1 "k8s.io/apimachinery/pkg/apis/meta/v1"
     "k8s.io/apimachinery/pkg/types"
     "reflect"

"context"
```

```
"github.com/go-logr/logr"
     "k8s.io/apimachinery/pkg/runtime"
     ctrl "sigs.k8s.io/controller-runtime"
     "sigs.k8s.io/controller-runtime/pkg/client"
     ctrllog "sigs.k8s.io/controller-runtime/pkg/log"
     cachev1 "github.com/example-inc/memcached-operator/api/v1"
)
// MemcachedReconciler reconciles a Memcached object
type MemcachedReconciler struct {
     client.Client
     Log logr.Logger
     Scheme *runtime.Scheme
}
+kubebuilder:rbac:groups=cache.example.com,resources=memcacheds,verbs=get;list;watcl
;create;update;patch;delete
+kubebuilder:rbac:groups=cache.example.com,resources=memcacheds/status,verbs=get;ur
date;patch
+kubebuilder:rbac:groups=cache.example.com,resources=memcacheds/finalizers,verbs=upi
+kubebuilder:rbac:groups=apps,resources=deployments,verbs=get;list;watch;create;update;
patch;delete
// +kubebuilder:rbac:groups=core,resources=pods,verbs=get;list;
// Reconcile is part of the main kubernetes reconciliation loop which aims to
// move the current state of the cluster closer to the desired state.
// TODO(user): Modify the Reconcile function to compare the state specified by
// the Memcached object against the actual cluster state, and then
// perform operations to make the cluster state reflect the state specified by
// the user.
// For more details, check Reconcile and its Result here:
// - https://pkg.go.dev/sigs.k8s.io/controller-runtime@v0.7.0/pkg/reconcile
func (r *MemcachedReconciler) Reconcile(ctx context.Context, req ctrl.Request)
(ctrl.Result, error) {
     //log := r.Log.WithValues("memcached", req.NamespacedName)
     log := ctrllog.FromContext(ctx)
    // Fetch the Memcached instance
     memcached := &cachev1.Memcached{}
     err := r.Get(ctx, req.NamespacedName, memcached)
     if err != nil {
          if errors.IsNotFound(err) {
              // Request object not found, could have been deleted after reconcile
request.
              // Owned objects are automatically garbage collected. For additional
cleanup logic use finalizers.
              // Return and don't requeue
              log.Info("Memcached resource not found. Ignoring since object must be
deleted")
              return ctrl.Result{}, nil
```

```
// Error reading the object - requeue the request.
         log.Error(err, "Failed to get Memcached")
         return ctrl.Result{}, err
    }
    // Check if the deployment already exists, if not create a new one
    found := &appsv1.Deployment{}
    err = r.Get(ctx, types.NamespacedName{Name: memcached.Name, Namespace:
memcached.Namespace}, found)
    if err != nil && errors.lsNotFound(err) {
         // Define a new deployment
         dep := r.deploymentForMemcached(memcached)
         log.Info("Creating a new Deployment", "Deployment.Namespace",
dep.Namespace, "Deployment.Name", dep.Name)
         err = r.Create(ctx, dep)
         if err != nil {
              log.Error(err, "Failed to create new Deployment",
"Deployment.Namespace", dep.Namespace, "Deployment.Name", dep.Name)
              return ctrl.Result{}, err
         }
         // Deployment created successfully - return and requeue
         return ctrl.Result{Requeue: true}, nil
    } else if err != nil {
         log.Error(err, "Failed to get Deployment")
         return ctrl.Result{}, err
    }
    // Ensure the deployment size is the same as the spec
    size := memcached.Spec.Size
    if *found.Spec.Replicas != size {
         found.Spec.Replicas = &size
         err = r.Update(ctx, found)
         if err != nil {
              log.Error(err, "Failed to update Deployment", "Deployment.Namespace",
found.Namespace, "Deployment.Name", found.Name)
              return ctrl.Result{}, err
         }
         // Spec updated - return and requeue
         return ctrl.Result{Requeue: true}, nil
    }
    // Update the Memcached status with the pod names
    // List the pods for this memcached's deployment
    podList := &corev1.PodList{}
    listOpts := []client.ListOption{
         client.InNamespace(memcached.Namespace),
         client.MatchingLabels(labelsForMemcached(memcached.Name)),
    if err = r.List(ctx, podList, listOpts...); err != nil {
         log.Error(err, "Failed to list pods", "Memcached.Namespace",
memcached.Namespace, "Memcached.Name", memcached.Name)
         return ctrl.Result{}, err
     podNames := getPodNames(podList.Items)
```

```
// Update status.Nodes if needed
    if !reflect.DeepEqual(podNames, memcached.Status.Nodes) {
         memcached.Status.Nodes = podNames
         err := r.Status().Update(ctx, memcached)
         if err != nil {
              log.Error(err, "Failed to update Memcached status")
              return ctrl.Result{}, err
         }
    return ctrl.Result{}, nil
}
// deploymentForMemcached returns a memcached Deployment object
func (r *MemcachedReconciler) deploymentForMemcached(m *cachev1.Memcached)
*appsv1.Deployment {
    ls := labelsForMemcached(m.Name)
    replicas := m.Spec.Size
    dep := &appsv1.Deployment{
         ObjectMeta: metav1.ObjectMeta{
              Name:
                        m.Name,
              Namespace: m.Namespace,
         Spec: appsv1.DeploymentSpec{
              Replicas: &replicas,
              Selector: &metav1.LabelSelector{
                   MatchLabels: ls,
              Template: corev1.PodTemplateSpec{
                   ObjectMeta: metav1.ObjectMeta{
                       Labels: ls,
                   Spec: corev1.PodSpec{
                        Containers: []corev1.Container{{
                            Image: "memcached:1.4.36-alpine",
                            Name: "memcached",
                            Command: []string{"memcached", "-m=64", "-o", "modern",
"-v"}.
                            Ports: []corev1.ContainerPort{{
                                 ContainerPort: 11211,
                                 Name: "memcached",
                            }},
                       }},
                  },
              },
         },
    // Set Memcached instance as the owner and controller
    ctrl.SetControllerReference(m, dep, r.Scheme)
    return dep
}
// labelsForMemcached returns the labels for selecting the resources
// belonging to the given memcached CR name.
func labelsForMemcached(name string) map[string]string {
```

```
return map[string]string{"app": "memcached", "memcached_cr": name}

// getPodNames returns the pod names of the array of pods passed in

func getPodNames(pods []corev1.Pod) []string {
    var podNames []string
    for _, pod := range pods {
        podNames = append(podNames, pod.Name)
    }
    return podNames

// SetupWithManager sets up the controller with the Manager.

func (r *MemcachedReconciler) SetupWithManager(mgr ctrl.Manager) error {
    return ctrl.NewControllerManagedBy(mgr).
        For(&cachev1.Memcached{}).
        Owns(&appsv1.Deployment{}).
        Complete(r)
}
```

The example controller runs the following reconciliation logic for each **Memcached** custom resource (CR):

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

The next subsections explain how the controller in the example implementation watches resources and how the reconcile loop is triggered. You can skip these subsections to go directly to Running the Operator.

### 5.3.2.4.1. Resources watched by the controller

The **SetupWithManager()** function in **controllers/memcached\_controller.go** specifies how the controller is built to watch a CR and other resources that are owned and managed by that controller.

```
import (
...
appsv1 "k8s.io/api/apps/v1"
...
)

func (r *MemcachedReconciler) SetupWithManager(mgr ctrl.Manager) error {
  return ctrl.NewControllerManagedBy(mgr).
  For(&cachev1.Memcached{}).
  Owns(&appsv1.Deployment{}).
  Complete(r)
}
```

**NewControllerManagedBy()** provides a controller builder that allows various controller configurations.

For(&cachev1.Memcached{}) specifies the Memcached type as the primary resource to watch. For each Add, Update, or Delete event for a Memcached type, the reconcile loop is sent a reconcile Request argument, which consists of a namespace and name key, for that Memcached object.

Owns(&appsv1.Deployment{}) specifies the **Deployment** type as the secondary resource to watch. For each **Deployment** type Add, Update, or Delete event, the event handler maps each event to a reconcile request for the owner of the deployment. In this case, the owner is the **Memcached** object for which the deployment was created.

# 5.3.2.4.2. Controller configurations

You can initialize a controller by using many other useful configurations. For example:

• Set the maximum number of concurrent reconciles for the controller by using the **MaxConcurrentReconciles** option, which defaults to **1**:

```
func (r *MemcachedReconciler) SetupWithManager(mgr ctrl.Manager) error {
    return ctrl.NewControllerManagedBy(mgr).
        For(&cachev1.Memcached{}).
        Owns(&appsv1.Deployment{}).
        WithOptions(controller.Options{
                MaxConcurrentReconciles: 2,
        }).
        Complete(r)
}
```

- Filter watch events using predicates.
- Choose the type of EventHandler to change how a watch event translates to reconcile requests for the reconcile loop. For Operator relationships that are more complex than primary and secondary resources, you can use the **EnqueueRequestsFromMapFunc** handler to transform a watch event into an arbitrary set of reconcile requests.

For more details on these and other configurations, see the upstream Builder and Controller GoDocs.

### 5.3.2.4.3. Reconcile loop

Every controller has a reconciler object with a **Reconcile()** method that implements the reconcile loop. The reconcile loop is passed the **Request** argument, which is a namespace and name key used to find the primary resource object, **Memcached**, from the cache:

```
import (
  ctrl "sigs.k8s.io/controller-runtime"

cachev1 "github.com/example-inc/memcached-operator/api/v1"
...
)

func (r *MemcachedReconciler) Reconcile(ctx context.Context, req ctrl.Request) (ctrl.Result, error) {
    // Lookup the Memcached instance for this reconcile request
    memcached := &cachev1.Memcached{}
    err := r.Get(ctx, req.NamespacedName, memcached)
    ...
}
```

Based on the return values, result, and error, the request might be requeued and the reconcile loop might be triggered again:

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

You can set the **Result.RequeueAfter** to requeue the request after a grace period as well:

```
import "time"

// Reconcile for any reason other than an error after 5 seconds
return ctrl.Result{RequeueAfter: time.Second*5}, nil
```



#### **NOTE**

You can return **Result** with **RequeueAfter** set to periodically reconcile a CR.

For more on reconcilers, clients, and interacting with resource events, see the Controller Runtime Client API documentation.

#### 5.3.2.4.4. Permissions and RBAC manifests

The controller requires certain RBAC permissions to interact with the resources it manages. These are specified using RBAC markers, such as the following:

```
//
+kubebuilder:rbac:groups=cache.example.com,resources=memcacheds,verbs=get;list;watch;create;upc
ate;patch;delete
//
+kubebuilder:rbac:groups=cache.example.com,resources=memcacheds/status,verbs=get;update;patch
// +kubebuilder:rbac:groups=cache.example.com,resources=memcacheds/finalizers,verbs=update
//
+kubebuilder:rbac:groups=apps,resources=deployments,verbs=get;list;watch;create;update;patch;delete
// +kubebuilder:rbac:groups=core,resources=pods,verbs=get;list;

func (r *MemcachedReconciler) Reconcile(ctx context.Context, req ctrl.Request) (ctrl.Result, error) {
...
}
```

The **ClusterRole** object manifest at **config/rbac/role.yaml** is generated from the previous markers by using the **controller-gen** utility whenever the **make manifests** command is run.

## 5.3.2.5. Enabling proxy support

Operator authors can develop Operators that support network proxies. Cluster administrators configure proxy support for the environment variables that are handled by Operator Lifecycle Manager (OLM). To support proxied clusters, your Operator must inspect the environment for the following standard proxy

variables and pass the values to Operands:

- HTTP\_PROXY
- HTTPS\_PROXY
- NO\_PROXY



### NOTE

This tutorial uses **HTTP\_PROXY** as an example environment variable.

### **Prerequisites**

• A cluster with cluster-wide egress proxy enabled.

### **Procedure**

- 1. Edit the **controllers/memcached\_controller.go** file to include the following:
  - a. Import the **proxy** package from the **operator-lib** library:

```
import (
...
"github.com/operator-framework/operator-lib/proxy"
)
```

b. Add the **proxy.ReadProxyVarsFromEnv** helper function to the reconcile loop and append the results to the Operand environments:

```
for i, container := range dep.Spec.Template.Spec.Containers {
   dep.Spec.Template.Spec.Containers[i].Env = append(container.Env,
   proxy.ReadProxyVarsFromEnv()...)
}
...
```

2. Set the environment variable on the Operator deployment by adding the following to the **config/manager/manager.yaml** file:

```
containers:
- args:
- --leader-elect
- --leader-election-id=ansible-proxy-demo image: controller:latest name: manager env:
- name: "HTTP_PROXY" value: "http_proxy_test"
```

## 5.3.2.6. Running the Operator

There are three ways you can use the Operator SDK CLI to build and run your Operator:

• Run locally outside the cluster as a Go program.

- Run as a deployment on the cluster.
- Bundle your Operator and use Operator Lifecycle Manager (OLM) to deploy on the cluster.



### NOTE

Before running your Go-based Operator as either a deployment on OpenShift Container Platform or as a bundle that uses OLM, ensure that your project has been updated to use supported images.

### 5.3.2.6.1. Running locally outside the cluster

You can run your Operator project as a Go program outside of the cluster. This is useful for development purposes to speed up deployment and testing.

#### **Procedure**

 Run the following command to install the custom resource definitions (CRDs) in the cluster configured in your ~/.kube/config file and run the Operator locally:

\$ make install run

# **Example output**

...

2021-01-10T21:09:29.016-0700 INFO controller-runtime.metrics metrics server is starting to listen {"addr": ":8080"}

2021-01-10T21:09:29.017-0700 INFO setup starting manager

2021-01-10T21:09:29.017-0700 INFO controller-runtime.manager starting metrics server {"path": "/metrics"}

2021-01-10T21:09:29.018-0700 INFO controller-runtime.manager.controller.memcached Starting EventSource {"reconciler group": "cache.example.com", "reconciler kind":

"Memcached", "source": "kind source: /, Kind="}

2021-01-10T21:09:29.218-0700 INFO controller-runtime.manager.controller.memcached Starting Controller {"reconciler group": "cache.example.com", "reconciler kind": "Memcached"}

2021-01-10T21:09:29.218-0700 INFO controller-runtime.manager.controller.memcached Starting workers {"reconciler group": "cache.example.com", "reconciler kind": "Memcached", "worker count": 1}

### 5.3.2.6.2. Running as a deployment on the cluster

You can run your Operator project as a deployment on your cluster.

# **Prerequisites**

 Prepared your Go-based Operator to run on OpenShift Container Platform by updating the project to use supported images

### Procedure

1. Run the following **make** commands to build and push the Operator image. Modify the **IMG** argument in the following steps to reference a repository that you have access to. You can obtain an account for storing containers at repository sites such as Quay.io.

a. Build the image:

\$ make docker-build IMG=<registry>/<user>/<image\_name>:<tag>



#### **NOTE**

The Dockerfile generated by the SDK for the Operator explicitly references **GOARCH=amd64** for **go build**. This can be amended to **GOARCH=\$TARGETARCH** for non-AMD64 architectures. Docker will automatically set the environment variable to the value specified by – **platform**. With Buildah, the –**build-arg** will need to be used for the purpose. For more information, see Multiple Architectures.

b. Push the image to a repository:

\$ make docker-push IMG=<registry>/<user>/<image\_name>:<tag>



#### NOTE

The name and tag of the image, for example **IMG=** <registry>/<user>/<image\_name>:<tag>, in both the commands can also be set in your Makefile. Modify the **IMG ?= controller:latest** value to set your default image name.

2. Run the following command to deploy the Operator:

\$ make deploy IMG=<registry>/<user>/<image\_name>:<tag>

By default, this command creates a namespace with the name of your Operator project in the form **cproject\_name>-system and is used for the deployment. This command also installs the RBAC manifests from <b>config/rbac**.

3. Run the following command to verify that the Operator is running:

### **Example output**

## 5.3.2.6.3. Bundling an Operator and deploying with Operator Lifecycle Manager

## 5.3.2.6.3.1. Bundling an Operator

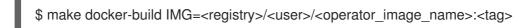
The Operator bundle format is the default packaging method for Operator SDK and Operator Lifecycle Manager (OLM). You can get your Operator ready for use on OLM by using the Operator SDK to build and push your Operator project as a bundle image.

## **Prerequisites**

- Operator SDK CLI installed on a development workstation
- OpenShift CLI (oc) v4.13+ installed
- Operator project initialized by using the Operator SDK
- If your Operator is Go-based, your project must be updated to use supported images for running on OpenShift Container Platform

#### **Procedure**

- 1. Run the following **make** commands in your Operator project directory to build and push your Operator image. Modify the **IMG** argument in the following steps to reference a repository that you have access to. You can obtain an account for storing containers at repository sites such as Quay.io.
  - a. Build the image:





#### NOTE

The Dockerfile generated by the SDK for the Operator explicitly references **GOARCH=amd64** for **go build**. This can be amended to **GOARCH=\$TARGETARCH** for non-AMD64 architectures. Docker will automatically set the environment variable to the value specified by – **platform**. With Buildah, the –**build-arg** will need to be used for the purpose. For more information, see Multiple Architectures.

- b. Push the image to a repository:
  - \$ make docker-push IMG=<registry>/<user>/<operator\_image\_name>:<tag>
- Create your Operator bundle manifest by running the make bundle command, which invokes several commands, including the Operator SDK generate bundle and bundle validate subcommands:
  - \$ make bundle IMG=<registry>/<user>/<operator\_image\_name>:<tag>

Bundle manifests for an Operator describe how to display, create, and manage an application. The **make bundle** command creates the following files and directories in your Operator project:

- A bundle manifests directory named bundle/manifests that contains a ClusterServiceVersion object
- A bundle metadata directory named **bundle/metadata**
- All custom resource definitions (CRDs) in a config/crd directory
- A Dockerfile bundle.Dockerfile

These files are then automatically validated by using **operator-sdk bundle validate** to ensure the on-disk bundle representation is correct.

- 3. Build and push your bundle image by running the following commands. OLM consumes Operator bundles using an index image, which reference one or more bundle images.
  - a. Build the bundle image. Set **BUNDLE\_IMG** with the details for the registry, user namespace, and image tag where you intend to push the image:
    - \$ make bundle-build BUNDLE\_IMG=<registry>/<user>/<bundle\_image\_name>:<tag>
  - b. Push the bundle image:
    - \$ docker push <registry>/<user>/<bundle\_image\_name>:<tag>

# 5.3.2.6.3.2. Deploying an Operator with Operator Lifecycle Manager

Operator Lifecycle Manager (OLM) helps you to install, update, and manage the lifecycle of Operators and their associated services on a Kubernetes cluster. OLM is installed by default on OpenShift Container Platform and runs as a Kubernetes extension so that you can use the web console and the OpenShift CLI (oc) for all Operator lifecycle management functions without any additional tools.

The Operator bundle format is the default packaging method for Operator SDK and OLM. You can use the Operator SDK to quickly run a bundle image on OLM to ensure that it runs properly.

### **Prerequisites**

- Operator SDK CLI installed on a development workstation
- Operator bundle image built and pushed to a registry
- OLM installed on a Kubernetes-based cluster (v1.16.0 or later if you use **apiextensions.k8s.io/v1** CRDs, for example OpenShift Container Platform 4.13)
- Logged in to the cluster with oc using an account with cluster-admin permissions
- If your Operator is Go-based, your project must be updated to use supported images for running on OpenShift Container Platform

### Procedure

• Enter the following command to run the Operator on the cluster:

\$ operator-sdk run bundle \1
-n <namespace> \2
<registry>/<user>/<bundle\_image\_name>:<tag> 3

- The **run bundle** command creates a valid file-based catalog and installs the Operator bundle on your cluster using OLM.
- Optional: By default, the command installs the Operator in the currently active project in your ~/.kube/config file. You can add the -n flag to set a different namespace scope for the installation.
- If you do not specify an image, the command uses **quay.io/operator-framework/opm:latest** as the default index image. If you specify an image, the command uses the bundle image itself as the index image.



### **IMPORTANT**

As of OpenShift Container Platform 4.11, the **run bundle** command supports the file-based catalog format for Operator catalogs by default. The deprecated SQLite database format for Operator catalogs continues to be supported; however, it will be removed in a future release. It is recommended that Operator authors migrate their workflows to the file-based catalog format.

This command performs the following actions:

- Create an index image referencing your bundle image. The index image is opaque and ephemeral, but accurately reflects how a bundle would be added to a catalog in production.
- Create a catalog source that points to your new index image, which enables OperatorHub to discover your Operator.
- Deploy your Operator to your cluster by creating an OperatorGroup, Subscription, InstallPlan, and all other required resources, including RBAC.

# 5.3.2.7. Creating a custom resource

After your Operator is installed, you can test it by creating a custom resource (CR) that is now provided on the cluster by the Operator.

### **Prerequisites**

• Example Memcached Operator, which provides the **Memcached** CR, installed on a cluster

### **Procedure**

- 1. Change to the namespace where your Operator is installed. For example, if you deployed the Operator using the **make deploy** command:
  - \$ oc project memcached-operator-system
- 2. Edit the sample **Memcached** CR manifest at **config/samples/cache\_v1\_memcached.yaml** to contain the following specification:

```
apiVersion: cache.example.com/v1 kind: Memcached metadata: name: memcached-sample ... spec: ... size: 3
```

3. Create the CR:

\$ oc apply -f config/samples/cache\_v1\_memcached.yaml

4. Ensure that the **Memcached** Operator creates the deployment for the sample CR with the correct size:

\$ oc get deployments

# **Example output**

```
NAME READY UP-TO-DATE AVAILABLE AGE memcached-operator-controller-manager 1/1 1 1 8m memcached-sample 3/3 3 1 1m
```

- 5. Check the pods and CR status to confirm the status is updated with the Memcached pod names.
  - a. Check the pods:

\$ oc get pods

# **Example output**

```
NAME READY STATUS RESTARTS AGE memcached-sample-6fd7c98d8-7dqdr 1/1 Running 0 1m memcached-sample-6fd7c98d8-g5k7v 1/1 Running 0 1m memcached-sample-6fd7c98d8-m7vn7 1/1 Running 0 1m
```

b. Check the CR status:

\$ oc get memcached/memcached-sample -o yaml

## **Example output**

```
apiVersion: cache.example.com/v1
kind: Memcached
metadata:
...
name: memcached-sample
...
spec:
size: 3
status:
nodes:
- memcached-sample-6fd7c98d8-7dqdr
- memcached-sample-6fd7c98d8-g5k7v
- memcached-sample-6fd7c98d8-m7vn7
```

- 6. Update the deployment size.
  - a. Update config/samples/cache\_v1\_memcached.yaml file to change the spec.size field in the Memcached CR from 3 to 5:

```
$ oc patch memcached memcached-sample \
  -p '{"spec":{"size": 5}}' \
  --type=merge
```

b. Confirm that the Operator changes the deployment size:

\$ oc get deployments

# **Example output**

NAME READY UP-TO-DATE AVAILABLE AGE memcached-operator-controller-manager 1/1 1 1 10m memcached-sample 5/5 5 5 3m

- 7. Delete the CR by running the following command:
  - \$ oc delete -f config/samples/cache\_v1\_memcached.yaml
- 8. Clean up the resources that have been created as part of this tutorial.
  - If you used the **make deploy** command to test the Operator, run the following command:
    - \$ make undeploy
  - If you used the **operator-sdk run bundle** command to test the Operator, run the following command:
    - \$ operator-sdk cleanup ct\_name>

### 5.3.2.8. Additional resources

- See Project layout for Go-based Operators to learn about the directory structures created by the Operator SDK.
- If a cluster-wide egress proxy is configured, cluster administrators can override the proxy settings or inject a custom CA certificate for specific Operators running on Operator Lifecycle Manager (OLM).

# 5.3.3. Project layout for Go-based Operators

The **operator-sdk** CLI can generate, or *scaffold*, a number of packages and files for each Operator project.

# 5.3.3.1. Go-based project layout

Go-based Operator projects, the default type, generated using the **operator-sdk init** command contain the following files and directories:

File or directory	Purpose
main.go	Main program of the Operator. This instantiates a new manager that registers all custom resource definitions (CRDs) in the <b>apis</b> / directory and starts all controllers in the <b>controllers</b> / directory.
apis/	Directory tree that defines the APIs of the CRDs. You must edit the apis/ <version>/<kind>_types.go files to define the API for each resource type and import these packages in your controllers to watch for these resource types.</kind></version>

File or directory	Purpose
controllers/	Controller implementations. Edit the <b>controller/<kind>_controller.go</kind></b> files to define the reconcile logic of the controller for handling a resource type of the specified kind.
config/	Kubernetes manifests used to deploy your controller on a cluster, including CRDs, RBAC, and certificates.
Makefile	Targets used to build and deploy your controller.
Dockerfile	Instructions used by a container engine to build your Operator.
manifests/	Kubernetes manifests for registering CRDs, setting up RBAC, and deploying the Operator as a deployment.

# 5.3.4. Updating Go-based Operator projects for newer Operator SDK versions

OpenShift Container Platform 4.13 supports Operator SDK 1.28.0. If you already have the 1.25.4 CLI installed on your workstation, you can update the CLI to 1.28.0 by installing the latest version.

However, to ensure your existing Operator projects maintain compatibility with Operator SDK 1.28.0, update steps are required for the associated breaking changes introduced since 1.25.4. You must perform the update steps manually in any of your Operator projects that were previously created or maintained with 1.25.4.

# 5.3.4.1. Updating Go-based Operator projects for Operator SDK 1.28.0

The following procedure updates an existing Go-based Operator project for compatibility with 1.28.0.

## **Prerequisites**

- Operator SDK 1.28.0 installed
- An Operator project created or maintained with Operator SDK 1.25.4

## **Procedure**

- Find the ose-kube-rbac-proxy pull spec in the following files, and update the image tag to v4.13:
  - config/default/manager\_auth\_proxy\_patch.yaml
  - bundle/manifests/memcached-operator.clusterserviceversion.yaml

containers:
- name: kube-rbac-proxy
image: registry.redhat.io/openshift4/ose-kube-rbac-proxy:v4.13 1

Update the tag version from **v4.12** to **v4.13**.

2. Modify your **go.mod** file to include the following dependencies and updated versions:

k8s.io/api v0.26.2 k8s.io/apiextensions-apiserver v0.26.2 k8s.io/apimachinery v0.26.2 k8s.io/cli-runtime v0.26.2 k8s.io/client-go v0.26.2 k8s.io/kubectl v0.26.2 sigs.k8s.io/controller-runtime v0.14.5 sigs.k8s.io/controller-tools v0.11.3 sigs.k8s.io/kubebuilder/v3 v3.9.1

3. Download the latest dependencies by running the following command:

\$ go mod tidy

- 4. Modify your Makefile with the following changes:
  - a. Change the ENVTEST\_K8S\_VERSION field from 1.25 to 1.26.
  - b. Change the build target from generate fmt vet to manifests generate fmt vet:
    - build: generate fmt vet ## Build manager binary.
    - + build: manifests generate fmt vet ## Build manager binary.

#### 5.3.4.2. Additional resources

- Migrating package manifest projects to bundle format
- Upgrading projects for Operator SDK 1.16.0
- Upgrading projects for Operator SDK v1.10.1
- Upgrading projects for Operator SDK v1.8.0

## 5.4. ANSIBLE-BASED OPERATORS

# 5.4.1. Getting started with Operator SDK for Ansible-based Operators

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

To demonstrate the basics of setting up and running an Ansible-based Operator using tools and libraries provided by the Operator SDK, Operator developers can build an example Ansible-based Operator for Memcached, a distributed key-value store, and deploy it to a cluster.

# 5.4.1.1. Prerequisites

- Operator SDK CLI installed
- OpenShift CLI (oc) v4.13+ installed
- Ansible v2.9.0

- Ansible Runner v2.0.2+
- Ansible Runner HTTP Event Emitter plugin v1.0.0+
- Python 3.8.6+
- OpenShift Python client v0.12.0+
- Logged into an OpenShift Container Platform 4.13 cluster with oc with an account that has cluster-admin permissions
- To allow the cluster to pull the image, the repository where you push your image must be set as public, or you must configure an image pull secret

#### Additional resources

- Installing the Operator SDK CLI
- Getting started with the OpenShift CLI

# 5.4.1.2. Creating and deploying Ansible-based Operators

You can build and deploy a simple Ansible-based Operator for Memcached by using the Operator SDK.

### Procedure

- 1. Create a project.
  - a. Create your project directory:
    - \$ mkdir memcached-operator
  - b. Change into the project directory:
    - \$ cd memcached-operator
  - c. Run the operator-sdk init command with the ansible plugin to initialize the project:
    - \$ operator-sdk init \ --plugins=ansible \ --domain=example.com
- 2. Create an API.

Create a simple Memcached API:

- \$ operator-sdk create api \ --group cache \

  - --version v1 \
  - --kind Memcached \
  - --generate-role
- Generates an Ansible role for the API.

3. Build and push the Operator image.

Use the default **Makefile** targets to build and push your Operator. Set **IMG** with a pull spec for your image that uses a registry you can push to:

\$ make docker-build docker-push IMG=<registry>/<user>/<image\_name>:<tag>

- 4. Run the Operator.
  - a. Install the CRD:
    - \$ make install
  - b. Deploy the project to the cluster. Set **IMG** to the image that you pushed:
    - \$ make deploy IMG=<registry>/<user>/<image\_name>:<tag>
- 5. Create a sample custom resource (CR).
  - a. Create a sample CR:
  - b. Watch for the CR to reconcile the Operator:
    - \$ oc logs deployment.apps/memcached-operator-controller-manager \
       -c manager \
      - -n memcached-operator-system

## **Example output**

...

10205 17:48:45.881666 7 leaderelection.go:253] successfully acquired lease memcached-operator-system/memcached-operator

{"level":"info","ts":1612547325.8819902,"logger":"controller-runtime.manager.controller.memcached-controller","msg":"Starting

EventSource","source":"kind source: cache.example.com/v1, Kind=Memcached"}

{"level":"info","ts":1612547325.98242,"logger":"controller-runtime.manager.controller.memcached-controller","msg":"Starting Controller"}

{"level":"info","ts":1612547325.9824686,"logger":"controller-runtime.manager.controller.memcached-controller","msg":"Starting workers","worker count":4}

{"level":"info","ts":1612547348.8311093,"logger":"runner","msg":"Ansible-runner exited successfully","job":"4037200794235010051","name":"memcached-sample","namespace":"memcached-operator-system"}

6. Delete a CR.

Delete a CR by running the following command:

 $\$ \ oc \ delete \ -f \ config/samples/cache\_v1\_memcached \ -n \ memcached-operator-system$ 

7. Clean up.

Run the following command to clean up the resources that have been created as part of this procedure:

\$ make undeploy

## 5.4.1.3. Next steps

• See Operator SDK tutorial for Ansible-based Operators for a more in-depth walkthrough on building an Ansible-based Operator.

# 5.4.2. Operator SDK tutorial for Ansible-based Operators

Operator developers can take advantage of Ansible support in the Operator SDK to build an example Ansible-based Operator for Memcached, a distributed key-value store, and manage its lifecycle. This tutorial walks through the following process:

- Create a Memcached deployment
- Ensure that the deployment size is the same as specified by the Memcached custom resource (CR) spec
- Update the Memcached CR status using the status writer with the names of the memcached pods

This process is accomplished by using two centerpieces of the Operator Framework:

## **Operator SDK**

The operator-sdk CLI tool and controller-runtime library API

## Operator Lifecycle Manager (OLM)

Installation, upgrade, and role-based access control (RBAC) of Operators on a cluster



#### NOTE

This tutorial goes into greater detail than Getting started with Operator SDK for Ansible-based Operators.

## 5.4.2.1. Prerequisites

- Operator SDK CLI installed
- OpenShift CLI (oc) v4.13+ installed
- Ansible v2.9.0
- Ansible Runner v2.0.2+
- Ansible Runner HTTP Event Emitter plugin v1.0.0+
- Python 3.8.6+
- OpenShift Python client v0.12.0+
- Logged into an OpenShift Container Platform 4.13 cluster with oc with an account that has cluster-admin permissions

• To allow the cluster to pull the image, the repository where you push your image must be set as public, or you must configure an image pull secret

### Additional resources

- Installing the Operator SDK CLI
- Getting started with the OpenShift CLI

# 5.4.2.2. Creating a project

Use the Operator SDK CLI to create a project called **memcached-operator**.

### **Procedure**

- 1. Create a directory for the project:
  - \$ mkdir -p \$HOME/projects/memcached-operator
- 2. Change to the directory:
  - \$ cd \$HOME/projects/memcached-operator
- 3. Run the **operator-sdk init** command with the **ansible** plugin to initialize the project:

```
$ operator-sdk init \
   --plugins=ansible \
   --domain=example.com
```

--domain=example.com

## 5.4.2.2.1. PROJECT file

Among the files generated by the **operator-sdk init** command is a Kubebuilder **PROJECT** file. Subsequent **operator-sdk** commands, as well as **help** output, that are run from the project root read this file and are aware that the project type is Ansible. For example:

```
domain: example.com
layout:
- ansible.sdk.operatorframework.io/v1
plugins:
    manifests.sdk.operatorframework.io/v2: {}
    scorecard.sdk.operatorframework.io/v2: {}
    sdk.x-openshift.io/v1: {}
    projectName: memcached-operator
    version: "3"
```

## 5.4.2.3. Creating an API

Use the Operator SDK CLI to create a Memcached API.

#### **Procedure**

• Run the following command to create an API with group **cache**, version, **v1**, and kind **Memcached**:

```
$ operator-sdk create api \
   --group cache \
   --version v1 \
   --kind Memcached \
   --generate-role 1
```

Generates an Ansible role for the API.

After creating the API, your Operator project updates with the following structure:

#### Memcached CRD

Includes a sample **Memcached** resource

## Manager

Program that reconciles the state of the cluster to the desired state by using:

- A reconciler, either an Ansible role or playbook
- A watches.yaml file, which connects the Memcached resource to the memcached Ansible role

# 5.4.2.4. Modifying the manager

Update your Operator project to provide the reconcile logic, in the form of an Ansible role, which runs every time a **Memcached** resource is created, updated, or deleted.

## Procedure

1. Update the **roles/memcached/tasks/main.yml** file with the following structure:

```
- name: start memcached
 k8s:
  definition:
   kind: Deployment
   apiVersion: apps/v1
   metadata:
    name: '{{ ansible_operator_meta.name }}-memcached'
    namespace: '{{ ansible_operator_meta.namespace }}'
   spec:
    replicas: "{{size}}"
    selector:
     matchLabels:
      app: memcached
    template:
     metadata:
      labels:
        app: memcached
     spec:
      containers:
      - name: memcached
        command:
        - memcached
```

```
- -m=64
- -0
- modern
- -v
image: "docker.io/memcached:1.4.36-alpine"
ports:
- containerPort: 11211
```

This **memcached** role ensures a **memcached** deployment exist and sets the deployment size.

2. Set default values for variables used in your Ansible role by editing the roles/memcached/defaults/main.yml file:

```
# defaults file for Memcached size: 1
```

3. Update the **Memcached** sample resource in the **config/samples/cache\_v1\_memcached.yaml** file with the following structure:

```
apiVersion: cache.example.com/v1
kind: Memcached
metadata:
labels:
app.kubernetes.io/name: memcached
app.kubernetes.io/instance: memcached-sample
app.kubernetes.io/part-of: memcached-operator
app.kubernetes.io/managed-by: kustomize
app.kubernetes.io/created-by: memcached-operator
name: memcached-sample
spec:
size: 3
```

The key-value pairs in the custom resource (CR) spec are passed to Ansible as extra variables.



#### **NOTE**

The names of all variables in the **spec** field are converted to snake case, meaning lowercase with an underscore, by the Operator before running Ansible. For example, **serviceAccount** in the spec becomes **service\_account** in Ansible.

You can disable this case conversion by setting the **snakeCaseParameters** option to **false** in your **watches.yaml** file. It is recommended that you perform some type validation in Ansible on the variables to ensure that your application is receiving expected input.

# 5.4.2.5. Enabling proxy support

Operator authors can develop Operators that support network proxies. Cluster administrators configure proxy support for the environment variables that are handled by Operator Lifecycle Manager (OLM). To support proxied clusters, your Operator must inspect the environment for the following standard proxy variables and pass the values to Operands:

- HTTP\_PROXY
- HTTPS\_PROXY

# NO\_PROXY



#### NOTE

This tutorial uses **HTTP\_PROXY** as an example environment variable.

# **Prerequisites**

• A cluster with cluster-wide egress proxy enabled.

### Procedure

1. Add the environment variables to the deployment by updating the roles/memcached/tasks/main.yml file with the following:

```
...
env:
    - name: HTTP_PROXY
    value: '{{ lookup("env", "HTTP_PROXY") | default("", True) }}'
    - name: http_proxy
    value: '{{ lookup("env", "HTTP_PROXY") | default("", True) }}'
...
```

2. Set the environment variable on the Operator deployment by adding the following to the **config/manager/manager.yaml** file:

```
containers:
- args:
- --leader-elect
- --leader-election-id=ansible-proxy-demo image: controller:latest name: manager env:
- name: "HTTP_PROXY" value: "http_proxy_test"
```

## 5.4.2.6. Running the Operator

There are three ways you can use the Operator SDK CLI to build and run your Operator:

- Run locally outside the cluster as a Go program.
- Run as a deployment on the cluster.
- Bundle your Operator and use Operator Lifecycle Manager (OLM) to deploy on the cluster.

### 5.4.2.6.1. Running locally outside the cluster

You can run your Operator project as a Go program outside of the cluster. This is useful for development purposes to speed up deployment and testing.

### Procedure

• Run the following command to install the custom resource definitions (CRDs) in the cluster configured in your ~/.kube/config file and run the Operator locally:

\$ make install run

# **Example output**

```
"level":"info","ts":1612589622.7888272,"logger":"ansible-controller","msg":"Watching resource","Options.Group":"cache.example.com","Options.Version":"v1","Options.Kind":"Memc ached"}

{"level":"info","ts":1612589622.7897573,"logger":"proxy","msg":"Starting to serve","Address":"127.0.0.1:8888"}

{"level":"info","ts":1612589622.789971,"logger":"controller-runtime.manager","msg":"starting metrics server","path":"/metrics"}

{"level":"info","ts":1612589622.789997,"logger":"controller-runtime.manager.controller.memcached-controller","msg":"Starting

EventSource","source":"kind source: cache.example.com/v1, Kind=Memcached"}

{"level":"info","ts":1612589622.8904517,"logger":"controller-runtime.manager.controller.memcached-controller","msg":"Starting Controller"}

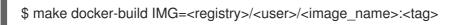
{"level":"info","ts":1612589622.8905244,"logger":"controller-runtime.manager.controller.memcached-controller","msg":"Starting workers","worker count":8}
```

#### 5.4.2.6.2. Running as a deployment on the cluster

You can run your Operator project as a deployment on your cluster.

## **Procedure**

- 1. Run the following **make** commands to build and push the Operator image. Modify the **IMG** argument in the following steps to reference a repository that you have access to. You can obtain an account for storing containers at repository sites such as Quay.io.
  - a. Build the image:





# NOTE

The Dockerfile generated by the SDK for the Operator explicitly references **GOARCH=amd64** for **go build**. This can be amended to **GOARCH=\$TARGETARCH** for non-AMD64 architectures. Docker will automatically set the environment variable to the value specified by – **platform**. With Buildah, the **–build-arg** will need to be used for the purpose. For more information, see Multiple Architectures.

b. Push the image to a repository:

\$ make docker-push IMG=<registry>/<user>/<image name>:<tag>



#### NOTE

The name and tag of the image, for example **IMG**= <registry>/<user>/<image\_name>:<tag>, in both the commands can also be set in your Makefile. Modify the **IMG** ?= controller:latest value to set your default image name.

2. Run the following command to deploy the Operator:

\$ make deploy IMG=<registry>/<user>/<image\_name>:<tag>

By default, this command creates a namespace with the name of your Operator project in the form **cproject\_name>-system and is used for the deployment. This command also installs the RBAC manifests from <b>config/rbac**.

3. Run the following command to verify that the Operator is running:

# **Example output**

NAME	READY	UP-TO	D-DATE	AVA	ILABLE	AGE
<pre><pre><pre><pre>oproject_name</pre><pre>-controller-manager</pre></pre></pre></pre>		1/1	1	1	8m	

### 5.4.2.6.3. Bundling an Operator and deploying with Operator Lifecycle Manager

## 5.4.2.6.3.1. Bundling an Operator

The Operator bundle format is the default packaging method for Operator SDK and Operator Lifecycle Manager (OLM). You can get your Operator ready for use on OLM by using the Operator SDK to build and push your Operator project as a bundle image.

## **Prerequisites**

- Operator SDK CLI installed on a development workstation
- OpenShift CLI (oc) v4.13+ installed
- Operator project initialized by using the Operator SDK

#### **Procedure**

- Run the following make commands in your Operator project directory to build and push your Operator image. Modify the IMG argument in the following steps to reference a repository that you have access to. You can obtain an account for storing containers at repository sites such as Quay.io.
  - a. Build the image:

\$ make docker-build IMG=<registry>/<user>/<operator\_image\_name>:<tag>



### **NOTE**

The Dockerfile generated by the SDK for the Operator explicitly references **GOARCH=amd64** for **go build**. This can be amended to **GOARCH=\$TARGETARCH** for non-AMD64 architectures. Docker will automatically set the environment variable to the value specified by – **platform**. With Buildah, the **–build-arg** will need to be used for the purpose. For more information, see Multiple Architectures.

- b. Push the image to a repository:
  - \$ make docker-push IMG=<registry>/<user>/<operator\_image\_name>:<tag>
- Create your Operator bundle manifest by running the make bundle command, which invokes several commands, including the Operator SDK generate bundle and bundle validate subcommands:
  - \$ make bundle IMG=<registry>/<user>/<operator\_image\_name>:<tag>

Bundle manifests for an Operator describe how to display, create, and manage an application. The **make bundle** command creates the following files and directories in your Operator project:

- A bundle manifests directory named bundle/manifests that contains a ClusterServiceVersion object
- A bundle metadata directory named bundle/metadata
- All custom resource definitions (CRDs) in a **config/crd** directory
- A Dockerfile bundle.Dockerfile

These files are then automatically validated by using **operator-sdk bundle validate** to ensure the on-disk bundle representation is correct.

- 3. Build and push your bundle image by running the following commands. OLM consumes Operator bundles using an index image, which reference one or more bundle images.
  - a. Build the bundle image. Set **BUNDLE\_IMG** with the details for the registry, user namespace, and image tag where you intend to push the image:
    - \$ make bundle-build BUNDLE\_IMG=<registry>/<user>/<bundle\_image\_name>:<tag>
  - b. Push the bundle image:
    - \$ docker push <registry>/<user>/<bundle\_image\_name>:<tag>

# 5.4.2.6.3.2. Deploying an Operator with Operator Lifecycle Manager

Operator Lifecycle Manager (OLM) helps you to install, update, and manage the lifecycle of Operators and their associated services on a Kubernetes cluster. OLM is installed by default on OpenShift Container Platform and runs as a Kubernetes extension so that you can use the web console and the OpenShift CLI (oc) for all Operator lifecycle management functions without any additional tools.

The Operator bundle format is the default packaging method for Operator SDK and OLM. You can use the Operator SDK to quickly run a bundle image on OLM to ensure that it runs properly.

### **Prerequisites**

- Operator SDK CLI installed on a development workstation
- Operator bundle image built and pushed to a registry
- OLM installed on a Kubernetes-based cluster (v1.16.0 or later if you use apiextensions.k8s.io/v1 CRDs, for example OpenShift Container Platform 4.13)
- Logged in to the cluster with **oc** using an account with **cluster-admin** permissions

### Procedure

• Enter the following command to run the Operator on the cluster:

\$ operator-sdk run bundle \1
-n <namespace> \2
<registry>/<user>/<bundle\_image\_name>:<tag> 3

- The **run bundle** command creates a valid file-based catalog and installs the Operator bundle on your cluster using OLM.
- Optional: By default, the command installs the Operator in the currently active project in your ~/.kube/config file. You can add the -n flag to set a different namespace scope for the installation.
- If you do not specify an image, the command uses **quay.io/operator-framework/opm:latest** as the default index image. If you specify an image, the command uses the bundle image itself as the index image.



#### **IMPORTANT**

As of OpenShift Container Platform 4.11, the **run bundle** command supports the file-based catalog format for Operator catalogs by default. The deprecated SQLite database format for Operator catalogs continues to be supported; however, it will be removed in a future release. It is recommended that Operator authors migrate their workflows to the file-based catalog format.

This command performs the following actions:

- Create an index image referencing your bundle image. The index image is opaque and ephemeral, but accurately reflects how a bundle would be added to a catalog in production.
- Create a catalog source that points to your new index image, which enables OperatorHub to discover your Operator.
- Deploy your Operator to your cluster by creating an OperatorGroup, Subscription, InstallPlan, and all other required resources, including RBAC.

# 5.4.2.7. Creating a custom resource

After your Operator is installed, you can test it by creating a custom resource (CR) that is now provided on the cluster by the Operator.

# **Prerequisites**

• Example Memcached Operator, which provides the **Memcached** CR, installed on a cluster

#### **Procedure**

1. Change to the namespace where your Operator is installed. For example, if you deployed the Operator using the **make deploy** command:

\$ oc project memcached-operator-system

2. Edit the sample **Memcached** CR manifest at **config/samples/cache\_v1\_memcached.yaml** to contain the following specification:

```
apiVersion: cache.example.com/v1 kind: Memcached metadata: name: memcached-sample ... spec: ... size: 3
```

3. Create the CR:

\$ oc apply -f config/samples/cache\_v1\_memcached.yaml

4. Ensure that the **Memcached** Operator creates the deployment for the sample CR with the correct size:

\$ oc get deployments

# **Example output**

```
NAME READY UP-TO-DATE AVAILABLE AGE memcached-operator-controller-manager 1/1 1 1 8m memcached-sample 3/3 3 1m
```

- 5. Check the pods and CR status to confirm the status is updated with the Memcached pod names.
  - a. Check the pods:

\$ oc get pods

# **Example output**

NAME	READY	STATUS	RESTARTS	AGE
memcached-sample-6fd	7c98d8-7dqdr	1/1	Running 0	1m
memcached-sample-6fd	7c98d8-g5k7v	1/1	Running 0	1m
memcached-sample-6fd	7c98d8-m7vn	7 1/1	Running 0	1m

b. Check the CR status:

\$ oc get memcached/memcached-sample -o yaml

# **Example output**

```
apiVersion: cache.example.com/v1
kind: Memcached
metadata:
...
name: memcached-sample
...
spec:
size: 3
status:
nodes:
- memcached-sample-6fd7c98d8-7dqdr
- memcached-sample-6fd7c98d8-g5k7v
- memcached-sample-6fd7c98d8-m7vn7
```

- 6. Update the deployment size.
  - a. Update config/samples/cache\_v1\_memcached.yaml file to change the spec.size field in the Memcached CR from 3 to 5:

```
$ oc patch memcached memcached-sample \
  -p '{"spec":{"size": 5}}' \
  --type=merge
```

b. Confirm that the Operator changes the deployment size:

\$ oc get deployments

# **Example output**

```
NAME READY UP-TO-DATE AVAILABLE AGE memcached-operator-controller-manager 1/1 1 1 10m memcached-sample 5/5 5 5 3m
```

- 7. Delete the CR by running the following command:
  - \$ oc delete -f config/samples/cache\_v1\_memcached.yaml
- 8. Clean up the resources that have been created as part of this tutorial.
  - If you used the **make deploy** command to test the Operator, run the following command:
    - \$ make undeploy
  - If you used the operator-sdk run bundle command to test the Operator, run the following command:
    - \$ operator-sdk cleanup <project\_name>

### 5.4.2.8. Additional resources

- See Project layout for Ansible-based Operators to learn about the directory structures created by the Operator SDK.
- If a cluster-wide egress proxy is configured, cluster administrators can override the proxy settings or inject a custom CA certificate for specific Operators running on Operator Lifecycle Manager (OLM).

# 5.4.3. Project layout for Ansible-based Operators

The **operator-sdk** CLI can generate, or *scaffold*, a number of packages and files for each Operator project.

# 5.4.3.1. Ansible-based project layout

Ansible-based Operator projects generated using the **operator-sdk init --plugins ansible** command contain the following directories and files:

File or directory	Purpose
Dockerfile	Dockerfile for building the container image for the Operator.
Makefile	Targets for building, publishing, deploying the container image that wraps the Operator binary, and targets for installing and uninstalling the custom resource definition (CRD).
PROJECT	YAML file containing metadata information for the Operator.
config/crd	Base CRD files and the <b>kustomization.yaml</b> file settings.
config/default	Collects all Operator manifests for deployment. Use by the <b>make deploy</b> command.
config/manager	Controller manager deployment.
config/prometh eus	ServiceMonitor resource for monitoring the Operator.
config/rbac	Role and role binding for leader election and authentication proxy.
config/samples	Sample resources created for the CRDs.
config/testing	Sample configurations for testing.
playbooks/	A subdirectory for the playbooks to run.
roles/	Subdirectory for the roles tree to run.
watches.yaml	Group/version/kind (GVK) of the resources to watch, and the Ansible invocation method. New entries are added by using the <b>create api</b> command.

File or directory	Purpose	
requirements.y ml	YAML file containing the Ansible collections and role dependencies to install during a build.	
molecule/	Molecule scenarios for end-to-end testing of your role and Operator.	

# 5.4.4. Updating projects for newer Operator SDK versions

OpenShift Container Platform 4.13 supports Operator SDK 1.28.0. If you already have the 1.25.4 CLI installed on your workstation, you can update the CLI to 1.28.0 by installing the latest version.

However, to ensure your existing Operator projects maintain compatibility with Operator SDK 1.28.0, update steps are required for the associated breaking changes introduced since 1.25.4. You must perform the update steps manually in any of your Operator projects that were previously created or maintained with 1.25.4.

# 5.4.4.1. Updating Ansible-based Operator projects for Operator SDK 1.28.0

The following procedure updates an existing Ansible-based Operator project for compatibility with 1.28.0.

# **Prerequisites**

- Operator SDK 1.28.0 installed
- An Operator project created or maintained with Operator SDK 1.25.4

#### Procedure

1. Update the image tag in your Operator's Dockerfile as shown in the following example:

### **Example Dockerfile**

- FROM registry.redhat.io/openshift4/ose-ansible-operator:v4.13
- Update the version tag to v4.13.
- 2. Find the **ose-kube-rbac-proxy** pull spec in the following files, and update the image tag to v4.13:
  - config/default/manager\_auth\_proxy\_patch.yaml
  - bundle/manifests/memcached-operator.clusterserviceversion.yaml

containers:

name: kube-rbac-proxy

image: registry.redhat.io/openshift4/ose-kube-rbac-proxy:v4.13

1

Update the tag version from **v4.12** to **v4.13**.

3. Update your Makefile's **run** target to the following:

.PHONY: run

ANSIBLE ROLES PATH?="\$(shell pwd)/roles"

run: ansible-operator ## Run against the configured Kubernetes cluster in ~/.kube/config

\$(ANSIBLE\_OPERATOR) run

4. To upgrade the **kubernetes.core** collection to v2.4.0, replace the following in your project's **requirements.yaml** file:

- name: kubernetes.core version: "2.3.1"

with:

- name: kubernetes.core

version: "2.4.0"

### 5.4.4.2. Additional resources

• Upgrading projects for Operator SDK v1.25.4

- Upgrading projects for Operator SDK v1.22.0
- Upgrading projects for Operator SDK v1.16.0
- Upgrading projects for Operator SDK v1.10.1
- Upgrading projects for Operator SDK v1.8.0
- Migrating package manifest projects to bundle format

# 5.4.5. Ansible support in Operator SDK

# 5.4.5.1. Custom resource files

Operators use the Kubernetes extension mechanism, custom resource definitions (CRDs), so your custom resource (CR) looks and acts just like the built-in, native Kubernetes objects.

The CR file format is a Kubernetes resource file. The object has mandatory and optional fields:

Table 5.1. Custom resource fields

Field	Description
apiVersion	Version of the CR to be created.
kind	Kind of the CR to be created.

Field	Description	
metadata	Kubernetes-specific metadata to be created.	
spec (optional)	Key-value list of variables which are passed to Ansible. This field is empty by default.	
status	Summarizes the current state of the object. For Ansible-based Operators, the <b>status</b> subresource is enabled for CRDs and managed by the <b>operator_sdk.util.k8s_status</b> Ansible module by default, which includes <b>condition</b> information to the CR <b>status</b> .	
annotations	Kubernetes-specific annotations to be appended to the CR.	

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

Table 5.2. Ansible-based Operator annotations

Annotation	Description
ansible.operator- sdk/reconcile-period	Specifies the reconciliation interval for the CR. This value is parsed using the standard Golang package <b>time</b> . Specifically, <b>ParseDuration</b> is used which applies the default suffix of <b>s</b> , giving the value in seconds.

# **Example Ansible-based Operator annotation**

apiVersion: "test1.example.com/v1alpha1"

kind: "Test1" metadata:

name: "example" annotations:

ansible.operator-sdk/reconcile-period: "30s"

# 5.4.5.2. watches.yaml file

A group/version/kind (GVK) is a unique identifier for a Kubernetes API. The **watches.yaml** file contains a list of mappings from custom resources (CRs), identified by its GVK, to an Ansible role or playbook. The Operator expects this mapping file in a predefined location at /opt/ansible/watches.yaml.

Table 5.3. watches.yaml file mappings

Field	Description
group	Group of CR to watch.
version	Version of CR to watch.
kind	Kind of CR to watch

Field	Description
role (default)	Path to the Ansible role added to the container. For example, if your <b>roles</b> directory is at / <b>opt/ansible/roles/</b> and your role is named <b>busybox</b> , this value would be / <b>opt/ansible/roles/busybox</b> . This field is mutually exclusive with the <b>playbook</b> field.
playbook	Path to the Ansible playbook added to the container. This playbook is expected to be a way to call roles. This field is mutually exclusive with the <b>role</b> field.
reconcilePeriod (optional)	The reconciliation interval, how often the role or playbook is run, for a given CR.
manageStatus (optional)	When set to <b>true</b> (default), the Operator manages the status of the CR generically. When set to <b>false</b> , the status of the CR is managed elsewhere, by the specified role or playbook or in a separate controller.

# Example watches.yaml file

- version: v1alpha1 1 group: test1.example.com

kind: Test1

role: /opt/ansible/roles/Test1

- version: v1alpha1 2 group: test2.example.com

kind: Test2

playbook: /opt/ansible/playbook.yml

- version: v1alpha1 3 group: test3.example.com

kind: Test3

playbook: /opt/ansible/test3.yml

reconcilePeriod: 0 manageStatus: false

- Simple example mapping **Test1** to the **test1** role.
- Simple example mapping **Test2** to a playbook.
- More complex example for the **Test3** kind. Disables re-queuing and managing the CR status in the playbook.

### 5.4.5.2.1. Advanced options

Advanced features can be enabled by adding them to your **watches.yaml** file per GVK. They can go below the **group**, **version**, **kind** and **playbook** or **role** fields.

Some features can be overridden per resource using an annotation on that CR. The options that can be overridden have the annotation specified below.

Table 5.4. Advanced watches.yaml file options

Feature	YAML key	Description	Annotation for override	Defa ult valu e
Reconcile period	reconcilePeri od	Time between reconcile runs for a particular CR.	ansible.oper ator- sdk/reconcil e-period	1m
Manage status	manageStatu s	Allows the Operator to manage the <b>conditions</b> section of each CR <b>status</b> section.		true
Watch dependent resources	watchDepen dentResourc es	Allows the Operator to dynamically watch resources that are created by Ansible.		true
Watch cluster-scoped resources	watchCluster ScopedReso urces	Allows the Operator to watch cluster-scoped resources that are created by Ansible.		fals e
Max runner artifacts	maxRunnerA rtifacts	Manages the number of artifact directories that Ansible Runner keeps in the Operator container for each individual resource.	ansible.oper ator- sdk/max- runner- artifacts	20

# Example watches.yml file with advanced options

version: v1alpha1 group: app.example.com

kind: AppService

playbook: /opt/ansible/playbook.yml

maxRunnerArtifacts: 30 reconcilePeriod: 5s manageStatus: False

watchDependentResources: False

# 5.4.5.3. Extra variables sent to Ansible

Extra variables can be sent to Ansible, which are then managed by the Operator. The **spec** section of the custom resource (CR) passes along the key-value pairs as extra variables. This is equivalent to extra variables passed in to the **ansible-playbook** command.

The Operator also passes along additional variables under the **meta** field for the name of the CR and the namespace of the CR.

For the following CR example:

```
apiVersion: "app.example.com/v1alpha1"
kind: "Database"
metadata:
name: "example"
spec:
message: "Hello world 2"
newParameter: "newParam"
```

The structure passed to Ansible as extra variables is:

```
{ "meta": {
          "name": "<cr_name>",
          "namespace": "<cr_namespace>",
     },
     "message": "Hello world 2",
     "new_parameter": "newParam",
     "_app_example_com_database": {
        <full_crd>
     },
}
```

The **message** and **newParameter** fields are set in the top level as extra variables, and **meta** provides the relevant metadata for the CR as defined in the Operator. The **meta** fields can be accessed using dot notation in Ansible, for example:

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

# 5.4.5.4. Ansible Runner directory

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

#### Additional resources

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

# 5.4.6. Kubernetes Collection for Ansible

To manage the lifecycle of your application on Kubernetes using Ansible, you can use the Kubernetes Collection for Ansible. This collection of Ansible modules allows a developer to either leverage their existing Kubernetes resource files written in YAML or express the lifecycle management in native Ansible.

One of the biggest benefits of using Ansible in conjunction with existing Kubernetes resource files is the ability to use Jinja templating so that you can customize resources with the simplicity of a few variables in Ansible.

This section goes into detail on usage of the Kubernetes Collection. To get started, install the collection on your local workstation and test it using a playbook before moving on to using it within an Operator.

### 5.4.6.1. Installing the Kubernetes Collection for Ansible

You can install the Kubernetes Collection for Ansible on your local workstation.

# **Procedure**

- 1. Install Ansible 2.9+:
  - \$ sudo dnf install ansible
- 2. Install the OpenShift python client package:
  - \$ pip3 install openshift
- 3. Install the Kubernetes Collection using one of the following methods:
  - You can install the collection directly from Ansible Galaxy:
    - \$ ansible-galaxy collection install community.kubernetes
  - If you have already initialized your Operator, you might have a requirements.yml file at the
    top level of your project. This file specifies Ansible dependencies that must be installed for
    your Operator to function. By default, this file installs the community.kubernetes
    collection as well as the operator\_sdk.util collection, which provides modules and plugins
    for Operator-specific fuctions.

To install the dependent modules from the **requirements.yml** file:

\$ ansible-galaxy collection install -r requirements.yml

# 5.4.6.2. Testing the Kubernetes Collection locally

Operator developers can run the Ansible code from their local machine as opposed to running and rebuilding the Operator each time.

# **Prerequisites**

- Initialize an Ansible-based Operator project and create an API that has a generated Ansible role by using the Operator SDK
- Install the Kubernetes Collection for Ansible

#### **Procedure**

In your Ansible-based Operator project directory, modify the roles/<kind>/tasks/main.yml file with the Ansible logic that you want. The roles/<kind>/ directory is created when you use the -- generate-role flag while creating an API. The <kind> replaceable matches the kind that you specified for the API.

The following example creates and deletes a config map based on the value of a variable named **state**:

---

- name: set ConfigMap example-config to {{ state }}community.kubernetes.k8s:

api\_version: v1 kind: ConfigMap

name: example-config
namespace: <operator\_namespace> 1
state: "{{ state }}"
ignore\_errors: true 2

- Specify the namespace where you want the config map created.
- Setting **ignore\_errors: true** ensures that deleting a nonexistent config map does not fail.
- 2. Modify the roles/<kind>/defaults/main.yml file to set state to present by default:

--state: present

- 3. Create an Ansible playbook by creating a **playbook.yml** file in the top-level of your project directory, and include your **<kind>** role:
  - --- hosts: localhost roles:
     <kind>
- 4. Run the playbook:

\$ ansible-playbook playbook.yml

# Example output

5. Verify that the config map was created:

\$ oc get configmaps

# **Example output**

NAME DATA AGE example-config 0 2m1s

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

\$ ansible-playbook playbook.yml --extra-vars state=absent

# **Example output**

TASK [Gathering Facts]  ***********************************	[WARNING]: provided hosts list is empty, only localhost is available. Note that the implicit localhost does not match 'all'
ok: [localhost]  TASK [memcached : set ConfigMap example-config to absent]  ***********************************	PLAY [localhost] ************************************
changed: [localhost]  PLAY RECAP ************************************	**************
PLAY RECAP ************************************	
localhost : ok=2 changed=1 unreachable=0 failed=0 skipped	changed: [localhost]
	PLAY RECAP ************************************

7. Verify that the config map was deleted:

\$ oc get configmaps

# 5.4.6.3. Next steps

• See Using Ansible inside an Operator for details on triggering your custom Ansible logic inside of an Operator when a custom resource (CR) changes.

# 5.4.7. Using Ansible inside an Operator

After you are familiar with using the Kubernetes Collection for Ansible locally, you can trigger the same Ansible logic inside of an Operator when a custom resource (CR) changes. This example maps an Ansible role to a specific Kubernetes resource that the Operator watches. This mapping is done in the **watches.yaml** file.

### 5.4.7.1. Custom resource files

Operators use the Kubernetes extension mechanism, custom resource definitions (CRDs), so your custom resource (CR) looks and acts just like the built-in, native Kubernetes objects.

The CR file format is a Kubernetes resource file. The object has mandatory and optional fields:

### Table 5.5. Custom resource fields

Field	Description	
apiVersion	Version of the CR to be created.	
kind	Kind of the CR to be created.	
metadata	Kubernetes-specific metadata to be created.	
spec (optional)	Key-value list of variables which are passed to Ansible. This field is empty by default.	
status	Summarizes the current state of the object. For Ansible-based Operators, the <b>status</b> subresource is enabled for CRDs and managed by the <b>operator_sdk.util.k8s_status</b> Ansible module by default, which includes <b>condition</b> information to the CR <b>status</b> .	
annotations	Kubernetes-specific annotations to be appended to the CR.	

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

Table 5.6. Ansible-based Operator annotations

Annotation	Description
ansible.operator- sdk/reconcile-period	Specifies the reconciliation interval for the CR. This value is parsed using the standard Golang package <b>time</b> . Specifically, <b>ParseDuration</b> is used which applies the default suffix of <b>s</b> , giving the value in seconds.

# **Example Ansible-based Operator annotation**

apiVersion: "test1.example.com/v1alpha1"

kind: "Test1" metadata:

name: "example" annotations:

ansible.operator-sdk/reconcile-period: "30s"

# 5.4.7.2. Testing an Ansible-based Operator locally

You can test the logic inside of an Ansible-based Operator running locally by using the **make run** command from the top-level directory of your Operator project. The **make run** Makefile target runs the **ansible-operator** binary locally, which reads from the **watches.yaml** file and uses your ~/.kube/config file to communicate with a Kubernetes cluster just as the **k8s** modules do.



### NOTE

You can customize the roles path by setting the environment variable **ANSIBLE\_ROLES\_PATH** or by using the **ansible-roles-path** flag. If the role is not found in the **ANSIBLE\_ROLES\_PATH** value, the Operator looks for it in **{{current directory}}/roles**.

### **Prerequisites**

- Ansible Runner v2.0.2+
- Ansible Runner HTTP Event Emitter plugin v1.0.0+
- Performed the previous steps for testing the Kubernetes Collection locally

#### **Procedure**

1. Install your custom resource definition (CRD) and proper role-based access control (RBAC) definitions for your custom resource (CR):

\$ make install

### Example output

/usr/bin/kustomize build config/crd | kubectl apply -f - customresourcedefinition.apiextensions.k8s.io/memcacheds.cache.example.com created

2. Run the **make run** command:

\$ make run

### Example output

```
/home/user/memcached-operator/bin/ansible-operator run
{"level":"info","ts":1612739145.2871568,"logger":"cmd","msg":"Version","Go
Version": "go1.15.5", "GOOS": "linux", "GOARCH": "amd64", "ansible-
operator":"v1.10.1","commit":"1abf57985b43bf6a59dcd18147b3c574fa57d3f6"}
{"level":"info", "ts":1612739148.347306, "logger": "controller-runtime.metrics", "msg": "metrics
server is starting to listen", "addr": ":8080"}
{"level":"info", "ts":1612739148.3488882, "logger": "watches", "msg": "Environment variable not
set: using default
value","envVar":"ANSIBLE VERBOSITY MEMCACHED CACHE EXAMPLE COM","default":
{"level":"info", "ts":1612739148.3490262, "logger": "cmd", "msg": "Environment variable not set;
using default
value","Namespace":"","envVar":"ANSIBLE_DEBUG_LOGS","ANSIBLE_DEBUG_LOGS":fals
{"level":"info", "ts":1612739148.3490646, "logger": "ansible-controller", "msg": "Watching
resource", "Options. Group": "cache.example.com", "Options. Version": "v1", "Options. Kind": "Memc
ached"}
{"level":"info", "ts":1612739148.350217, "logger": "proxy", "msg": "Starting to
serve","Address":"127.0.0.1:8888"}
{"level":"info", "ts":1612739148.3506632, "logger": "controller-runtime.manager", "msg": "starting
metrics server", "path": "/metrics"}
{"level":"info", "ts":1612739148.350784, "logger": "controller-
runtime.manager.controller.memcached-controller","msg":"Starting
EventSource", "source": "kind source: cache.example.com/v1, Kind=Memcached"}
{"level":"info", "ts":1612739148.5511978, "logger": "controller-
runtime.manager.controller.memcached-controller", "msg": "Starting Controller"}
```

{"level":"info","ts":1612739148.5512562,"logger":"controller-runtime.manager.controller.memcached-controller","msg":"Starting workers","worker count":8}

With the Operator now watching your CR for events, the creation of a CR will trigger your Ansible role to run.



#### **NOTE**

Consider an example config/samples/<gvk>.yaml CR manifest:

apiVersion: <group>.example.com/v1alpha1

kind: <kind> metadata:

name: "<kind>-sample"

Because the **spec** field is not set, Ansible is invoked with no extra variables. Passing extra variables from a CR to Ansible is covered in another section. It is important to set reasonable defaults for the Operator.

- 3. Create an instance of your CR with the default variable **state** set to **present**:
  - \$ oc apply -f config/samples/<gvk>.yaml
- 4. Check that the **example-config** config map was created:
  - \$ oc get configmaps

# Example output

NAME STATUS AGE example-config Active 3s

5. Modify your config/samples/<gvk>.yaml file to set the state field to absent. For example:

apiVersion: cache.example.com/v1

kind: Memcached

metadata:

name: memcached-sample

spec:

state: absent

6. Apply the changes:

\$ oc apply -f config/samples/<gvk>.yaml

7. Confirm that the config map is deleted:

\$ oc get configmap

# 5.4.7.3. Testing an Ansible-based Operator on the cluster

After you have tested your custom Ansible logic locally inside of an Operator, you can test the Operator inside of a pod on an OpenShift Container Platform cluster, which is preferred for production use.

You can run your Operator project as a deployment on your cluster.

#### **Procedure**

- 1. Run the following **make** commands to build and push the Operator image. Modify the **IMG** argument in the following steps to reference a repository that you have access to. You can obtain an account for storing containers at repository sites such as Quay.io.
  - a. Build the image:





#### **NOTE**

The Dockerfile generated by the SDK for the Operator explicitly references **GOARCH=amd64** for **go build**. This can be amended to **GOARCH=\$TARGETARCH** for non-AMD64 architectures. Docker will automatically set the environment variable to the value specified by – **platform**. With Buildah, the –**build-arg** will need to be used for the purpose. For more information, see Multiple Architectures.

b. Push the image to a repository:





# **NOTE**

The name and tag of the image, for example **IMG=** <registry>/<user>/<image\_name>:<tag>, in both the commands can also be set in your Makefile. Modify the **IMG ?= controller:latest** value to set your default image name.

2. Run the following command to deploy the Operator:

\$ make deploy IMG=<registry>/<user>/<image\_name>:<tag>

By default, this command creates a namespace with the name of your Operator project in the form **cproject\_name>-system and is used for the deployment. This command also installs the RBAC manifests from <b>config/rbac**.

3. Run the following command to verify that the Operator is running:

# **Example output**

# 5.4.7.4. Ansible logs

Ansible-based Operators provide logs about the Ansible run, which can be useful for debugging your Ansible tasks. The logs can also contain detailed information about the internals of the Operator and its interactions with Kubernetes.

### 5.4.7.4.1. Viewing Ansible logs

### **Prerequisites**

• Ansible-based Operator running as a deployment on a cluster

#### **Procedure**

• To view logs from an Ansible-based Operator, run the following command:

\$ oc logs deployment/c\_name>-controller-manager \
-c manager \1
-n <namespace> 2

- View logs from the manager container.
- 2 If you used the **make deploy** command to run the Operator as a deployment, use the project name>-system namespace.

# **Example output**

```
{"level":"info", "ts":1612732105.0579333, "logger": "cmd", "msg": "Version", "Go
Version": "go1.15.5", "GOOS": "linux", "GOARCH": "amd64", "ansible-
operator":"v1.10.1","commit":"1abf57985b43bf6a59dcd18147b3c574fa57d3f6"}
{"level":"info","ts":1612732105.0587437,"logger":"cmd","msg":"WATCH_NAMESPACE
environment variable not set. Watching all namespaces.","Namespace":""}
10207 21:08:26.110949
                           7 request.go:645] Throttling request took 1.035521578s, request:
GET:https://172.30.0.1:443/apis/flowcontrol.apiserver.k8s.io/v1alpha1?timeout=32s
{"level":"info", "ts":1612732107.768025, "logger": "controller-runtime.metrics", "msg": "metrics"
server is starting to listen","addr":"127.0.0.1:8080"}
{"level":"info", "ts":1612732107.768796, "logger": "watches", "msg": "Environment variable not
set; using default
value","envVar":"ANSIBLE_VERBOSITY_MEMCACHED_CACHE_EXAMPLE_COM","default":
{"level":"info", "ts":1612732107.7688773, "logger": "cmd", "msg": "Environment variable not set;
using default
value","Namespace":"","envVar":"ANSIBLE DEBUG LOGS","ANSIBLE DEBUG LOGS":fals
{"level":"info", "ts":1612732107.7688901, "logger": "ansible-controller", "msg": "Watching
resource","Options.Group":"cache.example.com","Options.Version":"v1","Options.Kind":"Memc
ached"}
{"level":"info","ts":1612732107.770032,"logger":"proxy","msg":"Starting to
serve","Address":"127.0.0.1:8888"}
10207 21:08:27.770185
                          7 leaderelection.go:243] attempting to acquire leader lease
memcached-operator-system/memcached-operator...
{"level":"info", "ts":1612732107.770202, "logger": "controller-runtime.manager", "msg": "starting
metrics server","path":"/metrics"}
10207 21:08:27.784854
                           7 leaderelection.go:253] successfully acquired lease
```

```
memcached-operator-system/memcached-operator {"level":"info","ts":1612732107.7850506,"logger":"controller-runtime.manager.controller.memcached-controller","msg":"Starting EventSource","source":"kind source: cache.example.com/v1, Kind=Memcached"} {"level":"info","ts":1612732107.8853772,"logger":"controller-runtime.manager.controller.memcached-controller","msg":"Starting Controller"} {"level":"info","ts":1612732107.8854098,"logger":"controller-runtime.manager.controller.memcached-controller","msg":"Starting workers","worker count":4}
```

### 5.4.7.4.2. Enabling full Ansible results in logs

You can set the environment variable **ANSIBLE\_DEBUG\_LOGS** to **True** to enable checking the full Ansible result in logs, which can be helpful when debugging.

#### **Procedure**

 Edit the config/manager/manager.yaml and config/default/manager\_auth\_proxy\_patch.yaml files to include the following configuration:

```
containers:name: managerenv:name: ANSIBLE_DEBUG_LOGSvalue: "True"
```

# 5.4.7.4.3. Enabling verbose debugging in logs

While developing an Ansible-based Operator, it can be helpful to enable additional debugging in logs.

### Procedure

• Add the **ansible.sdk.operatorframework.io/verbosity** annotation to your custom resource to enable the verbosity level that you want. For example:

```
apiVersion: "cache.example.com/v1alpha1" kind: "Memcached" metadata: name: "example-memcached" annotations: "ansible.sdk.operatorframework.io/verbosity": "4" spec: size: 4
```

# 5.4.8. Custom resource status management

# 5.4.8.1. About custom resource status in Ansible-based Operators

Ansible-based Operators automatically update custom resource (CR) **status** subresources with generic information about the previous Ansible run. This includes the number of successful and failed tasks and relevant error messages as shown:

status:

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

Ansible-based Operators also allow Operator authors to supply custom status values with the **k8s\_status** Ansible module, which is included in the **operator\_sdk.util** collection. This allows the author to update the **status** from within Ansible with any key-value pair as desired.

By default, Ansible-based Operators always include the generic Ansible run output as shown above. If you would prefer your application did *not* update the status with Ansible output, you can track the status manually from your application.

# 5.4.8.2. Tracking custom resource status manually

You can use the **operator\_sdk.util** collection to modify your Ansible-based Operator to track custom resource (CR) status manually from your application.

### **Prerequisites**

type: Running

• Ansible-based Operator project created by using the Operator SDK

### Procedure

1. Update the watches.yaml file with a manageStatus field set to false:

version: v1 group: api.example.com kind: <kind> role: <role> manageStatus: false

2. Use the **operator\_sdk.util.k8s\_status** Ansible module to update the subresource. For example, to update with key **test** and value **data**, **operator\_sdk.util** can be used as shown:

```
- operator_sdk.util.k8s_status:
    api_version: app.example.com/v1
    kind: <kind>
    name: "{{ ansible_operator_meta.name }}"
```

```
namespace: "{{ ansible_operator_meta.namespace }}"
status:
test: data
```

3. You can declare collections in the **meta/main.yml** file for the role, which is included for scaffolded Ansible-based Operators:

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

4. After declaring collections in the role meta, you can invoke the **k8s\_status** module directly:

```
k8s_status:
...
status:
key1: value1
```

### 5.5. HELM-BASED OPERATORS

# 5.5.1. Getting started with Operator SDK for Helm-based Operators

The Operator SDK includes options for generating an Operator project that leverages existing Helm charts to deploy Kubernetes resources as a unified application, without having to write any Go code.

To demonstrate the basics of setting up and running an Helm-based Operator using tools and libraries provided by the Operator SDK, Operator developers can build an example Helm-based Operator for Nginx and deploy it to a cluster.

### 5.5.1.1. Prerequisites

- Operator SDK CLI installed
- OpenShift CLI (oc) v4.13+ installed
- Logged into an OpenShift Container Platform 4.13 cluster with **oc** with an account that has **cluster-admin** permissions
- To allow the cluster to pull the image, the repository where you push your image must be set as public, or you must configure an image pull secret

### Additional resources

- Installing the Operator SDK CLI
- Getting started with the OpenShift CLI

# 5.5.1.2. Creating and deploying Helm-based Operators

You can build and deploy a simple Helm-based Operator for Nginx by using the Operator SDK.

# **Procedure**

1. Create a project.

a. Create your project directory:

\$ mkdir nginx-operator

b. Change into the project directory:

\$ cd nginx-operator

c. Run the **operator-sdk init** command with the **helm** plugin to initialize the project:

```
$ operator-sdk init \
--plugins=helm
```

### 2. Create an API.

Create a simple Nginx API:

\$ operator-sdk create api \

- --group demo \
  - --version v1 \
  - --kind Nginx

This API uses the built-in Helm chart boilerplate from the **helm create** command.

### 3. Build and push the Operator image.

Use the default **Makefile** targets to build and push your Operator. Set **IMG** with a pull spec for your image that uses a registry you can push to:

\$ make docker-build docker-push IMG=<registry>/<user>/<image\_name>:<tag>

# 4. Run the Operator.

- a. Install the CRD:
  - \$ make install
- b. Deploy the project to the cluster. Set **IMG** to the image that you pushed:
  - \$ make deploy IMG=<registry>/<user>/<image\_name>:<tag>

### 5. Add a security context constraint (SCC).

The Nginx service account requires privileged access to run in OpenShift Container Platform. Add the following SCC to the service account for the **nginx-sample** pod:

\$ oc adm policy add-scc-to-user \
 anyuid system:serviceaccount:nginx-operator-system:nginx-sample

# 6. Create a sample custom resource (CR).

a. Create a sample CR:

```
$ oc apply -f config/samples/demo_v1_nginx.yaml \
    -n nginx-operator-system
```

b. Watch for the CR to reconcile the Operator:

\$ oc logs deployment.apps/nginx-operator-controller-manager \

- -c manager \
- -n nginx-operator-system

#### 7. Delete a CR.

Delete a CR by running the following command:

\$ oc delete -f config/samples/demo\_v1\_nginx -n nginx-operator-system

#### 8. Clean up.

Run the following command to clean up the resources that have been created as part of this procedure:

\$ make undeploy

# 5.5.1.3. Next steps

• See Operator SDK tutorial for Helm-based Operators for a more in-depth walkthrough on building a Helm-based Operator.

# 5.5.2. Operator SDK tutorial for Helm-based Operators

Operator developers can take advantage of Helm support in the Operator SDK to build an example Helm-based Operator for Nginx and manage its lifecycle. This tutorial walks through the following process:

- Create a Nginx deployment
- Ensure that the deployment size is the same as specified by the **Nginx** custom resource (CR) spec
- Update the Nginx CR status using the status writer with the names of the nginx pods

This process is accomplished using two centerpieces of the Operator Framework:

#### Operator SDK

The operator-sdk CLI tool and controller-runtime library API

### Operator Lifecycle Manager (OLM)

Installation, upgrade, and role-based access control (RBAC) of Operators on a cluster



### **NOTE**

This tutorial goes into greater detail than Getting started with Operator SDK for Helmbased Operators.

### 5.5.2.1. Prerequisites

- Operator SDK CLI installed
- OpenShift CLI (oc) v4.13+ installed

- Logged into an OpenShift Container Platform 4.13 cluster with **oc** with an account that has **cluster-admin** permissions
- To allow the cluster to pull the image, the repository where you push your image must be set as public, or you must configure an image pull secret

#### Additional resources

- Installing the Operator SDK CLI
- Getting started with the OpenShift CLI

# 5.5.2.2. Creating a project

Use the Operator SDK CLI to create a project called **nginx-operator**.

### **Procedure**

- 1. Create a directory for the project:
  - \$ mkdir -p \$HOME/projects/nginx-operator
- 2. Change to the directory:
  - \$ cd \$HOME/projects/nginx-operator
- 3. Run the **operator-sdk init** command with the **helm** plugin to initialize the project:
  - \$ operator-sdk init \
    - --plugins=helm \
    - --domain=example.com \
    - --group=demo \
    - --version=v1 \
    - --kind=Nginx



#### NOTE

By default, the **helm** plugin initializes a project using a boilerplate Helm chart. You can use additional flags, such as the **--helm-chart** flag, to initialize a project using an existing Helm chart.

The **init** command creates the **nginx-operator** project specifically for watching a resource with API version **example.com/v1** and kind **Nginx**.

4. For Helm-based projects, the init command generates the RBAC rules in the config/rbac/role.yaml file based on the resources that would be deployed by the default manifest for the chart. Verify that the rules generated in this file meet the permission requirements of the Operator.

### 5.5.2.2.1. Existing Helm charts

Instead of creating your project with a boilerplate Helm chart, you can alternatively use an existing chart, either from your local file system or a remote chart repository, by using the following flags:

- --helm-chart
- --helm-chart-repo
- --helm-chart-version

If the **--helm-chart** flag is specified, the **--group**, **--version**, and **--kind** flags become optional. If left unset, the following default values are used:

Flag	Value
domain	my.domain
group	charts
version	v1
kind	Deduced from the specified chart

If the **--helm-chart** flag specifies a local chart archive, for example **example-chart-1.2.0.tgz**, or directory, the chart is validated and unpacked or copied into the project. Otherwise, the Operator SDK attempts to fetch the chart from a remote repository.

If a custom repository URL is not specified by the **--helm-chart-repo** flag, the following chart reference formats are supported:

Format	Description
<repo_name>/&lt; chart_name&gt;</repo_name>	Fetch the Helm chart named <b><chart_name></chart_name></b> from the helm chart repository named <b><repo_name></repo_name></b> , as specified in the <b>\$HELM_HOME/repositories/repositories.yaml</b> file. Use the <b>helm repo add</b> command to configure this file.
<url></url>	Fetch the Helm chart archive at the specified URL.

If a custom repository URL is specified by **--helm-chart-repo**, the following chart reference format is supported:

Format	Description
<chart_name></chart_name>	Fetch the Helm chart named <b><chart_name></chart_name></b> in the Helm chart repository specified by the <b>helm-chart-repo</b> URL value.

If the **--helm-chart-version** flag is unset, the Operator SDK fetches the latest available version of the Helm chart. Otherwise, it fetches the specified version. The optional **--helm-chart-version** flag is not used when the chart specified with the **--helm-chart** flag refers to a specific version, for example when it is a local path or a URL.

For more details and examples, run:

\$ operator-sdk init --plugins helm --help

#### 5.5.2.2.2. PROJECT file

Among the files generated by the **operator-sdk init** command is a Kubebuilder **PROJECT** file. Subsequent **operator-sdk** commands, as well as **help** output, that are run from the project root read this file and are aware that the project type is Helm. For example:

```
domain: example.com
layout:
- helm.sdk.operatorframework.io/v1
plugins:
 manifests.sdk.operatorframework.io/v2: {}
 scorecard.sdk.operatorframework.io/v2: {}
 sdk.x-openshift.io/v1: {}
projectName: nginx-operator
resources:
- api:
  crdVersion: v1
  namespaced: true
 domain: example.com
 group: demo
 kind: Nginx
 version: v1
version: "3"
```

# 5.5.2.3. Understanding the Operator logic

For this example, the **nginx-operator** project executes the following reconciliation logic for each **Nginx** custom resource (CR):

- Create an Nginx deployment if it does not exist.
- Create an Nginx service if it does not exist.
- Create an Nginx ingress if it is enabled and does not exist.
- Ensure that the deployment, service, and optional ingress match the desired configuration as specified by the **Nginx** CR, for example the replica count, image, and service type.

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

```
# Use the 'create api' subcommand to add watches to this file.
- group: demo
version: v1
kind: Nginx
chart: helm-charts/nginx
# +kubebuilder:scaffold:watch
```

### 5.5.2.3.1. Sample Helm chart

When a Helm Operator project is created, the Operator SDK creates a sample Helm chart that contains a set of templates for a simple Nginx release.

For this example, templates are available for deployment, service, and ingress resources, along with a **NOTES.txt** template, which Helm chart developers use to convey helpful information about a release.

If you are not already familiar with Helm charts, review the Helm developer documentation.

### 5.5.2.3.2. Modifying the custom resource spec

Helm uses a concept called values to provide customizations to the defaults of a Helm chart, which are defined in the **values.yaml** file.

You can override these defaults by setting the desired values in the custom resource (CR) spec. You can use the number of replicas as an example.

#### **Procedure**

The helm-charts/nginx/values.yaml file has a value called replicaCount set to 1 by default. To have two Nginx instances in your deployment, your CR spec must contain replicaCount: 2.
 Edit the config/samples/demo v1 nginx.yaml file to set replicaCount: 2:

```
apiVersion: demo.example.com/v1 kind: Nginx metadata: name: nginx-sample ... spec: ... replicaCount: 2
```

 Similarly, the default service port is set to 80. To use 8080, edit the config/samples/demo\_v1\_nginx.yaml file to set spec.port: 8080, which adds the service port override:

```
apiVersion: demo.example.com/v1
kind: Nginx
metadata:
name: nginx-sample
spec:
replicaCount: 2
service:
port: 8080
```

The Helm Operator applies the entire spec as if it was the contents of a values file, just like the **helm install -f ./overrides.yaml** command.

### 5.5.2.4. Enabling proxy support

Operator authors can develop Operators that support network proxies. Cluster administrators configure proxy support for the environment variables that are handled by Operator Lifecycle Manager (OLM). To support proxied clusters, your Operator must inspect the environment for the following standard proxy variables and pass the values to Operands:

- HTTP PROXY
- HTTPS\_PROXY

# NO\_PROXY



#### NOTE

This tutorial uses **HTTP\_PROXY** as an example environment variable.

# **Prerequisites**

• A cluster with cluster-wide egress proxy enabled.

### Procedure

1. Edit the **watches.yaml** file to include overrides based on an environment variable by adding the **overrideValues** field:

```
...
- group: demo.example.com
version: v1alpha1
kind: Nginx
chart: helm-charts/nginx
overrideValues:
proxy.http: $HTTP_PROXY
...
```

2. Add the proxy.http value in the helm-charts/nginx/values.yaml file:

```
...
proxy:
http: ""
https: ""
no_proxy: ""
```

3. To make sure the chart template supports using the variables, edit the chart template in the **helm-charts/nginx/templates/deployment.yaml** file to contain the following:

```
containers:
    - name: {{ .Chart.Name }}
    securityContext:
     - toYaml {{ .Values.securityContext | nindent 12 }}
    image: "{{ .Values.image.repository }}:{{ .Values.image.tag | default .Chart.AppVersion }}"
    imagePullPolicy: {{ .Values.image.pullPolicy }}
    env:
     - name: http_proxy
     value: "{{ .Values.proxy.http }}"
```

4. Set the environment variable on the Operator deployment by adding the following to the **config/manager/manager.yaml** file:

```
containers:
- args:
- --leader-elect
- --leader-election-id=ansible-proxy-demo
image: controller:latest
name: manager
```

env:

name: "HTTP\_PROXY" value: "http\_proxy\_test"

# 5.5.2.5. Running the Operator

There are three ways you can use the Operator SDK CLI to build and run your Operator:

- Run locally outside the cluster as a Go program.
- Run as a deployment on the cluster.
- Bundle your Operator and use Operator Lifecycle Manager (OLM) to deploy on the cluster.

# 5.5.2.5.1. Running locally outside the cluster

You can run your Operator project as a Go program outside of the cluster. This is useful for development purposes to speed up deployment and testing.

#### **Procedure**

 Run the following command to install the custom resource definitions (CRDs) in the cluster configured in your ~/.kube/config file and run the Operator locally:

\$ make install run

# **Example output**

```
{"level":"info","ts":1612652419.9289865,"logger":"controller-runtime.metrics","msg":"metrics server is starting to listen","addr":":8080"}
{"level":"info","ts":1612652419.9296563,"logger":"helm.controller","msg":"Watching resource","apiVersion":"demo.example.com/v1","kind":"Nginx","namespace":"","reconcilePeriod ":"1m0s"}
{"level":"info","ts":1612652419.929983,"logger":"controller-runtime.manager","msg":"starting metrics server","path":"/metrics"}
{"level":"info","ts":1612652419.930015,"logger":"controller-runtime.manager.controller.nginx-controller","msg":"Starting EventSource","source":"kind source: demo.example.com/v1, Kind=Nginx"}
{"level":"info","ts":1612652420.2307851,"logger":"controller-runtime.manager.controller.nginx-controller","msg":"Starting Controller"}
{"level":"info","ts":1612652420.2309358,"logger":"controller-runtime.manager.controller.nginx-controller","msg":"Starting Controller"}
```

### 5.5.2.5.2. Running as a deployment on the cluster

You can run your Operator project as a deployment on your cluster.

controller", "msg": "Starting workers", "worker count": 8}

### Procedure

1. Run the following **make** commands to build and push the Operator image. Modify the **IMG** argument in the following steps to reference a repository that you have access to. You can obtain an account for storing containers at repository sites such as Quay.io.

a. Build the image:

\$ make docker-build IMG=<registry>/<user>/<image\_name>:<tag>



#### NOTE

The Dockerfile generated by the SDK for the Operator explicitly references **GOARCH=amd64** for **go build**. This can be amended to **GOARCH=\$TARGETARCH** for non-AMD64 architectures. Docker will automatically set the environment variable to the value specified by – **platform**. With Buildah, the –**build-arg** will need to be used for the purpose. For more information, see Multiple Architectures.

b. Push the image to a repository:

\$ make docker-push IMG=<registry>/<user>/<image\_name>:<tag>



#### NOTE

The name and tag of the image, for example **IMG**= <registry>/<user>/<image\_name>:<tag>, in both the commands can also be set in your Makefile. Modify the **IMG** ?= controller:latest value to set your default image name.

2. Run the following command to deploy the Operator:

\$ make deploy IMG=<registry>/<user>/<image\_name>:<tag>

By default, this command creates a namespace with the name of your Operator project in the form **<project\_name>-system** and is used for the deployment. This command also installs the RBAC manifests from **config/rbac**.

3. Run the following command to verify that the Operator is running:

# **Example output**

# 5.5.2.5.3. Bundling an Operator and deploying with Operator Lifecycle Manager

#### 5.5.2.5.3.1. Bundling an Operator

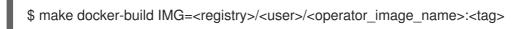
The Operator bundle format is the default packaging method for Operator SDK and Operator Lifecycle Manager (OLM). You can get your Operator ready for use on OLM by using the Operator SDK to build and push your Operator project as a bundle image.

### **Prerequisites**

- Operator SDK CLI installed on a development workstation
- OpenShift CLI (oc) v4.13+ installed
- Operator project initialized by using the Operator SDK

#### Procedure

- 1. Run the following **make** commands in your Operator project directory to build and push your Operator image. Modify the **IMG** argument in the following steps to reference a repository that you have access to. You can obtain an account for storing containers at repository sites such as Quay.io.
  - a. Build the image:





#### NOTE

The Dockerfile generated by the SDK for the Operator explicitly references **GOARCH=amd64** for **go build**. This can be amended to **GOARCH=\$TARGETARCH** for non-AMD64 architectures. Docker will automatically set the environment variable to the value specified by – **platform**. With Buildah, the –**build-arg** will need to be used for the purpose. For more information, see Multiple Architectures.

- b. Push the image to a repository:
  - \$ make docker-push IMG=<registry>/<user>/<operator\_image\_name>:<tag>
- Create your Operator bundle manifest by running the make bundle command, which invokes several commands, including the Operator SDK generate bundle and bundle validate subcommands:
  - \$ make bundle IMG=<registry>/<user>/<operator\_image\_name>:<tag>

Bundle manifests for an Operator describe how to display, create, and manage an application. The **make bundle** command creates the following files and directories in your Operator project:

- A bundle manifests directory named bundle/manifests that contains a ClusterServiceVersion object
- A bundle metadata directory named **bundle/metadata**
- All custom resource definitions (CRDs) in a **config/crd** directory
- A Dockerfile bundle.Dockerfile

These files are then automatically validated by using **operator-sdk bundle validate** to ensure the on-disk bundle representation is correct.

3. Build and push your bundle image by running the following commands. OLM consumes Operator bundles using an index image, which reference one or more bundle images.

and the second of the second o

- a. Build the bundle image. Set **BUNDLE\_IMG** with the details for the registry, user namespace, and image tag where you intend to push the image:
  - \$ make bundle-build BUNDLE\_IMG=<registry>/<user>/<bundle\_image\_name>:<tag>
- b. Push the bundle image:
  - \$ docker push <registry>/<user>/<bundle\_image\_name>:<tag>

# 5.5.2.5.3.2. Deploying an Operator with Operator Lifecycle Manager

Operator Lifecycle Manager (OLM) helps you to install, update, and manage the lifecycle of Operators and their associated services on a Kubernetes cluster. OLM is installed by default on OpenShift Container Platform and runs as a Kubernetes extension so that you can use the web console and the OpenShift CLI (oc) for all Operator lifecycle management functions without any additional tools.

The Operator bundle format is the default packaging method for Operator SDK and OLM. You can use the Operator SDK to quickly run a bundle image on OLM to ensure that it runs properly.

# **Prerequisites**

- Operator SDK CLI installed on a development workstation
- Operator bundle image built and pushed to a registry
- OLM installed on a Kubernetes-based cluster (v1.16.0 or later if you use **apiextensions.k8s.io/v1** CRDs, for example OpenShift Container Platform 4.13)
- Logged in to the cluster with oc using an account with cluster-admin permissions

### Procedure

• Enter the following command to run the Operator on the cluster:

\$ operator-sdk run bundle \1
-n <namespace> \2
<registry>/<user>/<bundle\_image\_name>:<tag> 3

- The **run bundle** command creates a valid file-based catalog and installs the Operator bundle on your cluster using OLM.
- Optional: By default, the command installs the Operator in the currently active project in your ~/.kube/config file. You can add the -n flag to set a different namespace scope for the installation.
- If you do not specify an image, the command uses **quay.io/operator-framework/opm:latest** as the default index image. If you specify an image, the command uses the bundle image itself as the index image.



#### **IMPORTANT**

As of OpenShift Container Platform 4.11, the **run bundle** command supports the file-based catalog format for Operator catalogs by default. The deprecated SQLite database format for Operator catalogs continues to be supported; however, it will be removed in a future release. It is recommended that Operator authors migrate their workflows to the file-based catalog format.

This command performs the following actions:

- Create an index image referencing your bundle image. The index image is opaque and ephemeral, but accurately reflects how a bundle would be added to a catalog in production.
- Create a catalog source that points to your new index image, which enables OperatorHub to discover your Operator.
- Deploy your Operator to your cluster by creating an OperatorGroup, Subscription, InstallPlan, and all other required resources, including RBAC.

# 5.5.2.6. Creating a custom resource

After your Operator is installed, you can test it by creating a custom resource (CR) that is now provided on the cluster by the Operator.

### **Prerequisites**

• Example Nginx Operator, which provides the Nginx CR, installed on a cluster

### **Procedure**

- 1. Change to the namespace where your Operator is installed. For example, if you deployed the Operator using the **make deploy** command:
  - \$ oc project nginx-operator-system
- 2. Edit the sample **Nginx** CR manifest at **config/samples/demo\_v1\_nginx.yaml** to contain the following specification:

```
apiVersion: demo.example.com/v1 kind: Nginx metadata: name: nginx-sample ... spec: ... replicaCount: 3
```

3. The Nginx service account requires privileged access to run in OpenShift Container Platform. Add the following security context constraint (SCC) to the service account for the **nginx-sample** pod:

```
$ oc adm policy add-scc-to-user \
anyuid system:serviceaccount:nginx-operator-system:nginx-sample
```

4. Create the CR:

\$ oc apply -f config/samples/demo\_v1\_nginx.yaml

5. Ensure that the **Nginx** Operator creates the deployment for the sample CR with the correct size:

\$ oc get deployments

# **Example output**

```
NAME READY UP-TO-DATE AVAILABLE AGE nginx-operator-controller-manager 1/1 1 1 8m nginx-sample 3/3 3 3 1m
```

- 6. Check the pods and CR status to confirm the status is updated with the Nginx pod names.
  - a. Check the pods:

\$ oc get pods

# Example output

```
NAME READY STATUS RESTARTS AGE nginx-sample-6fd7c98d8-7dqdr 1/1 Running 0 1m nginx-sample-6fd7c98d8-g5k7v 1/1 Running 0 1m nginx-sample-6fd7c98d8-m7vn7 1/1 Running 0 1m
```

b. Check the CR status:

\$ oc get nginx/nginx-sample -o yaml

# **Example output**

```
apiVersion: demo.example.com/v1 kind: Nginx metadata: ... name: nginx-sample ... spec: replicaCount: 3 status: nodes: - nginx-sample-6fd7c98d8-7dqdr - nginx-sample-6fd7c98d8-g5k7v - nginx-sample-6fd7c98d8-m7vn7
```

- 7. Update the deployment size.
  - a. Update config/samples/demo\_v1\_nginx.yaml file to change the spec.size field in the Nginx CR from 3 to 5:

```
$ oc patch nginx nginx-sample \
  -p '{"spec":{"replicaCount": 5}}' \
  --type=merge
```

b. Confirm that the Operator changes the deployment size:

\$ oc get deployments

# **Example output**

```
NAME READY UP-TO-DATE AVAILABLE AGE nginx-operator-controller-manager 1/1 1 1 10m nginx-sample 5/5 5 5 3m
```

8. Delete the CR by running the following command:

\$ oc delete -f config/samples/demo\_v1\_nginx.yaml

- 9. Clean up the resources that have been created as part of this tutorial.
  - If you used the **make deploy** command to test the Operator, run the following command:

\$ make undeploy

 If you used the operator-sdk run bundle command to test the Operator, run the following command:

\$ operator-sdk cleanup project\_name>

### 5.5.2.7. Additional resources

- See Project layout for Helm-based Operators to learn about the directory structures created by the Operator SDK.
- If a cluster-wide egress proxy is configured, cluster administrators can override the proxy settings or inject a custom CA certificate for specific Operators running on Operator Lifecycle Manager (OLM).

# 5.5.3. Project layout for Helm-based Operators

The **operator-sdk** CLI can generate, or *scaffold*, a number of packages and files for each Operator project.

# 5.5.3.1. Helm-based project layout

Helm-based Operator projects generated using the **operator-sdk init --plugins helm** command contain the following directories and files:

File/folders	Purpose
config/	Kustomize manifests for deploying the Operator on a Kubernetes cluster.

File/folders	Purpose
helm-charts/	Helm chart initialized with the <b>operator-sdk create api</b> command.
Dockerfile	Used to build the Operator image with the <b>make docker-build</b> command.
watches.yaml	Group/version/kind (GVK) and Helm chart location.
Makefile	Targets used to manage the project.
PROJECT	YAML file containing metadata information for the Operator.

# 5.5.4. Updating Helm-based projects for newer Operator SDK versions

OpenShift Container Platform 4.13 supports Operator SDK 1.28.0. If you already have the 1.25.4 CLI installed on your workstation, you can update the CLI to 1.28.0 by installing the latest version.

However, to ensure your existing Operator projects maintain compatibility with Operator SDK 1.28.0, update steps are required for the associated breaking changes introduced since 1.25.4. You must perform the update steps manually in any of your Operator projects that were previously created or maintained with 1.25.4.

# 5.5.4.1. Updating Helm-based Operator projects for Operator SDK 1.28.0

The following procedure updates an existing Helm-based Operator project for compatibility with 1.28.0.

### **Prerequisites**

- Operator SDK 1.28.0 installed
- An Operator project created or maintained with Operator SDK 1.25.4

### Procedure

1. Update the image tag in your Operator's Dockerfile as shown in the following example:

### **Example Dockerfile**

FROM registry.redhat.io/openshift4/ose-helm-operator:v4.13 1

- Update the version tag to **v4.13**.
- 2. Find the **ose-kube-rbac-proxy** pull spec in the following files, and update the image tag to **v4.13**:
  - config/default/manager\_auth\_proxy\_patch.yaml
  - bundle/manifests/memcached-operator.clusterserviceversion.yaml

. . .

containers:

- name: kube-rbac-proxy

image: registry.redhat.io/openshift4/ose-kube-rbac-proxy:v4.13 1

. . .

1

Update the tag version from v4.12 to v4.13.

#### 5.5.4.2. Additional resources

- Migrating package manifest projects to bundle format
- Upgrading projects for Operator SDK 1.16.0
- Upgrading projects for Operator SDK v1.10.1
- Upgrading projects for Operator SDK v1.8.0

# 5.5.5. Helm support in Operator SDK

#### 5.5.5.1. Helm charts

One of the Operator SDK options for generating an Operator project includes leveraging an existing Helm chart to deploy Kubernetes resources as a unified application, without having to write any Go code. Such Helm-based Operators are designed to excel at stateless applications that require very little logic when rolled out, because changes should be applied to the Kubernetes objects that are generated as part of the chart. This may sound limiting, but can be sufficient for a surprising amount of use-cases as shown by the proliferation of Helm charts built by the Kubernetes community.

The main function of an Operator is to read from a custom object that represents your application instance and have its desired state match what is running. In the case of a Helm-based Operator, the **spec** field of the object is a list of configuration options that are typically described in the Helm **values.yaml** file. Instead of setting these values with flags using the Helm CLI (for example, **helm install -f values.yaml**), you can express them within a custom resource (CR), which, as a native Kubernetes object, enables the benefits of RBAC applied to it and an audit trail.

For an example of a simple CR called **Tomcat**:

apiVersion: apache.org/v1alpha1

kind: Tomcat metadata:

name: example-app

spec:

replicaCount: 2

The **replicaCount** value, **2** in this case, is propagated into the template of the chart where the following is used:

{{ .Values.replicaCount }}

After an Operator is built and deployed, you can deploy a new instance of an app by creating a new instance of a CR, or list the different instances running in all environments using the **oc** command:

\$ oc get Tomcats --all-namespaces

There is no requirement use the Helm CLI or install Tiller; Helm-based Operators import code from the Helm project. All you have to do is have an instance of the Operator running and register the CR with a custom resource definition (CRD). Because it obeys RBAC, you can more easily prevent production changes.

# 5.5.6. Operator SDK tutorial for Hybrid Helm Operators

The standard Helm-based Operator support in the Operator SDK has limited functionality compared to the Go-based and Ansible-based Operator support that has reached the Auto Pilot capability (level V) in the Operator maturity model.

The Hybrid Helm Operator enhances the existing Helm-based support's abilities through Go APIs. With this hybrid approach of Helm and Go, the Operator SDK enables Operator authors to use the following process:

- Generate a default structure for, or scaffold, a Go API in the same project as Helm.
- Configure the Helm reconciler in the **main.go** file of the project, through the libraries provided by the Hybrid Helm Operator.



#### **IMPORTANT**

The Hybrid Helm Operator 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.

This tutorial walks through the following process using the Hybrid Helm Operator:

- Create a **Memcached** deployment through a Helm chart if it does not exist
- Ensure that the deployment size is the same as specified by Memcached custom resource (CR) spec
- Create a MemcachedBackup deployment by using the Go API

# 5.5.6.1. Prerequisites

- Operator SDK CLI installed
- OpenShift CLI (oc) v4.13+ installed
- Logged into an OpenShift Container Platform 4.13 cluster with oc with an account that has cluster-admin permissions
- To allow the cluster to pull the image, the repository where you push your image must be set as public, or you must configure an image pull secret

#### Additional resources

Installing the Operator SDK CLI

Getting started with the OpenShift CLI

# 5.5.6.2. Creating a project

Use the Operator SDK CLI to create a project called **memcached-operator**.

#### Procedure

- 1. Create a directory for the project:
  - \$ mkdir -p \$HOME/github.com/example/memcached-operator
- 2. Change to the directory:
  - \$ cd \$HOME/github.com/example/memcached-operator
- 3. Run the **operator-sdk init** command to initialize the project. Use a domain of **example.com** so that all API groups are **<group>.example.com**:
  - \$ operator-sdk init \
    - --plugins=hybrid.helm.sdk.operatorframework.io \
    - --project-version="3" \
    - --domain example.com \
    - --repo=github.com/example/memcached-operator

The **init** command generates the RBAC rules in the **config/rbac/role.yaml** file based on the resources that would be deployed by the chart's default manifests. Verify that the rules generated in the **config/rbac/role.yaml** file meet your Operator's permission requirements.

# Additional resources

• This procedure creates a project structure that is compatible with both Helm and Go APIs. To learn more about the project directory structure, see Project layout.

# 5.5.6.3. Creating a Helm API

Use the Operator SDK CLI to create a Helm API.

#### Procedure

Run the following command to create a Helm API with group cache, version v1, and kind
 Memcached:

\$ operator-sdk create api \

- --plugins helm.sdk.operatorframework.io/v1 \
- --group cache \
- --version v1 \
- --kind Memcached



#### **NOTE**

This procedure also configures your Operator project to watch the **Memcached** resource with API version **v1** and scaffolds a boilerplate Helm chart. Instead of creating the project from the boilerplate Helm chart scaffolded by the Operator SDK, you can alternatively use an existing chart from your local file system or remote chart repository.

For more details and examples for creating Helm API based on existing or new charts, run the following command:

\$ operator-sdk create api --plugins helm.sdk.operatorframework.io/v1 --help

#### Additional resources

Existing Helm charts

### 5.5.6.3.1. Operator logic for the Helm API

By default, your scaffolded Operator project watches **Memcached** resource events as shown in the **watches.yaml** file and executes Helm releases using the specified chart.

# Example 5.2. Example watches.yaml file

# Use the 'create api' subcommand to add watches to this file.

- group: cache.my.domain

version: v1 kind: Memcached

chart: helm-charts/memcached #+kubebuilder:scaffold:watch

### Additional resources

• For detailed documentation on customizing the Helm Operator logic through the chart, see Understanding the Operator logic.

### 5.5.6.3.2. Custom Helm reconciler configurations using provided library APIs

A disadvantage of existing Helm-based Operators is the inability to configure the Helm reconciler, because it is abstracted from users. For a Helm-based Operator to reach the Seamless Upgrades capability (level II and later) that reuses an already existing Helm chart, a hybrid between the Go and Helm Operator types adds value.

The APIs provided in the **helm-operator-plugins** library allow Operator authors to make the following configurations:

- Customize value mapping based on cluster state
- Execute code in specific events by configuring the reconciler's event recorder
- Customize the reconciler's logger
- Setup **Install**, **Upgrade**, and **Uninstall** annotations to enable Helm's actions to be configured based on the annotations found in custom resources watched by the reconciler

• Configure the reconciler to run with **Pre** and **Post** hooks

The above configurations to the reconciler can be done in the **main.go** file:

# Example main.go file

```
// Operator's main.go
// With the help of helpers provided in the library, the reconciler can be
// configured here before starting the controller with this reconciler.
reconciler := reconciler.New(
  reconciler.WithChart(*chart),
  reconciler.WithGroupVersionKind(gvk),
)

if err := reconciler.SetupWithManager(mgr); err != nil {
  panic(fmt.Sprintf("unable to create reconciler: %s", err))
}
```

### 5.5.6.4. Creating a Go API

Use the Operator SDK CLI to create a Go API.

#### **Procedure**

1. Run the following command to create a Go API with group **cache**, version **v1**, and kind **MemcachedBackup**:

```
$ operator-sdk create api \
    --group=cache \
    --version v1 \
    --kind MemcachedBackup \
    --resource \
    --controller \
    --plugins=go/v3
```

2. When prompted, enter **y** for creating both resource and controller:

```
$ Create Resource [y/n]
y
Create Controller [y/n]
y
```

This procedure generates the **MemcachedBackup** resource API at api/v1/memcachedbackup\_types.go and the controller at controllers/memcachedbackup\_controller.go.

### 5.5.6.4.1. Defining the API

Define the API for the **MemcachedBackup** custom resource (CR).

Represent this Go API by defining the **MemcachedBackup** type, which will have a **MemcachedBackupSpec.Size** field to set the quantity of Memcached backup instances (CRs) to be deployed, and a **MemcachedBackupStatus.Nodes** field to store a CR's pod names.



#### NOTE

The **Node** field is used to illustrate an example of a **Status** field.

#### Procedure

1. Define the API for the **MemcachedBackup** CR by modifying the Go type definitions in the **api/v1/memcachedbackup\_types.go** file to have the following **spec** and **status**:

# Example 5.3. Example api/v1/memcachedbackup\_types.go file

```
// MemcachedBackupSpec defines the desired state of MemcachedBackup
type MemcachedBackupSpec struct {
    // INSERT ADDITIONAL SPEC FIELDS - desired state of cluster
    // Important: Run "make" to regenerate code after modifying this file

    //+kubebuilder:validation:Minimum=0
    // Size is the size of the memcached deployment
    Size int32 `json:"size"`
}

// MemcachedBackupStatus defines the observed state of MemcachedBackup
type MemcachedBackupStatus struct {
    // INSERT ADDITIONAL STATUS FIELD - define observed state of cluster
    // Important: Run "make" to regenerate code after modifying this file
    // Nodes are the names of the memcached pods
    Nodes []string `json:"nodes"`
}
```

2. Update the generated code for the resource type:

\$ make generate

# TIP

After you modify a \*\_types.go file, you must run the **make generate** command to update the generated code for that resource type.

3. After the API is defined with **spec** and **status** fields and CRD validation markers, generate and update the CRD manifests:

\$ make manifests

This Makefile target invokes the **controller-gen** utility to generate the CRD manifests in the **config/crd/bases/cache.my.domain memcachedbackups.yaml** file.

### 5.5.6.4.2. Controller implementation

The controller in this tutorial performs the following actions:

- Create a **Memcached** deployment if it does not exist.
- Ensure that the deployment size is the same as specified by the **Memcached** CR spec.

• Update the **Memcached** CR status with the names of the **memcached** pods.

For a detailed explanation on how to configure the controller to perform the above mentioned actions, see Implementing the controller in the Operator SDK tutorial for standard Go-based Operators.

### 5.5.6.4.3. Differences in main.go

For standard Go-based Operators and the Hybrid Helm Operator, the **main.go** file handles the scaffolding the initialization and running of the **Manager** program for the Go API. For the Hybrid Helm Operator, however, the **main.go** file also exposes the logic for loading the **watches.yaml** file and configuring the Helm reconciler.

# Example 5.4. Example main.go file

```
for _, w := range ws {
// Register controller with the factory
 reconcilePeriod := defaultReconcilePeriod
 if w.ReconcilePeriod != nil {
 reconcilePeriod = w.ReconcilePeriod.Duration
}
 maxConcurrentReconciles := defaultMaxConcurrentReconciles
 if w.MaxConcurrentReconciles != nil {
 maxConcurrentReconciles = *w.MaxConcurrentReconciles
}
 r, err := reconciler.New(
 reconciler.WithChart(*w.Chart),
 reconciler.WithGroupVersionKind(w.GroupVersionKind),
 reconciler.WithOverrideValues(w.OverrideValues),
 reconciler.SkipDependentWatches(w.WatchDependentResources != nil &&
!*w.WatchDependentResources),
 reconciler.WithMaxConcurrentReconciles(maxConcurrentReconciles),
 reconciler.WithReconcilePeriod(reconcilePeriod),
 reconciler.WithInstallAnnotations(annotation.DefaultInstallAnnotations...),
 reconciler.WithUpgradeAnnotations(annotation.DefaultUpgradeAnnotations...),
 reconciler.WithUninstallAnnotations(annotation.DefaultUninstallAnnotations...),
```

The manager is initialized with both **Helm** and **Go** reconcilers:

# Example 5.5. Example **Helm** and **Go** reconcilers

```
...
// Setup manager with Go API
if err = (&controllers.MemcachedBackupReconciler{
    Client: mgr.GetClient(),
    Scheme: mgr.GetScheme(),
}).SetupWithManager(mgr); err != nil {
    setupLog.Error(err, "unable to create controller", "controller", "MemcachedBackup")
    os.Exit(1)
}
```

```
// Setup manager with Helm API

for _, w := range ws {

...

if err := r.SetupWithManager(mgr); err != nil {

    setupLog.Error(err, "unable to create controller", "controller", "Helm")

    os.Exit(1)

}

setupLog.Info("configured watch", "gvk", w.GroupVersionKind, "chartPath", w.ChartPath,
"maxConcurrentReconciles", maxConcurrentReconciles, "reconcilePeriod", reconcilePeriod)

}

// Start the manager

if err := mgr.Start(ctrl.SetupSignalHandler()); err != nil {

    setupLog.Error(err, "problem running manager")

    os.Exit(1)

}
```

#### 5.5.6.4.4. Permissions and RBAC manifests

The controller requires certain role-based access control (RBAC) permissions to interact with the resources it manages. For the Go API, these are specified with RBAC markers, as shown in the Operator SDK tutorial for standard Go-based Operators.

For the Helm API, the permissions are scaffolded by default in **roles.yaml**. Currently, however, due to a known issue when the Go API is scaffolded, the permissions for the Helm API are overwritten. As a result of this issue, ensure that the permissions defined in **roles.yaml** match your requirements.



### **NOTE**

This known issue is being tracked in https://github.com/operator-framework/helm-operator-plugins/issues/142.

The following is an example **role.yaml** for a Memcached Operator:

# Example 5.6. Example Helm and Go reconcilers

```
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRole
metadata:
name: manager-role
rules:
- apiGroups:
- ""
resources:
- namespaces
verbs:
- get
- apiGroups:
- apps
```

# resources:

- deployments
- daemonsets
- replicasets
- statefulsets

### verbs:

- create
- delete
- get
- list
- patch
- update
- watch
- apiGroups:
- cache.my.domain

### resources:

- memcachedbackups

### verbs:

- create
- delete
- get
- list
- patch
- update
- watch
- apiGroups:
- cache.my.domain

# resources:

- memcachedbackups/finalizers

# verbs:

- create
- delete
- get
- list
- patch
- update
- watch
- apiGroups:
  - \_ '''

# resources:

- pods
- services
- services/finalizers
- endpoints
- persistentvolumeclaims
- events
- configmaps
- secrets
- serviceaccounts

### verbs:

- create
- delete
- get
- list
- patch
- update

- watch
- apiGroups:
- cache.my.domain

#### resources:

- memcachedbackups/status

### verbs:

- get
- patch
- update
- apiGroups:
- policy

#### resources:

- events
- poddisruptionbudgets

### verbs:

- create
- delete
- get
- list
- patch
- update
- watch
- apiGroups:
- cache.my.domain

#### resources:

- memcacheds
- memcacheds/status
- memcacheds/finalizers

#### verbs:

- create
- delete
- get
- list
- patch
- update
- watch

# Additional resources

• RBAC markers for Go-based Operators

# 5.5.6.5. Running locally outside the cluster

You can run your Operator project as a Go program outside of the cluster. This is useful for development purposes to speed up deployment and testing.

#### Procedure

• Run the following command to install the custom resource definitions (CRDs) in the cluster configured in your ~/.kube/config file and run the Operator locally:

\$ make install run

# 5.5.6.6. Running as a deployment on the cluster

You can run your Operator project as a deployment on your cluster.

#### Procedure

- 1. Run the following **make** commands to build and push the Operator image. Modify the **IMG** argument in the following steps to reference a repository that you have access to. You can obtain an account for storing containers at repository sites such as Quay.io.
  - a. Build the image:

\$ make docker-build IMG=<registry>/<user>/<image\_name>:<tag>



### **NOTE**

The Dockerfile generated by the SDK for the Operator explicitly references **GOARCH=amd64** for **go build**. This can be amended to **GOARCH=\$TARGETARCH** for non-AMD64 architectures. Docker will automatically set the environment variable to the value specified by – **platform**. With Buildah, the –**build-arg** will need to be used for the purpose. For more information, see Multiple Architectures.

b. Push the image to a repository:





#### **NOTE**

The name and tag of the image, for example **IMG**= <registry>/<user>/<image\_name>:<tag>, in both the commands can also be set in your Makefile. Modify the **IMG** ?= controller:latest value to set your default image name.

2. Run the following command to deploy the Operator:

\$ make deploy IMG=<registry>/<user>/<image\_name>:<tag>

By default, this command creates a namespace with the name of your Operator project in the form **<project\_name>-system** and is used for the deployment. This command also installs the RBAC manifests from **config/rbac**.

3. Run the following command to verify that the Operator is running:

# Example output

### 5.5.6.7. Creating custom resources

After your Operator is installed, you can test it by creating custom resources (CRs) that are now provided on the cluster by the Operator.

#### **Procedure**

1. Change to the namespace where your Operator is installed:

```
$ oc project project_name>-system
```

 Update the sample Memcached CR manifest at the config/samples/cache\_v1\_memcached.yaml file by updating the replicaCount field to 3:

Example 5.7. Example config/samples/cache\_v1\_memcached.yaml file

```
apiVersion: cache.my.domain/v1
kind: Memcached
metadata:
 name: memcached-sample
 affinity: {}
 autoscaling:
  enabled: false
  maxReplicas: 100
  minReplicas: 1
  targetCPUUtilizationPercentage: 80
 fullnameOverride: ""
 image:
  pullPolicy: IfNotPresent
  repository: nginx
  tag: ""
 imagePullSecrets: []
 ingress:
  annotations: {}
  className: ""
  enabled: false
  hosts:
  - host: chart-example.local
   paths:
   - path: /
    pathType: ImplementationSpecific
  tls: []
 nameOverride: ""
 nodeSelector: {}
 podAnnotations: {}
 podSecurityContext: {}
 replicaCount: 3
 resources: {}
 securityContext: {}
 service:
  port: 80
  type: ClusterIP
 serviceAccount:
  annotations: {}
```

create: true name: "" tolerations: []

3. Create the Memcached CR:

\$ oc apply -f config/samples/cache\_v1\_memcached.yaml

4. Ensure that the Memcached Operator creates the deployment for the sample CR with the correct size:

\$ oc get pods

# **Example output**

NAME	READY	STATUS	RESTAR	TS	AGE
memcached-sample-6fo	d7c98d8-7dqdr	1/1	Running	0	18m
memcached-sample-6fo	d7c98d8-g5k7v	1/1	Running	0	18m
memcached-sample-6fo	d7c98d8-m7vn	7 1/1	Running	0	18m

5. Update the sample **MemcachedBackup** CR manifest at the **config/samples/cache v1 memcachedbackup.yaml** file by updating the **size** to **2**:

Example 5.8. Example config/samples/cache\_v1\_memcachedbackup.yaml file

apiVersion: cache.my.domain/v1 kind: MemcachedBackup

metadata:

name: memcachedbackup-sample

spec: size: 2

6. Create the **MemcachedBackup** CR:

\$ oc apply -f config/samples/cache\_v1\_memcachedbackup.yaml

7. Ensure that the count of **memcachedbackup** pods is the same as specified in the CR:

\$ oc get pods

# Example output

NAME READY STATUS RESTARTS AGE memcachedbackup-sample-864969989-4bbzg 1/1 Running 0 22m memcachedbackup-sample-864969989-mq6mx 1/1 Running 0 22m

- 8. You can update the **spec** in each of the above CRs, and then apply them again. The controller reconciles again and ensures that the size of the pods is as specified in the **spec** of the respective CRs.
- 9. Clean up the resources that have been created as part of this tutorial:

a. Delete the **Memcached** resource:

\$ oc delete -f config/samples/cache\_v1\_memcached.yaml

b. Delete the **MemcachedBackup** resource:

\$ oc delete -f config/samples/cache\_v1\_memcachedbackup.yaml

c. If you used the **make deploy** command to test the Operator, run the following command:

\$ make undeploy

# 5.5.6.8. Project layout

The Hybrid Helm Operator scaffolding is customized to be compatible with both Helm and Go APIs.

File/folders	Purpose
Dockerfile	Instructions used by a container engine to build your Operator image with the <b>make</b> docker-build command.
Makefile	Build file with helper targets to help you work with your project.
PROJECT	YAML file containing metadata information for the Operator. Represents the project's configuration and is used to track useful information for the CLI and plugins.
bin/	Contains useful binaries such as the <b>manager</b> which is used to run your project locally and the <b>kustomize</b> utility used for the project configuration.

File/folders	Purpose
config/	Contains configuration files, including all Kustomize manifests, to launch your Operator project on a cluster. Plugins might use it to provide functionality. For example, for the Operator SDK to help create your Operator bundle, the CLI looks up the CRDs and CRs which are scaffolded in this directory.
	config/crd/ Contains custom resource definitions (CRDs).
	config/default/ Contains a Kustomize base for launching the controller in a standard configuration.
	config/manager/ Contains the manifests to launch your Operator project as pods on the cluster. config/manifests/
	Contains the base to generate your OLM manifests in the <b>bundle</b> / directory. <b>config/prometheus</b> /
	Contains the manifests required to enable project to serve metrics to Prometheus such as the <b>ServiceMonitor</b> resource.
	<pre>config/scorecard/     Contains the manifests required to allow you test your project with the scorecard     tool. config/rbac/</pre>
	Contains the RBAC permissions required to run your project.  config/samples/  Contains samples for custom resources.
api/	Contains the Go API definition.
controllers/	Contains the controllers for the Go API.
hack/	Contains utility files, such as the file used to scaffold the license header for your project files.
main.go	Main program of the Operator. Instantiates a new manager that registers all custom resource definitions (CRDs) in the <b>apis</b> / directory and starts all controllers in the <b>controllers</b> / directory.
helm-charts/	Contains the Helm charts which can be specified using the <b>create api</b> command with the Helm plugin.
watches.yaml	Contains group/version/kind (GVK) and Helm chart location. Used to configure the Helm watches.

# 5.5.7. Updating Hybrid Helm-based projects for newer Operator SDK versions

OpenShift Container Platform 4.13 supports Operator SDK 1.28.0. If you already have the 1.25.4 CLI installed on your workstation, you can update the CLI to 1.28.0 by installing the latest version.

However, to ensure your existing Operator projects maintain compatibility with Operator SDK 1.28.0,

update steps are required for the associated breaking changes introduced since 1.25.4. You must perform the update steps manually in any of your Operator projects that were previously created or maintained with 1.25.4.

# 5.5.7.1. Updating Hybrid Helm-based Operator projects for Operator SDK 1.28.0

The following procedure updates an existing Hybrid Helm-based Operator project for compatibility with 1.28.0.

### **Prerequisites**

- Operator SDK 1.28.0 installed
- An Operator project created or maintained with Operator SDK 1.25.4

#### **Procedure**

- Find the **ose-kube-rbac-proxy** pull spec in the following files, and update the image tag to **v4.13**:
  - config/default/manager\_auth\_proxy\_patch.yaml
  - o bundle/manifests/memcached-operator.clusterserviceversion.yaml

```
...
containers:
- name: kube-rbac-proxy
image: registry.redhat.io/openshift4/ose-kube-rbac-proxy:v4.13 1
...
```

Update the tag version from **v4.12** to **v4.13**.

### 5.5.7.2. Additional resources

- Migrating package manifest projects to bundle format
- Upgrading projects for Operator SDK 1.16.0
- Upgrading projects for Operator SDK v1.10.1
- Upgrading projects for Operator SDK v1.8.0

# 5.6. JAVA-BASED OPERATORS

# 5.6.1. Getting started with Operator SDK for Java-based Operators



### **IMPORTANT**

Java-based Operator SDK 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.

To demonstrate the basics of setting up and running a Java-based Operator using tools and libraries provided by the Operator SDK, Operator developers can build an example Java-based Operator for Memcached, a distributed key-value store, and deploy it to a cluster.

# 5.6.1.1. Prerequisites

- Operator SDK CLI installed
- OpenShift CLI (oc) v4.13+ installed
- Java v11+
- Maven v3.6.3+
- Logged into an OpenShift Container Platform 4.13 cluster with **oc** with an account that has **cluster-admin** permissions
- To allow the cluster to pull the image, the repository where you push your image must be set as public, or you must configure an image pull secret

#### Additional resources

- Installing the Operator SDK CLI
- Getting started with the OpenShift CLI

### 5.6.1.2. Creating and deploying Java-based Operators

You can build and deploy a simple Java-based Operator for Memcached by using the Operator SDK.

### Procedure

- 1. Create a project.
  - a. Create your project directory:
    - \$ mkdir memcached-operator
  - b. Change into the project directory:
    - \$ cd memcached-operator
  - c. Run the **operator-sdk init** command with the **quarkus** plugin to initialize the project:

\$ operator-sdk init \

- --plugins=quarkus \
- --domain=example.com \
- --project-name=memcached-operator

### 2. Create an API.

Create a simple Memcached API:

\$ operator-sdk create api \

- --plugins quarkus \
- --group cache \
- --version v1 \
- --kind Memcached

### 3. Build and push the Operator image.

Use the default **Makefile** targets to build and push your Operator. Set **IMG** with a pull spec for your image that uses a registry you can push to:

\$ make docker-build docker-push IMG=<registry>/<user>/<image\_name>:<tag>

# 4. Run the Operator.

- a. Install the CRD:
  - \$ make install
- b. Deploy the project to the cluster. Set **IMG** to the image that you pushed:
  - \$ make deploy IMG=<registry>/<user>/<image\_name>:<tag>

### 5. Create a sample custom resource (CR).

a. Create a sample CR:

\$ oc apply -f config/samples/cache\_v1\_memcached.yaml \ -n memcached-operator-system

b. Watch for the CR to reconcile the Operator:

\$ oc logs deployment.apps/memcached-operator-controller-manager \
 -c manager \

-n memcached-operator-system

#### 6. Delete a CR.

Delete a CR by running the following command:

\$ oc delete -f config/samples/cache\_v1\_memcached -n memcached-operator-system

#### 7. Clean up.

Run the following command to clean up the resources that have been created as part of this procedure:

\$ make undeploy

# 5.6.1.3. Next steps

• See Operator SDK tutorial for Java-based Operators for a more in-depth walkthrough on building a Java-based Operator.

# 5.6.2. Operator SDK tutorial for Java-based Operators



#### **IMPORTANT**

Java-based Operator SDK 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 developers can take advantage of Java programming language support in the Operator SDK to build an example Java-based Operator for Memcached, a distributed key-value store, and manage its lifecycle.

This process is accomplished using two centerpieces of the Operator Framework:

#### **Operator SDK**

The operator-sdk CLI tool and java-operator-sdk library API

### Operator Lifecycle Manager (OLM)

Installation, upgrade, and role-based access control (RBAC) of Operators on a cluster



# **NOTE**

This tutorial goes into greater detail than Getting started with Operator SDK for Javabased Operators.

# 5.6.2.1. Prerequisites

- Operator SDK CLI installed
- OpenShift CLI (oc) v4.13+ installed
- Java v11+
- Maven v3.6.3+
- Logged into an OpenShift Container Platform 4.13 cluster with **oc** with an account that has **cluster-admin** permissions
- To allow the cluster to pull the image, the repository where you push your image must be set as public, or you must configure an image pull secret

#### Additional resources

- Installing the Operator SDK CLI
- Getting started with the OpenShift CLI

# 5.6.2.2. Creating a project

Use the Operator SDK CLI to create a project called **memcached-operator**.

#### **Procedure**

- 1. Create a directory for the project:
  - \$ mkdir -p \$HOME/projects/memcached-operator
- 2. Change to the directory:
  - \$ cd \$HOME/projects/memcached-operator
- 3. Run the **operator-sdk init** command with the **quarkus** plugin to initialize the project:
  - \$ operator-sdk init \
    - --plugins=quarkus \
    - --domain=example.com \
    - --project-name=memcached-operator

#### 5.6.2.2.1. PROJECT file

Among the files generated by the **operator-sdk init** command is a Kubebuilder **PROJECT** file. Subsequent **operator-sdk** commands, as well as **help** output, that are run from the project root read this file and are aware that the project type is Java. For example:

domain: example.com

layout:

- quarkus.javaoperatorsdk.io/v1-alpha projectName: memcached-operator

version: "3"

# 5.6.2.3. Creating an API and controller

Use the Operator SDK CLI to create a custom resource definition (CRD) API and controller.

#### **Procedure**

1. Run the following command to create an API:

\$ operator-sdk create api \

- --plugins=quarkus \ 1
- --group=cache \ 2
- --version=v1\3
- --kind=Memcached 4

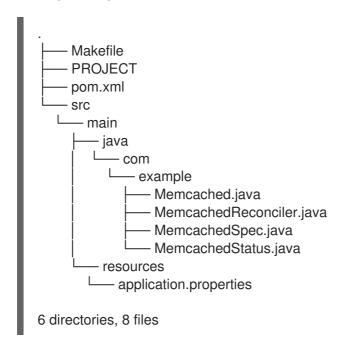
- Set the plugin flag to **quarkus**.
- Set the group flag to **cache**.
- 3 Set the version flag to **v1**.
- A Set the kind flag to **Memcached**.

# Verification

1. Run the **tree** command to view the file structure:

```
$ tree
```

# Example output



# 5.6.2.3.1. Defining the API

Define the API for the **Memcached** custom resource (CR).

# Procedure

- Edit the following files that were generated as part of the **create api** process:
  - a. Update the following attributes in the **MemcachedSpec.java** file to define the desired state of the **Memcached** CR:

```
public class MemcachedSpec {
    private Integer size;
    public Integer getSize() {
        return size;
    }
    public void setSize(Integer size) {
```

```
this.size = size;
}
}
```

b. Update the following attributes in the **MemcachedStatus.java** file to define the observed state of the **Memcached** CR:



#### **NOTE**

The example below illustrates a Node status field. It is recommended that you use typical status properties in practice.

```
import java.util.ArrayList;
import java.util.List;

public class MemcachedStatus {

    // Add Status information here
    // Nodes are the names of the memcached pods
    private List<String> nodes;

public List<String> getNodes() {
    if (nodes == null) {
        nodes = new ArrayList<>();
    }
    return nodes;
}

public void setNodes(List<String> nodes) {
    this.nodes = nodes;
}
```

c. Update the **Memcached.java** file to define the Schema for Memcached APIs that extends to both **MemcachedSpec.java** and **MemcachedStatus.java** files.

```
@Version("v1")
@Group("cache.example.com")
public class Memcached extends CustomResource<MemcachedSpec,
MemcachedStatus> implements Namespaced {}
```

# 5.6.2.3.2. Generating CRD manifests

After the API is defined with **MemcachedSpec** and **MemcachedStatus** files, you can generate CRD manifests.

### Procedure

• Run the following command from the **memcached-operator** directory to generate the CRD:

\$ mvn clean install

# Verification

 Verify the contents of the CRD in the target/kubernetes/memcacheds.cache.example.comv1.yml file as shown in the following example:

\$ cat target/kubernetes/memcacheds.cache.example.com-v1.yaml

# Example output

```
# Generated by Fabric8 CRDGenerator, manual edits might get overwritten!
apiVersion: apiextensions.k8s.io/v1
kind: CustomResourceDefinition
metadata:
 name: memcacheds.cache.example.com
 group: cache.example.com
 names:
  kind: Memcached
  plural: memcacheds
  singular: memcached
 scope: Namespaced
 versions:
 - name: v1
  schema:
   openAPIV3Schema:
    properties:
      spec:
       properties:
        size:
         type: integer
       type: object
      status:
       properties:
        nodes:
         items:
          type: string
         type: array
       type: object
    type: object
  served: true
  storage: true
  subresources:
   status: {}
```

# 5.6.2.3.3. Creating a Custom Resource

After generating the CRD manifests, you can create the Custom Resource (CR).

### Procedure

• Create a Memcached CR called memcached-sample.yaml:

```
apiVersion: cache.example.com/v1
kind: Memcached
metadata:
name: memcached-sample
```

```
spec:
  # Add spec fields here
  size: 1
```

# 5.6.2.4. Implementing the controller

After creating a new API and controller, you can implement the controller logic.

#### **Procedure**

1. Append the following dependency to the **pom.xml** file:

```
<dependency>
    <groupId>commons-collections</groupId>
    <artifactId>commons-collections</artifactId>
    <version>3.2.2</version>
</dependency>
```

2. For this example, replace the generated controller file **MemcachedReconciler.java** with following example implementation:

# Example 5.9. Example MemcachedReconciler.java

```
package com.example;
import io.fabric8.kubernetes.client.KubernetesClient;
import io.javaoperatorsdk.operator.api.reconciler.Context;
import io.javaoperatorsdk.operator.api.reconciler.Reconciler;
import io.javaoperatorsdk.operator.api.reconciler.UpdateControl;
import io.fabric8.kubernetes.api.model.ContainerBuilder;
import io.fabric8.kubernetes.api.model.ContainerPortBuilder;
import io.fabric8.kubernetes.api.model.LabelSelectorBuilder;
import io.fabric8.kubernetes.api.model.ObjectMetaBuilder;
import io.fabric8.kubernetes.api.model.OwnerReferenceBuilder;
import io.fabric8.kubernetes.api.model.Pod;
import io.fabric8.kubernetes.api.model.PodSpecBuilder;
import io.fabric8.kubernetes.api.model.PodTemplateSpecBuilder;
import io.fabric8.kubernetes.api.model.apps.Deployment;
import io.fabric8.kubernetes.api.model.apps.DeploymentBuilder;
import io.fabric8.kubernetes.api.model.apps.DeploymentSpecBuilder;
import org.apache.commons.collections.CollectionUtils;
import java.util.HashMap;
import java.util.List;
import java.util.Map;
import java.util.stream.Collectors;
public class MemcachedReconciler implements Reconciler<Memcached> {
 private final KubernetesClient client;
 public MemcachedReconciler(KubernetesClient client) {
  this.client = client;
 }
 // TODO Fill in the rest of the reconciler
```

```
@Override
 public UpdateControl<Memcached> reconcile(
   Memcached resource, Context context) {
   // TODO: fill in logic
   Deployment deployment = client.apps()
        .deployments()
        .inNamespace(resource.getMetadata().getNamespace())
        .withName(resource.getMetadata().getName())
        .get();
   if (deployment == null) {
      Deployment newDeployment = createMemcachedDeployment(resource);
      client.apps().deployments().create(newDeployment);
      return UpdateControl.noUpdate();
   }
   int currentReplicas = deployment.getSpec().getReplicas();
   int requiredReplicas = resource.getSpec().getSize();
   if (currentReplicas != requiredReplicas) {
      deployment.getSpec().setReplicas(requiredReplicas);
      client.apps().deployments().createOrReplace(deployment);
      return UpdateControl.noUpdate();
   }
   List<Pod> pods = client.pods()
      .inNamespace(resource.getMetadata().getNamespace())
      .withLabels(labelsForMemcached(resource))
      .list()
      .getItems();
   List<String> podNames =
      pods.stream().map(p -> p.getMetadata().getName()).collect(Collectors.toList());
   if (resource.getStatus() == null
         | !CollectionUtils.isEqualCollection(podNames,
resource.getStatus().getNodes())) {
      if (resource.getStatus() == null) resource.setStatus(new MemcachedStatus());
      resource.getStatus().setNodes(podNames);
      return UpdateControl.updateResource(resource);
   return UpdateControl.noUpdate();
 private Map<String, String> labelsForMemcached(Memcached m) {
  Map<String, String> labels = new HashMap<>();
  labels.put("app", "memcached");
  labels.put("memcached cr", m.getMetadata().getName());
  return labels;
 private Deployment createMemcachedDeployment(Memcached m) {
   Deployment deployment = new DeploymentBuilder()
      .withMetadata(
```

```
new ObjectMetaBuilder()
          .withName(m.getMetadata().getName())
          .withNamespace(m.getMetadata().getNamespace())
      .withSpec(
        new DeploymentSpecBuilder()
          .withReplicas(m.getSpec().getSize())
          .withSelector(
LabelSelectorBuilder().withMatchLabels(labelsForMemcached(m)).build())
          .withTemplate(
             new PodTemplateSpecBuilder()
               .withMetadata(
                  new ObjectMetaBuilder().withLabels(labelsForMemcached(m)).build())
               .withSpec(
                 new PodSpecBuilder()
                    .withContainers(
                      new ContainerBuilder()
                         .withImage("memcached:1.4.36-alpine")
                         .withName("memcached")
                         .withCommand("memcached", "-m=64", "-o", "modern", "-v")
                         .withPorts(
                           new ContainerPortBuilder()
                              .withContainerPort(11211)
                              .withName("memcached")
                              .build())
                         .build())
                    .build())
               .build())
          .build())
      .build();
  deployment.addOwnerReference(m);
  return deployment;
```

The example controller runs the following reconciliation logic for each **Memcached** custom resource (CR):

- Creates a Memcached deployment if it does not exist.
- Ensures that the deployment size matches the size specified by the **Memcached** CR spec.
- Updates the **Memcached** CR status with the names of the **memcached** pods.

The next subsections explain how the controller in the example implementation watches resources and how the reconcile loop is triggered. You can skip these subsections to go directly to Running the Operator.

#### 5.6.2.4.1. Reconcile loop

 Every controller has a reconciler object with a **Reconcile()** method that implements the reconcile loop. The reconcile loop is passed the **Deployment** argument, as shown in the following example:

```
Deployment deployment = client.apps()
    .deployments()
    .inNamespace(resource.getMetadata().getNamespace())
    .withName(resource.getMetadata().getName())
    .get();
```

2. As shown in the following example, if the **Deployment** is **null**, the deployment needs to be created. After you create the **Deployment**, you can determine if reconciliation is necessary. If there is no need of reconciliation, return the value of **UpdateControl.noUpdate()**, otherwise, return the value of `UpdateControl.updateStatus(resource):

```
if (deployment == null) {
    Deployment newDeployment = createMemcachedDeployment(resource);
    client.apps().deployments().create(newDeployment);
    return UpdateControl.noUpdate();
}
```

3. After getting the **Deployment**, get the current and required replicas, as shown in the following example:

```
int currentReplicas = deployment.getSpec().getReplicas();
int requiredReplicas = resource.getSpec().getSize();
```

4. If **currentReplicas** does not match the **requiredReplicas**, you must update the **Deployment**, as shown in the following example:

```
if (currentReplicas != requiredReplicas) {
    deployment.getSpec().setReplicas(requiredReplicas);
    client.apps().deployments().createOrReplace(deployment);
    return UpdateControl.noUpdate();
}
```

5. The following example shows how to obtain the list of pods and their names:

6. Check if resources were created and verify podnames with the Memcached resources. If a mismatch exists in either of these conditions, perform a reconciliation as shown in the following example:

```
if (resource.getStatus() == null
      || !CollectionUtils.isEqualCollection(podNames, resource.getStatus().getNodes())) {
    if (resource.getStatus() == null) resource.setStatus(new MemcachedStatus());
    resource.getStatus().setNodes(podNames);
    return UpdateControl.updateResource(resource);
}
```

### 5.6.2.4.2. Defining labelsForMemcached

labelsForMemcached is a utility to return a map of the labels to attach to the resources:

```
private Map<String, String> labelsForMemcached(Memcached m) {
    Map<String, String> labels = new HashMap<>();
    labels.put("app", "memcached");
    labels.put("memcached_cr", m.getMetadata().getName());
    return labels;
}
```

### 5.6.2.4.3. Define the createMemcachedDeployment

The createMemcachedDeployment method uses the fabric8 DeploymentBuilder class:

```
private Deployment createMemcachedDeployment(Memcached m) {
  Deployment deployment = new DeploymentBuilder()
    .withMetadata(
       new ObjectMetaBuilder()
         .withName(m.getMetadata().getName())
         .withNamespace(m.getMetadata().getNamespace())
         .build())
    .withSpec(
       new DeploymentSpecBuilder()
         .withReplicas(m.getSpec().getSize())
         .withSelector(
            new LabelSelectorBuilder().withMatchLabels(labelsForMemcached(m)).build())
         .withTemplate(
           new PodTemplateSpecBuilder()
              .withMetadata(
                new ObjectMetaBuilder().withLabels(labelsForMemcached(m)).build())
              .withSpec(
                new PodSpecBuilder()
                   .withContainers(
                     new ContainerBuilder()
                        .withImage("memcached:1.4.36-alpine")
                        .withName("memcached")
                        .withCommand("memcached", "-m=64", "-o", "modern", "-v")
                        .withPorts(
                          new ContainerPortBuilder()
                            .withContainerPort(11211)
                            .withName("memcached")
                            .build())
                       .build())
                   .build())
              .build())
         .build())
    .build();
 deployment.addOwnerReference(m);
 return deployment;
```

# 5.6.2.5. Running the Operator

There are three ways you can use the Operator SDK CLI to build and run your Operator:

- Run locally outside the cluster as a Go program.
- Run as a deployment on the cluster.
- Bundle your Operator and use Operator Lifecycle Manager (OLM) to deploy on the cluster.

### 5.6.2.5.1. Running locally outside the cluster

You can run your Operator project as a Go program outside of the cluster. This is useful for development purposes to speed up deployment and testing.

#### **Procedure**

1. Run the following command to compile the Operator:

\$ mvn clean install

# **Example output**

2. Run the following command to install the CRD to the default namespace:

\$ oc apply -f target/kubernetes/memcacheds.cache.example.com-v1.yml

# **Example output**

customresourcedefinition.apiextensions.k8s.io/memcacheds.cache.example.com created

3. Create a file called **rbac.yaml** as shown in the following example:

apiVersion: rbac.authorization.k8s.io/v1

kind: ClusterRoleBinding

metadata:

name: memcached-operator-admin

subjects:

- kind: ServiceAccount

name: memcached-quarkus-operator-operator

namespace: <operator\_namespace>

roleRef:

kind: ClusterRole name: cluster-admin

apiGroup: ""

4. Run the following command to grant **cluster-admin** privileges to the **memcached-quarkus-operator-operator** by applying the **rbac.yaml** file:

\$ oc apply -f rbac.yaml

5. Enter the following command to run the Operator:

\$ java -jar target/quarkus-app/quarkus-run.jar



#### NOTE

The **java** command will run the Operator and remain running until you end the process. You will need another terminal to complete the rest of these commands.

6. Apply the **memcached-sample.yaml** file with the following command:

\$ kubectl apply -f memcached-sample.yaml

# **Example output**

memcached.cache.example.com/memcached-sample created

#### Verification

• Run the following command to confirm that the pod has started:

\$ oc get all

# Example output

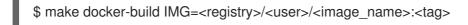
NAME READY STATUS RESTARTS AGE pod/memcached-sample-6c765df685-mfqnz 1/1 Running 0 18s

# 5.6.2.5.2. Running as a deployment on the cluster

You can run your Operator project as a deployment on your cluster.

#### **Procedure**

- 1. Run the following **make** commands to build and push the Operator image. Modify the **IMG** argument in the following steps to reference a repository that you have access to. You can obtain an account for storing containers at repository sites such as Quay.io.
  - a. Build the image:





#### **NOTE**

The Dockerfile generated by the SDK for the Operator explicitly references **GOARCH=amd64** for **go build**. This can be amended to **GOARCH=\$TARGETARCH** for non-AMD64 architectures. Docker will automatically set the environment variable to the value specified by – **platform**. With Buildah, the –**build-arg** will need to be used for the purpose. For more information, see Multiple Architectures.

b. Push the image to a repository:

\$ make docker-push IMG=<registry>/<user>/<image\_name>:<tag>



#### NOTE

The name and tag of the image, for example **IMG**= <registry>/<user>/<image\_name>:<tag>, in both the commands can also be set in your Makefile. Modify the **IMG** ?= controller:latest value to set your default image name.

2. Run the following command to install the CRD to the default namespace:

\$ oc apply -f target/kubernetes/memcacheds.cache.example.com-v1.yml

# **Example output**

customresourcedefinition.apiextensions.k8s.io/memcacheds.cache.example.com created

3. Create a file called **rbac.yaml** as shown in the following example:

apiVersion: rbac.authorization.k8s.io/v1

kind: ClusterRoleBinding

metadata:

name: memcached-operator-admin

subjects:

- kind: ServiceAccount

name: memcached-quarkus-operator-operator

namespace: <operator\_namespace>

roleRef:

kind: ClusterRole name: cluster-admin

apiGroup: ""



#### **IMPORTANT**

The **rbac.yaml** file will be applied at a later step.

4. Run the following command to deploy the Operator:

\$ make deploy IMG=<registry>/<user>/<image\_name>:<tag>

5. Run the following command to grant **cluster-admin** privileges to the **memcached-quarkus-operator-operator** by applying the **rbac.yaml** file created in a previous step:

\$ oc apply -f rbac.yaml

6. Run the following command to verify that the Operator is running:

\$ oc get all -n default

# **Example output**

NAME READY UP-TO-DATE AVAILABLE AGE pod/memcached-quarkus-operator-operator-7db86ccf58-k4mlm 0/1 Running 0 18s

7. Run the following command to apply the **memcached-sample.yaml** and create the **memcached-sample** pod:

\$ oc apply -f memcached-sample.yaml

### Example output

memcached.cache.example.com/memcached-sample created

### Verification

• Run the following command to confirm the pods have started:

\$ oc get all

# **Example output**

NAME READY STATUS RESTARTS AGE pod/memcached-quarkus-operator-operator-7b766f4896-kxnzt 1/1 Running 1 79s pod/memcached-sample-6c765df685-mfqnz 1/1 Running 0 18s

# 5.6.2.5.3. Bundling an Operator and deploying with Operator Lifecycle Manager

### 5.6.2.5.3.1. Bundling an Operator

The Operator bundle format is the default packaging method for Operator SDK and Operator Lifecycle Manager (OLM). You can get your Operator ready for use on OLM by using the Operator SDK to build and push your Operator project as a bundle image.

### **Prerequisites**

- Operator SDK CLI installed on a development workstation
- OpenShift CLI (oc) v4.13+ installed
- Operator project initialized by using the Operator SDK

# **Procedure**

- 1. Run the following **make** commands in your Operator project directory to build and push your Operator image. Modify the **IMG** argument in the following steps to reference a repository that you have access to. You can obtain an account for storing containers at repository sites such as Quay.io.
  - a. Build the image:

\$ make docker-build IMG=<registry>/<user>/<operator\_image\_name>:<tag>



#### NOTE

The Dockerfile generated by the SDK for the Operator explicitly references **GOARCH=amd64** for **go build**. This can be amended to **GOARCH=\$TARGETARCH** for non-AMD64 architectures. Docker will automatically set the environment variable to the value specified by – **platform**. With Buildah, the –**build-arg** will need to be used for the purpose. For more information, see Multiple Architectures.

- b. Push the image to a repository:
  - \$ make docker-push IMG=<registry>/<user>/<operator\_image\_name>:<tag>
- Create your Operator bundle manifest by running the make bundle command, which invokes several commands, including the Operator SDK generate bundle and bundle validate subcommands:
  - \$ make bundle IMG=<registry>/<user>/<operator\_image\_name>:<tag>

Bundle manifests for an Operator describe how to display, create, and manage an application. The **make bundle** command creates the following files and directories in your Operator project:

- A bundle manifests directory named bundle/manifests that contains a ClusterServiceVersion object
- A bundle metadata directory named bundle/metadata
- All custom resource definitions (CRDs) in a config/crd directory
- A Dockerfile bundle.Dockerfile

These files are then automatically validated by using **operator-sdk bundle validate** to ensure the on-disk bundle representation is correct.

- 3. Build and push your bundle image by running the following commands. OLM consumes Operator bundles using an index image, which reference one or more bundle images.
  - a. Build the bundle image. Set **BUNDLE\_IMG** with the details for the registry, user namespace, and image tag where you intend to push the image:
    - \$ make bundle-build BUNDLE\_IMG=<registry>/<user>/<bundle\_image\_name>:<tag>
  - b. Push the bundle image:
    - \$ docker push <registry>/<user>/<bundle\_image\_name>:<tag>

### 5.6.2.5.3.2. Deploying an Operator with Operator Lifecycle Manager

Operator Lifecycle Manager (OLM) helps you to install, update, and manage the lifecycle of Operators and their associated services on a Kubernetes cluster. OLM is installed by default on OpenShift Container Platform and runs as a Kubernetes extension so that you can use the web console and the

OpenShift CLI (oc) for all Operator lifecycle management functions without any additional tools.

The Operator bundle format is the default packaging method for Operator SDK and OLM. You can use the Operator SDK to quickly run a bundle image on OLM to ensure that it runs properly.

# **Prerequisites**

- Operator SDK CLI installed on a development workstation
- Operator bundle image built and pushed to a registry
- OLM installed on a Kubernetes-based cluster (v1.16.0 or later if you use apiextensions.k8s.io/v1 CRDs, for example OpenShift Container Platform 4.13)
- Logged in to the cluster with **oc** using an account with **cluster-admin** permissions

#### **Procedure**

• Enter the following command to run the Operator on the cluster:

```
$ operator-sdk run bundle \1
-n <namespace> \2
<registry>/<user>/<bundle_image_name>:<tag> 3
```

- The **run bundle** command creates a valid file-based catalog and installs the Operator bundle on your cluster using OLM.
- Optional: By default, the command installs the Operator in the currently active project in your ~/.kube/config file. You can add the -n flag to set a different namespace scope for the installation.
- If you do not specify an image, the command uses **quay.io/operator-framework/opm:latest** as the default index image. If you specify an image, the command uses the bundle image itself as the index image.



#### **IMPORTANT**

As of OpenShift Container Platform 4.11, the **run bundle** command supports the file-based catalog format for Operator catalogs by default. The deprecated SQLite database format for Operator catalogs continues to be supported; however, it will be removed in a future release. It is recommended that Operator authors migrate their workflows to the file-based catalog format.

This command performs the following actions:

- Create an index image referencing your bundle image. The index image is opaque and ephemeral, but accurately reflects how a bundle would be added to a catalog in production.
- Create a catalog source that points to your new index image, which enables OperatorHub to discover your Operator.
- Deploy your Operator to your cluster by creating an OperatorGroup, Subscription, InstallPlan, and all other required resources, including RBAC.

#### 5.6.2.6. Additional resources

- See Project layout for Java-based Operators to learn about the directory structures created by the Operator SDK.
- If a cluster-wide egress proxy is configured, cluster administrators can override the proxy settings or inject a custom CA certificate for specific Operators running on Operator Lifecycle Manager (OLM).

# 5.6.3. Project layout for Java-based Operators



### **IMPORTANT**

Java-based Operator SDK 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 **operator-sdk** CLI can generate, or *scaffold*, a number of packages and files for each Operator project.

# 5.6.3.1. Java-based project layout

Java-based Operator projects generated by the **operator-sdk init** command contain the following files and directories:

File or directory	Purpose
pom.xml	File that contains the dependencies required to run the Operator.
<domain>/</domain>	Directory that contains the files that represent the API. If the domain is <b>example.com</b> , this folder is called <b>example</b> /.
MemcachedRec onciler.java	Java file that defines controller implementations.
MemcachedSpe c.java	Java file that defines the desired state of the Memcached CR.
MemcachedStat us.java	Java file that defines the observed state of the Memcached CR.
Memcached.jav a	Java file that defines the Schema for Memcached APIs.
target/kubernet es/	Directory that contains the CRD yaml files.

# 5.6.4. Updating projects for newer Operator SDK versions

OpenShift Container Platform 4.13 supports Operator SDK 1.28.0. If you already have the 1.25.4 CLI installed on your workstation, you can update the CLI to 1.28.0 by installing the latest version.

However, to ensure your existing Operator projects maintain compatibility with Operator SDK 1.28.0, update steps are required for the associated breaking changes introduced since 1.25.4. You must perform the update steps manually in any of your Operator projects that were previously created or maintained with 1.25.4.

# 5.6.4.1. Updating Java-based Operator projects for Operator SDK 1.28.0

The following procedure updates an existing Java-based Operator project for compatibility with 1.28.0.

### **Prerequisites**

- Operator SDK 1.28.0 installed
- An Operator project created or maintained with Operator SDK 1.25.4

#### **Procedure**

- Find the ose-kube-rbac-proxy pull spec in the following files, and update the image tag to v4.13:
  - config/default/manager\_auth\_proxy\_patch.yaml
  - o bundle/manifests/memcached-operator.clusterserviceversion.yaml

```
...
containers:
- name: kube-rbac-proxy
image: registry.redhat.io/openshift4/ose-kube-rbac-proxy:v4.13 1
...
```

Update the tag version from **v4.12** to **v4.13**.

# 5.6.4.2. Additional resources

• Migrating package manifest projects to bundle format

# 5.7. DEFINING CLUSTER SERVICE VERSIONS (CSVS)

A cluster service version (CSV), defined by a **ClusterServiceVersion** object, is a YAML manifest created from Operator metadata that assists Operator Lifecycle Manager (OLM) in running the Operator in a cluster. It is the metadata that accompanies an Operator container image, used to populate user interfaces with information such as its logo, description, and version. It is also a source of technical information that is required to run the Operator, like the RBAC rules it requires and which custom resources (CRs) it manages or depends on.

The Operator SDK includes the CSV generator to generate a CSV for the current Operator project, customized using information contained in YAML manifests and Operator source files.

A CSV-generating command removes the responsibility of Operator authors having in-depth OLM

knowledge in order for their Operator to interact with OLM or publish metadata to the Catalog Registry. Further, because the CSV spec will likely change over time as new Kubernetes and OLM features are implemented, the Operator SDK is equipped to easily extend its update system to handle new CSV features going forward.

# 5.7.1. How CSV generation works

Operator bundle manifests, which include cluster service versions (CSVs), describe how to display, create, and manage an application with Operator Lifecycle Manager (OLM). The CSV generator in the Operator SDK, called by the **generate bundle** subcommand, is the first step towards publishing your Operator to a catalog and deploying it with OLM. The subcommand requires certain input manifests to construct a CSV manifest; all inputs are read when the command is invoked, along with a CSV base, to idempotently generate or regenerate a CSV.

Typically, the **generate kustomize manifests** subcommand would be run first to generate the input Kustomize bases that are consumed by the **generate bundle** subcommand. However, the Operator SDK provides the **make bundle** command, which automates several tasks, including running the following subcommands in order:

- 1. generate kustomize manifests
- 2. generate bundle
- 3. bundle validate

#### Additional resources

See Bundling an Operator for a full procedure that includes generating a bundle and CSV.

## 5.7.1.1. Generated files and resources

The **make bundle** command creates the following files and directories in your Operator project:

- A bundle manifests directory named bundle/manifests that contains a ClusterServiceVersion (CSV) object
- A bundle metadata directory named **bundle/metadata**
- All custom resource definitions (CRDs) in a **config/crd** directory
- A Dockerfile bundle.Dockerfile

The following resources are typically included in a CSV:

# Role

Defines Operator permissions within a namespace.

## ClusterRole

Defines cluster-wide Operator permissions.

#### Deployment

Defines how an Operand of an Operator is run in pods.

## CustomResourceDefinition (CRD)

Defines custom resources that your Operator reconciles.

#### **Custom resource examples**

Examples of resources adhering to the spec of a particular CRD.

# 5.7.1.2. Version management

The **--version** flag for the **generate bundle** subcommand supplies a semantic version for your bundle when creating one for the first time and when upgrading an existing one.

By setting the **VERSION** variable in your **Makefile**, the **--version** flag is automatically invoked using that value when the **generate bundle** subcommand is run by the **make bundle** command. The CSV version is the same as the Operator version, and a new CSV is generated when upgrading Operator versions.

# 5.7.2. Manually-defined CSV fields

Many CSV fields cannot be populated using generated, generic manifests that are not specific to Operator SDK. These fields are mostly human-written metadata about the Operator and various custom resource definitions (CRDs).

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

The following tables detail which manually-defined CSV fields are required and which are optional.

Table 5.7. Required CSV fields

Field	Description
metadata.name	A unique name for this CSV. Operator version should be included in the name to ensure uniqueness, for example <b>app-operator.v0.1.1</b> .
metadata.capab ilities	The capability level according to the Operator maturity model. Options include <b>Basic</b> Install, Seamless Upgrades, Full Lifecycle, Deep Insights, and Auto Pilot.
spec.displayNa me	A public name to identify the Operator.
spec.descriptio	A short description of the functionality of the Operator.
spec.keywords	Keywords describing the Operator.
spec.maintainer	Human or organizational entities maintaining the Operator, with a <b>name</b> and <b>email</b> .
spec.provider	The provider of the Operator (usually an organization), with a <b>name</b> .
spec.labels	Key-value pairs to be used by Operator internals.
spec.version	Semantic version of the Operator, for example <b>0.1.1</b> .

Description
Any CRDs the Operator uses. This field is populated automatically by the Operator SDK if any CRD YAML files are present in <b>deploy</b> /. However, several fields not in the CRD manifest spec require user input:
• <b>description</b> : description of the CRD.
<ul> <li>resources: any Kubernetes resources leveraged by the CRD, for example Pod and StatefulSet objects.</li> </ul>
• <b>specDescriptors</b> : UI hints for inputs and outputs of the Operator.

Table 5.8. Optional CSV fields

Field	Description
spec.replaces	The name of the CSV being replaced by this CSV.
spec.links	URLs (for example, websites and documentation) pertaining to the Operator or application being managed, each with a <b>name</b> and <b>url</b> .
spec.selector	Selectors by which the Operator can pair resources in a cluster.
spec.icon	A base64-encoded icon unique to the Operator, set in a <b>base64data</b> field with a <b>mediatype</b> .
spec.maturity	The level of maturity the software has achieved at this version. Options include planning, pre-alpha, alpha, beta, stable, mature, inactive, and deprecated.

Further details on what data each field above should hold are found in the CSV spec.



# NOTE

Several YAML fields currently requiring user intervention can potentially be parsed from Operator code.

# Additional resources

Operator maturity model

# 5.7.2.1. Operator metadata annotations

Operator developers can manually define certain annotations in the metadata of a cluster service version (CSV) to enable features or highlight capabilities in user interfaces (UIs), such as OperatorHub.

The following table lists Operator metadata annotations that can be manually defined using **metadata.annotations** fields.

Table 5.9. Annotations

Field	Description		
alm-examples	Provide custom resource definition (CRD) templates with a minimum set of configuration. Compatible Uls pre-fill this template for users to further customize.		
operatorframework.io/initialization-resource	Specify a single required custom resource by adding operatorframework.io/initialization-resource annotation to the cluster service version (CSV) during Operator installation. The user is then prompted to create the custom resource through a template provided in the CSV. Must include a template that contains a complete YAML definition.		
operatorframework.io/suggested-namespace	Set a suggested namespace where the Operator should be deployed.		
operatorframework.io/suggested- namespace-template	Set a manifest for a Namespace object with the namespace default node selector specified.		
operators.openshift.io/infrastructure-features	Infrastructure features supported by the Operator. Users can view and filter by these features when discovering Operators through OperatorHub in the web console. Valid, case-sensitive values:  • disconnected: Operator supports being mirrored into disconnected catalogs, including all dependencies, and does not require internet access. All related images required for mirroring are listed by the Operator.  • cnf: Operator provides a Cloud-native Network Functions (CNF) Kubernetes plugin.  • cni: Operator provides a Container Network Interface (CNI) Kubernetes plugin.  • csi: Operator provides a Container Storage Interface (CSI) Kubernetes plugin.  • proxy-aware: Operator supports running on a cluster behind a proxy. Operator accepts the standard proxy environment variables HTTP_PROXY and HTTPS_PROXY, which Operator Lifecycle Manager (OLM) provides to the Operator automatically when the cluster is configured to use a proxy. Required environment variables are passed down to Operands for managed workloads.		

Field	Description
operators.openshift.io/valid-subscription	Free-form array for listing any specific subscriptions that are required to use the Operator. For example, '["3Scale Commercial License", "Red Hat Managed Integration"]'.
operators.operatorframework.io/internal- objects	Hides CRDs in the UI that are not meant for user manipulation.

# Example use cases

# Operator supports disconnected and proxy-aware

operators.openshift.io/infrastructure-features: '["disconnected", "proxy-aware"]'

# Operator requires an OpenShift Container Platform license

operators.openshift.io/valid-subscription: '["OpenShift Container Platform"]'

# Operator requires a 3scale license

operators.openshift.io/valid-subscription: '["3Scale Commercial License", "Red Hat Managed Integration"]'

# Operator supports disconnected and proxy-aware, and requires an OpenShift Container Platform license

operators.openshift.io/infrastructure-features: '["disconnected", "proxy-aware"]' operators.openshift.io/valid-subscription: '["OpenShift Container Platform"]'

#### Additional resources

- CRD templates
- Initializing required custom resources
- Setting a suggested namespace
- Setting a suggested namespace with default node selector
- Enabling your Operator for restricted network environments (disconnected mode)
- Hiding internal objects

# 5.7.3. Enabling your Operator for restricted network environments

As an Operator author, your Operator must meet additional requirements to run properly in a restricted network, or disconnected, environment.

## Operator requirements for supporting disconnected mode

- Replace hard-coded image references with environment variables.
- In the cluster service version (CSV) of your Operator:
  - List any *related images*, or other container images that your Operator might require to perform their functions.
  - Reference all specified images by a digest (SHA) and not by a tag.
- All dependencies of your Operator must also support running in a disconnected mode.
- Your Operator must not require any off-cluster resources.

## **Prerequisites**

• An Operator project with a CSV. The following procedure uses the Memcached Operator as an example for Go-, Ansible-, and Helm-based projects.

#### **Procedure**

1. Set an environment variable for the additional image references used by the Operator in the **config/manager/manager.yaml** file:

# Example 5.10. Example config/manager/manager.yaml file

```
...
spec:
...
spec:
...
containers:
- command:
- /manager
...
env:
- name: <related_image_environment_variable> 1
value: "<related_image_reference_with_tag>" 2
```

- Define the environment variable, such as **RELATED\_IMAGE\_MEMCACHED**.
- 2 Set the related image reference and tag, such as **docker.io/memcached:1.4.36-alpine**.
- 2. Replace hard-coded image references with environment variables in the relevant file for your Operator project type:
  - For Go-based Operator projects, add the environment variable to the controllers/memcached\_controller.go file as shown in the following example:

Example 5.11. Example controllers/memcached\_controller.go file

// deploymentForMemcached returns a memcached Deployment object

• •

Spec: corev1.PodSpec{ Containers: []corev1.Container{{ Image: "memcached:1.4.36-alpine", + Image: os.Getenv("<related\_image\_environment\_variable>"), Name: "memcached", Command: []string{"memcached", "-m=64", "-o", "modern", "-v"}, Ports: []corev1.ContainerPort{{ Delete the image reference and tag.

- Use the os.Getenv function to call the <related image environment variable>.



#### NOTE

The **os.Getenv** function returns an empty string if a variable is not set. Set the <related\_image\_environment\_variable> before changing the file.

For Ansible-based Operator projects, add the environment variable to the roles/memcached/tasks/main.yml file as shown in the following example:

Example 5.12. Example roles/memcached/tasks/main.yml file

spec: containers: - name: memcached command: - memcached -m=64- -0 - modern image: "docker.io/memcached:1.4.36-alpine" + image: "{{ lookup('env', '<related\_image\_environment\_variable>') }}" ports: - containerPort: 11211

- Delete the image reference and tag.
- Use the **lookup** function to call the **<related\_image\_environment\_variable>**.
- For Helm-based Operator projects, add the **overrideValues** field to the **watches.yaml** file as shown in the following example:

# Example 5.13. Example watches.yaml file

- group: demo.example.com

version: v1alpha1 kind: Memcached

chart: helm-charts/memcached

overrideValues: 1

relatedImage: \${<related\_image\_environment\_variable>}

- Add the **overrideValues** field.
- Define the **overrideValues** field by using the **<related\_image\_environment\_variable>**, such as **RELATED\_IMAGE\_MEMCACHED**.
- a. Add the value of the overrideValues field to the helmcharts/memchached/values.yaml file as shown in the following example:

Example helm-charts/memchached/values.yaml file

```
...
relatedImage: ""
```

b. Edit the chart template in the **helm-charts/memcached/templates/deployment.yaml** file as shown in the following example:

# Example 5.14. Example helm-charts/memcached/templates/deployment.yaml file

```
containers:
```

- name: {{ .Chart.Name }}
  - security Context:
    - to Yaml  $\{\{ \ . \ Values.security Context \ | \ nindent \ 12 \ \}\}$

image: "{{ .Values.image.pullPolicy }}

env: 1

- name: related\_image 2

value: "{{ .Values.relatedImage }}" 3

- Add the **env** field.
- Name the environment variable.
- 3 Define the value of the environment variable.
- 3. Add the **BUNDLE\_GEN\_FLAGS** variable definition to your **Makefile** with the following changes:

## **Example Makefile**

```
BUNDLE_GEN_FLAGS ?= -q --overwrite --version $(VERSION)
$(BUNDLE_METADATA_OPTS)

# USE_IMAGE_DIGESTS defines if images are resolved via tags or digests
# You can enable this value if you would like to use SHA Based Digests
# To enable set flag to true
```

- Delete this line in the **Makefile**.
- Replace the line above with this line.
- 4. To update your Operator image to use a digest (SHA) and not a tag, run the **make bundle** command and set **USE\_IMAGE\_DIGESTS** to **true**:

```
$ make bundle USE_IMAGE_DIGESTS=true
```

5. Add the **disconnected** annotation, which indicates that the Operator works in a disconnected environment:

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

Operators can be filtered in OperatorHub by this infrastructure feature.

# 5.7.4. Enabling your Operator for multiple architectures and operating systems

Operator Lifecycle Manager (OLM) assumes that all Operators run on Linux hosts. However, as an Operator author, you can specify whether your Operator supports managing workloads on other architectures, if worker nodes are available in the OpenShift Container Platform cluster.

If your Operator supports variants other than AMD64 and Linux, you can add labels to the cluster service version (CSV) that provides the Operator to list the supported variants. Labels indicating supported architectures and operating systems are defined by the following:

```
labels:
```

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

- Set **<arch>** to a supported string.
- Set **<os>** to a supported string.



#### **NOTE**

Only the labels on the channel head of the default channel are considered for filtering package manifests by label. This means, for example, that providing an additional architecture for an Operator in the non-default channel is possible, but that architecture is not available for filtering in the **PackageManifest** API.

If a CSV does not include an **os** label, it is treated as if it has the following Linux support label by default:

labels:

operatorframework.io/os.linux: supported

If a CSV does not include an **arch** label, it is treated as if it has the following AMD64 support label by default:

labels:

operatorframework.io/arch.amd64: supported

If an Operator supports multiple node architectures or operating systems, you can add multiple labels, as well.

# **Prerequisites**

- An Operator project with a CSV.
- To support listing multiple architectures and operating systems, your Operator image referenced in the CSV must be a manifest list image.
- For the Operator to work properly in restricted network, or disconnected, environments, the image referenced must also be specified using a digest (SHA) and not by a tag.

## Procedure

 Add a label in the **metadata.labels** of your CSV for each supported architecture and operating system that your Operator supports:

labels:

operatorframework.io/arch.s390x: supported operatorframework.io/os.zos: supported operatorframework.io/os.linux: supported 1 operatorframework.io/arch.amd64: supported 2

1)2 After you add a new architecture or operating system, you must also now include the default **os.linux** and **arch.amd64** variants explicitly.

## Additional resources

• See the Image Manifest V 2, Schema 2 specification for more information on manifest lists.

# 5.7.4.1. Architecture and operating system support for Operators

The following strings are supported in Operator Lifecycle Manager (OLM) on OpenShift Container Platform when labeling or filtering Operators that support multiple architectures and operating systems:

Table 5.10. Architectures supported on OpenShift Container Platform

Architecture	String
AMD64	amd64
IBM Power	ppc64le
IBM Z	s390x

Table 5.11. Operating systems supported on OpenShift Container Platform

Operating system	String
Linux	linux
z/OS	zos



#### NOTE

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

# 5.7.5. Setting a suggested namespace

Some Operators must be deployed in a specific namespace, or with ancillary resources in specific namespaces, to work properly. If resolved from a subscription, Operator Lifecycle Manager (OLM) defaults the namespaced resources of an Operator to the namespace of its subscription.

As an Operator author, you can instead express a desired target namespace as part of your cluster service version (CSV) to maintain control over the final namespaces of the resources installed for their Operators. When adding the Operator to a cluster using OperatorHub, this enables the web console to autopopulate the suggested namespace for the cluster administrator during the installation process.

#### Procedure

• In your CSV, set the **operatorframework.io/suggested-namespace** annotation to your suggested namespace:

metadata:

annotations:

operatorframework.io/suggested-namespace: <namespace> 1



Set your suggested namespace.

# 5.7.6. Setting a suggested namespace with default node selector

Some Operators expect to run only on control plane nodes, which can be done by setting a **nodeSelector** in the **Pod** spec by the Operator itself.

To avoid getting duplicated and potentially conflicting cluster-wide default **nodeSelector**, you can set a default node selector on the namespace where the Operator runs. The default node selector will take precedence over the cluster default so the cluster default will not be applied to the pods in the Operators namespace.

When adding the Operator to a cluster using OperatorHub, the web console auto-populates the suggested namespace for the cluster administrator during the installation process. The suggested namespace is created using the namespace manifest in YAML which is included in the cluster service version (CSV).

#### **Procedure**

 In your CSV, set the operatorframework.io/suggested-namespace-template with a manifest for a Namespace object. The following sample is a manifest for an example Namespace with the namespace default node selector specified:

```
metadata:
annotations:
operatorframework.io/suggested-namespace-template:

{
    "apiVersion": "v1",
    "kind": "Namespace",
    "metadata": {
        "name": "vertical-pod-autoscaler-suggested-template",
        "annotations": {
            "openshift.io/node-selector": ""
        }
    }
}
```

1 Set your suggested namespace.



#### NOTE

If both **suggested-namespace** and **suggested-namespace-template** annotations are present in the CSV, **suggested-namespace-template** should take precedence.

# 5.7.7. Enabling Operator conditions

Operator Lifecycle Manager (OLM) provides Operators with a channel to communicate complex states that influence OLM behavior while managing the Operator. By default, OLM creates an **OperatorCondition** custom resource definition (CRD) when it installs an Operator. Based on the conditions set in the **OperatorCondition** custom resource (CR), the behavior of OLM changes accordingly.

To support Operator conditions, an Operator must be able to read the **OperatorCondition** CR created by OLM and have the ability to complete the following tasks:

- Get the specific condition.
- Set the status of a specific condition.

This can be accomplished by using the **operator-lib** library. An Operator author can provide a **controller-runtime** client in their Operator for the library to access the **OperatorCondition** CR owned by the Operator in the cluster.

The library provides a generic **Conditions** interface, which has the following methods to **Get** and **Set** a **conditionType** in the **OperatorCondition** CR:

#### Get

To get the specific condition, the library uses the **client.Get** function from **controller-runtime**, which requires an **ObjectKey** of type **types.NamespacedName** present in **conditionAccessor**.

#### Set

To update the status of the specific condition, the library uses the **client.Update** function from **controller-runtime**. An error occurs if the **conditionType** is not present in the CRD.

The Operator is allowed to modify only the **status** subresource of the CR. Operators can either delete or update the **status.conditions** array to include the condition. For more details on the format and description of the fields present in the conditions, see the upstream Condition GoDocs.



#### NOTE

Operator SDK 1.28.0 supports operator-lib v0.11.0.

## **Prerequisites**

• An Operator project generated using the Operator SDK.

## **Procedure**

To enable Operator conditions in your Operator project:

 In the go.mod file of your Operator project, add operator-framework/operator-lib as a required library:

```
module github.com/example-inc/memcached-operator
go 1.15

require (
    k8s.io/apimachinery v0.19.2
    k8s.io/client-go v0.19.2
    sigs.k8s.io/controller-runtime v0.7.0
    operator-framework/operator-lib v0.3.0
```

- 2. Write your own constructor in your Operator logic that will result in the following outcomes:
  - Accepts a controller-runtime client.
  - Accepts a conditionType.
  - Returns a **Condition** interface to update or add conditions.

Because OLM currently supports the **Upgradeable** condition, you can create an interface that has methods to access the **Upgradeable** condition. For example:

```
import (
...
    apiv1 "github.com/operator-framework/api/pkg/operators/v1"
)

func NewUpgradeable(cl client.Client) (Condition, error) {
    return NewCondition(cl, "apiv1.OperatorUpgradeable")
}

cond, err := NewUpgradeable(cl);
```

In this example, the **NewUpgradeable** constructor is further used to create a variable **cond** of type **Condition**. The **cond** variable would in turn have **Get** and **Set** methods, which can be used for handling the OLM **Upgradeable** condition.

## Additional resources

Operator conditions

# 5.7.8. Defining webhooks

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

The cluster service version (CSV) resource of an Operator can include a **webhookdefinitions** section to define the following types of webhooks:

- Admission webhooks (validating and mutating)
- Conversion webhooks

#### **Procedure**

Add a webhookdefinitions section to the spec section of the CSV of your Operator and include any webhook definitions using a type of ValidatingAdmissionWebhook,
 MutatingAdmissionWebhook, or ConversionWebhook. The following example contains all three types of webhooks:

# CSV containing webhooks

```
apiVersion: operators.coreos.com/v1alpha1
kind: ClusterServiceVersion
metadata:
name: webhook-operator.v0.0.1
spec:
customresourcedefinitions:
owned:
- kind: WebhookTest
name: webhooktests.webhook.operators.coreos.io
version: v1
install:
spec:
deployments:
- name: webhook-operator-webhook
```

```
strategy: deployment
installModes:
- supported: false
 type: OwnNamespace
- supported: false
 type: SingleNamespace
- supported: false
 type: MultiNamespace
- supported: true
 type: AllNamespaces
webhookdefinitions:
- type: ValidatingAdmissionWebhook 2
 admissionReviewVersions:
 - v1beta1
 - v1
 containerPort: 443
 targetPort: 4343
 deploymentName: webhook-operator-webhook
 failurePolicy: Fail
 generateName: vwebhooktest.kb.io
 rules:
 - apiGroups:
  - webhook.operators.coreos.io
  apiVersions:
  - v1
  operations:
  - CREATE
  - UPDATE
  resources:
  - webhooktests
 sideEffects: None
 webhookPath: /validate-webhook-operators-coreos-io-v1-webhooktest
- type: MutatingAdmissionWebhook 3
 admissionReviewVersions:
 - v1beta1
 - v1
 containerPort: 443
 targetPort: 4343
 deploymentName: webhook-operator-webhook
 failurePolicy: Fail
 generateName: mwebhooktest.kb.io
 rules:
 - apiGroups:
  - webhook.operators.coreos.io
  apiVersions:
  - v1
  operations:
  - CREATE
  - UPDATE
  resources:
  - webhooktests
 sideEffects: None
 webhookPath: /mutate-webhook-operators-coreos-io-v1-webhooktest
```

 type: ConversionWebhook 4 admissionReviewVersions:

- v1beta1

- v1

containerPort: 443 targetPort: 4343

deploymentName: webhook-operator-webhook

generateName: cwebhooktest.kb.io

sideEffects: None webhookPath: /convert conversionCRDs:

- webhooktests.webhook.operators.coreos.io 5

The CRDs targeted by the conversion webhook must exist here.

- A validating admission webhook.
- A mutating admission webhook.
- A conversion webhook.
- The **spec.PreserveUnknownFields** property of each CRD must be set to **false** or **nil**.

#### Additional resources

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

## 5.7.8.1. Webhook considerations for OLM

When deploying an Operator with webhooks using Operator Lifecycle Manager (OLM), you must define the following:

- The **type** field must be set to either **ValidatingAdmissionWebhook**, MutatingAdmissionWebhook, or ConversionWebhook, or the CSV will be placed in a failed phase.
- The CSV must contain a deployment whose name is equivalent to the value supplied in the deploymentName field of the webhookdefinition.

When the webhook is created, OLM ensures that the webhook only acts upon namespaces that match the Operator group that the Operator is deployed in.

## Certificate authority constraints

OLM is configured to provide each deployment with a single certificate authority (CA). The logic that generates and mounts the CA into the deployment was originally used by the API service lifecycle logic. As a result:

- The TLS certificate file is mounted to the deployment at /apiserver.local.config/certificates/apiserver.crt.
- The TLS key file is mounted to the deployment at /apiserver.local.config/certificates/apiserver.key.

#### Admission webhook rules constraints

To prevent an Operator from configuring the cluster into an unrecoverable state, OLM places the CSV in the failed phase if the rules defined in an admission webhook intercept any of the following requests:

- Requests that target all groups
- Requests that target the **operators.coreos.com** group
- Requests that target the ValidatingWebhookConfigurations or MutatingWebhookConfigurations resources

#### Conversion webhook constraints

OLM places the CSV in the failed phase if a conversion webhook definition does not adhere to the following constraints:

- CSVs featuring a conversion webhook can only support the **AllNamespaces** install mode.
- The CRD targeted by the conversion webhook must have its **spec.preserveUnknownFields** field set to **false** or **nil**.
- The conversion webhook defined in the CSV must target an owned CRD.
- There can only be one conversion webhook on the entire cluster for a given CRD.

# 5.7.9. Understanding your custom resource definitions (CRDs)

There are two types of custom resource definitions (CRDs) that your Operator can use: ones that are owned by it and ones that it depends on, which are required.

# 5.7.9.1. Owned CRDs

The custom resource definitions (CRDs) owned by your Operator are the most important part of your CSV. This establishes the link between your Operator and the required RBAC rules, dependency management, and other Kubernetes concepts.

It is common for your Operator to use multiple CRDs to link together concepts, such as top-level database configuration in one object and a representation of replica sets in another. Each one should be listed out in the CSV file.

Table 5.12. Owned CRD fields

Field	Description	Required/optional
Name	The full name of your CRD.	Required
Version	The version of that object API.	Required
Kind	The machine readable name of your CRD.	Required

Field	Description	Required/optional
DisplayName	A human readable version of your CRD name, for example <b>MongoDB Standalone</b> .	Required
Description	A short description of how this CRD is used by the Operator or a description of the functionality provided by the CRD.	Required
Group	The API group that this CRD belongs to, for example database.example.com.	Optional
Resources	Your CRDs own one or more types of Kubernetes objects. These are listed in the <b>resources</b> section to inform your users of the objects they might need to troubleshoot or how to connect to the application, such as the service or ingress rule that exposes a database.  It is recommended to only list out the objects that are important to a human, not an exhaustive list of everything you orchestrate. For example, do not list config maps that store internal state that are not meant to be modified by a user.	Optional

Field	Description	Required/optional
SpecDescriptors, StatusDescriptors , and ActionDescriptors	These descriptors are a way to hint UIs with certain inputs or outputs of your Operator that are most important to an end user. If your CRD contains the name of a secret or config map that the user must provide, you can specify that here. These items are linked and highlighted in compatible UIs.	Optional
	There are three types of descriptors:	
	<ul> <li>SpecDescriptors: A reference to fields in the spec block of an object.</li> </ul>	
	• <b>StatusDescriptors</b> : A reference to fields in the <b>status</b> block of an object.	
	<ul> <li>ActionDescriptors: A reference to actions that can be performed on an object.</li> </ul>	
	All descriptors accept the following fields:	
	<ul> <li>DisplayName: A human readable name for the Spec, Status, or Action.</li> </ul>	
	<ul> <li>Description: A short description of the Spec, Status, or Action and how it is used by the Operator.</li> </ul>	
	<ul> <li>Path: A dot-delimited path of the field on the object that this descriptor describes.</li> </ul>	
	<ul> <li>X-Descriptors: Used to determine which "capabilities" this descriptor has and which UI component to use. See the openshift/console project for a canonical list of React UI X- Descriptors for OpenShift Container Platform.</li> </ul>	
	Also see the <b>openshift/console</b> project for more	
	information on Descriptors in general.	

The following example depicts a **MongoDB Standalone** CRD that requires some user input in the form of a secret and config map, and orchestrates services, stateful sets, pods and config maps:

# **Example owned CRD**

- displayName: MongoDB Standalone

group: mongodb.com kind: MongoDbStandalone

name: mongodbstandalones.mongodb.com

resources:
- kind: Service
name: "

version: v1
- kind: StatefulSet name: "

version: v1beta2

- kind: Pod name: " version: v1
- kind: ConfigMap

name: "
version: v1
specDescriptors:

- description: Credentials for Ops Manager or Cloud Manager.

displayName: Credentials

path: credentials x-descriptors:

- 'urn:alm:descriptor:com.tectonic.ui:selector:core:v1:Secret'

- description: Project this deployment belongs to.

displayName: Project

path: project x-descriptors:

- 'urn:alm:descriptor:com.tectonic.ui:selector:core:v1:ConfigMap'

- description: MongoDB version to be installed.

displayName: Version

path: version x-descriptors:

- 'urn:alm:descriptor:com.tectonic.ui:label'

statusDescriptors:

- description: The status of each of the pods for the MongoDB cluster.

displayName: Pod Status

path: pods x-descriptors:

- 'urn:alm:descriptor:com.tectonic.ui:podStatuses'

version: v1 description: >-

MongoDB Deployment consisting of only one host. No replication of

data.

# 5.7.9.2. Required CRDs

Relying on other required CRDs is completely optional and only exists to reduce the scope of individual Operators and provide a way to compose multiple Operators together to solve an end-to-end use case.

An example of this is an Operator that might set up an application and install an etcd cluster (from an etcd Operator) to use for distributed locking and a Postgres database (from a Postgres Operator) for data storage.

Operator Lifecycle Manager (OLM) checks against the available CRDs and Operators in the cluster to fulfill these requirements. If suitable versions are found, the Operators are started within the desired namespace and a service account created for each Operator to create, watch, and modify the Kubernetes resources required.

Table 5.13. Required CRD fields

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

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

# **Example required CRD**

required:

- name: etcdclusters.etcd.database.coreos.com

version: v1beta2 kind: EtcdCluster

displayName: etcd Cluster

description: Represents a cluster of etcd nodes.

# 5.7.9.3. CRD upgrades

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

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

## 5.7.9.3.1. Adding a new CRD version

## **Procedure**

To add a new version of a CRD to your Operator:

Add a new entry in the CRD resource under the **versions** section of your CSV.
 For example, if the current CRD has a version **v1alpha1** and you want to add a new version **v1beta1** and mark it as the new storage version, add a new entry for **v1beta1**:

#### versions:

name: v1alpha1
served: true
storage: false
name: v1beta1
served: true
storage: true

New entry.

2. Ensure the referencing version of the CRD in the **owned** section of your CSV is updated if the CSV intends to use the new version:

customresourcedefinitions:

#### owned:

- name: cluster.example.com

version: v1beta1 1

kind: cluster

displayName: Cluster



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

# 5.7.9.3.2. Deprecating or removing a CRD version

Operator Lifecycle Manager (OLM) does not allow a serving version of a custom resource definition (CRD) to be removed right away. Instead, a deprecated version of the CRD must be first disabled by setting the **served** field in the CRD to **false**. Then, the non-serving version can be removed on the subsequent CRD upgrade.

#### **Procedure**

To deprecate and remove a specific version of a CRD:

1. Mark the deprecated version as non-serving to indicate this version is no longer in use and may be removed in a subsequent upgrade. For example:

#### versions:

- name: v1alpha1 served: false 1 storage: true



2. Switch the **storage** version to a serving version if the version to be deprecated is currently the **storage** version. For example:

#### versions:

name: v1alpha1 served: false storage: false 1
name: v1beta1 served: true storage: true 2

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



## NOTE

To remove a specific version that is or was the **storage** version from a CRD, that version must be removed from the **storedVersion** in the status of the CRD. OLM will attempt to do this for you if it detects a stored version no longer exists in the new CRD.

3. Upgrade the CRD with the above changes.

4. In subsequent upgrade cycles, the non-serving version can be removed completely from the CRD. For example:

versions:
- name: v1beta1
served: true

storage: true

5. Ensure the referencing CRD version in the **owned** section of your CSV is updated accordingly if that version is removed from the CRD.

# 5.7.9.4. CRD templates

Users of your Operator must be made aware of which options are required versus optional. You can provide templates for each of your custom resource definitions (CRDs) with a minimum set of configuration as an annotation named **alm-examples**. Compatible UIs will pre-fill this template for users to further customize.

The annotation consists of a list of the kind, for example, the CRD name and the corresponding **metadata** and **spec** of the Kubernetes object.

The following full example provides templates for **EtcdCluster**, **EtcdBackup** and **EtcdRestore**:

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

# 5.7.9.5. Hiding internal objects

It is common practice for Operators to use custom resource definitions (CRDs) internally to accomplish a task. These objects are not meant for users to manipulate and can be confusing to users of the Operator. For example, a database Operator might have a **Replication** CRD that is created whenever a user creates a Database object with **replication: true**.

As an Operator author, you can hide any CRDs in the user interface that are not meant for user manipulation by adding the **operators.operatorframework.io/internal-objects** annotation to the cluster service version (CSV) of your Operator.

#### **Procedure**

- 1. Before marking one of your CRDs as internal, ensure that any debugging information or configuration that might be required to manage the application is reflected on the status or **spec** block of your CR, if applicable to your Operator.
- 2. Add the **operators.operatorframework.io/internal-objects** annotation to the CSV of your Operator to specify any internal objects to hide in the user interface:

# Internal object annotation

```
apiVersion: operators.coreos.com/v1alpha1 kind: ClusterServiceVersion metadata: name: my-operator-v1.2.3 annotations: operators.operatorframework.io/internal-objects: '["my.internal.crd1.io","my.internal.crd2.io"]' 1 ...
```

1

Set any internal CRDs as an array of strings.

## 5.7.9.6. Initializing required custom resources

An Operator might require the user to instantiate a custom resource before the Operator can be fully functional. However, it can be challenging for a user to determine what is required or how to define the resource.

As an Operator developer, you can specify a single required custom resource by adding **operatorframework.io/initialization-resource** to the cluster service version (CSV) during Operator installation. You are then prompted to create the custom resource through a template that is provided in the CSV. The annotation must include a template that contains a complete YAML definition that is required to initialize the resource during installation.

If this annotation is defined, after installing the Operator from the OpenShift Container Platform web console, the user is prompted to create the resource using the template provided in the CSV.

#### Procedure

Add the operatorframework.io/initialization-resource annotation to the CSV of your Operator
to specify a required custom resource. For example, the following annotation requires the
creation of a StorageCluster resource and provides a full YAML definition:

### Initialization resource annotation

```
apiVersion: operators.coreos.com/v1alpha1
kind: ClusterServiceVersion
metadata:
 name: my-operator-v1.2.3
 annotations:
  operatorframework.io/initialization-resource: |-
       "apiVersion": "ocs.openshift.io/v1",
       "kind": "StorageCluster",
       "metadata": {
         "name": "example-storagecluster"
       "spec": {
         "manageNodes": false,
         "monPVCTemplate": {
            "spec": {
              "accessModes": [
                 "ReadWriteOnce"
```

```
"resources": {
       "requests": {
          "storage": "10Gi"
     "storageClassName": "gp2"
"storageDeviceSets": [
     "count": 3,
     "dataPVCTemplate": {
        "spec": {
          "accessModes": [
            "ReadWriteOnce"
          "resources": {
            "requests": {
               "storage": "1Ti"
          "storageClassName": "gp2",
          "volumeMode": "Block"
       }
     },
     "name": "example-deviceset",
     "placement": {},
     "portable": true,
     "resources": {}
]
```

# 5.7.10. Understanding your API services

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

## 5.7.10.1. Owned API services

When a CSV owns an API service, it is responsible for describing the deployment of the extension **api-server** that backs it and the group/version/kind (GVK) it provides.

An API service is uniquely identified by the group/version it provides and can be listed multiple times to denote the different kinds it is expected to provide.

Table 5.14. Owned API service fields

Field	Description	Required/optional
Group	Group that the API service provides, for example database.example.com.	Required

Field	Description	Required/optional
Version	Version of the API service, for example <b>v1alpha1</b> .	Required
Kind	A kind that the API service is expected to provide.	Required
Name	The plural name for the API service provided.	Required
DeploymentName	Name of the deployment defined by your CSV that corresponds to your API service (required for owned API services). During the CSV pending phase, the OLM Operator searches the <b>InstallStrategy</b> of your CSV for a <b>Deployment</b> spec with a matching name, and if not found, does not transition the CSV to the "Install Ready" phase.	Required
DisplayName	A human readable version of your API service name, for example <b>MongoDB Standalone</b> .	Required
Description	A short description of how this API service is used by the Operator or a description of the functionality provided by the API service.	Required
Resources	Your API services own one or more types of Kubernetes objects. These are listed in the resources section to inform your users of the objects they might need to troubleshoot or how to connect to the application, such as the service or ingress rule that exposes a database.  It is recommended to only list out the objects that are important to a human, not an exhaustive list of everything you orchestrate. For example, do not list config maps that store internal state that are not meant to be modified by a user.	Optional
SpecDescriptors, StatusDescriptors , and ActionDescriptors	Essentially the same as for owned CRDs.	Optional

# 5.7.10.1.1. API service resource creation

Operator Lifecycle Manager (OLM) is responsible for creating or replacing the service and API service resources for each unique owned API service:

- Service pod selectors are copied from the CSV deployment matching the **DeploymentName** field of the API service description.
- A new CA key/certificate pair is generated for each installation and the base64-encoded CA bundle is embedded in the respective API service resource.

# 5.7.10.1.2. API service serving certificates

OLM handles generating a serving key/certificate pair whenever an owned API service is being installed. The serving certificate has a common name (CN) containing the hostname of the generated **Service** resource and is signed by the private key of the CA bundle embedded in the corresponding API service resource.

The certificate is stored as a type **kubernetes.io/tls** secret in the deployment namespace, and a volume named **apiservice-cert** is automatically appended to the volumes section of the deployment in the CSV matching the **DeploymentName** field of the API service description.

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

# 5.7.10.2. Required API services

OLM ensures all required CSVs have an API service that is available and all expected GVKs are discoverable before attempting installation. This allows a CSV to rely on specific kinds provided by API services it does not own.

Table 5.15. Required API service fields

Field	Description	Required/optional
Group	Group that the API service provides, for example database.example.com.	Required
Version	Version of the API service, for example <b>v1alpha1</b> .	Required
Kind	A kind that the API service is expected to provide.	Required
DisplayName	A human readable version of your API service name, for example <b>MongoDB Standalone</b> .	Required
Description	A short description of how this API service is used by the Operator or a description of the functionality provided by the API service.	Required

## 5.8. WORKING WITH BUNDLE IMAGES

You can use the Operator SDK to package, deploy, and upgrade Operators in the bundle format for use on Operator Lifecycle Manager (OLM).

# 5.8.1. Bundling an Operator

The Operator bundle format is the default packaging method for Operator SDK and Operator Lifecycle Manager (OLM). You can get your Operator ready for use on OLM by using the Operator SDK to build and push your Operator project as a bundle image.

## **Prerequisites**

Operator SDK CLI installed on a development workstation

- OpenShift CLI (oc) v4.13+ installed
- Operator project initialized by using the Operator SDK
- If your Operator is Go-based, your project must be updated to use supported images for running on OpenShift Container Platform

#### Procedure

- 1. Run the following **make** commands in your Operator project directory to build and push your Operator image. Modify the **IMG** argument in the following steps to reference a repository that you have access to. You can obtain an account for storing containers at repository sites such as Quay.io.
  - a. Build the image:
    - \$ make docker-build IMG=<registry>/<user>/<operator\_image\_name>:<tag>



#### **NOTE**

The Dockerfile generated by the SDK for the Operator explicitly references **GOARCH=amd64** for **go build**. This can be amended to **GOARCH=\$TARGETARCH** for non-AMD64 architectures. Docker will automatically set the environment variable to the value specified by – **platform**. With Buildah, the –**build-arg** will need to be used for the purpose. For more information, see Multiple Architectures.

- b. Push the image to a repository:
  - \$ make docker-push IMG=<registry>/<user>/<operator\_image\_name>:<tag>
- Create your Operator bundle manifest by running the make bundle command, which invokes several commands, including the Operator SDK generate bundle and bundle validate subcommands:
  - \$ make bundle IMG=<registry>/<user>/<operator\_image\_name>:<tag>

Bundle manifests for an Operator describe how to display, create, and manage an application. The **make bundle** command creates the following files and directories in your Operator project:

- A bundle manifests directory named bundle/manifests that contains a ClusterServiceVersion object
- A bundle metadata directory named bundle/metadata
- All custom resource definitions (CRDs) in a **config/crd** directory
- A Dockerfile bundle.Dockerfile

These files are then automatically validated by using **operator-sdk bundle validate** to ensure the on-disk bundle representation is correct.

3. Build and push your bundle image by running the following commands. OLM consumes Operator bundles using an index image, which reference one or more bundle images.

- a. Build the bundle image. Set **BUNDLE\_IMG** with the details for the registry, user namespace, and image tag where you intend to push the image:
  - \$ make bundle-build BUNDLE\_IMG=<registry>/<user>/<bundle\_image\_name>:<tag>
- b. Push the bundle image:
  - \$ docker push <registry>/<user>/<bundle\_image\_name>:<tag>

# 5.8.2. Deploying an Operator with Operator Lifecycle Manager

Operator Lifecycle Manager (OLM) helps you to install, update, and manage the lifecycle of Operators and their associated services on a Kubernetes cluster. OLM is installed by default on OpenShift Container Platform and runs as a Kubernetes extension so that you can use the web console and the OpenShift CLI (oc) for all Operator lifecycle management functions without any additional tools.

The Operator bundle format is the default packaging method for Operator SDK and OLM. You can use the Operator SDK to quickly run a bundle image on OLM to ensure that it runs properly.

# **Prerequisites**

- Operator SDK CLI installed on a development workstation
- Operator bundle image built and pushed to a registry
- OLM installed on a Kubernetes-based cluster (v1.16.0 or later if you use apiextensions.k8s.io/v1 CRDs, for example OpenShift Container Platform 4.13)
- Logged in to the cluster with oc using an account with cluster-admin permissions
- If your Operator is Go-based, your project must be updated to use supported images for running on OpenShift Container Platform

#### Procedure

• Enter the following command to run the Operator on the cluster:

\$ operator-sdk run bundle \1
-n <namespace> \2
<registry>/<user>/<bundle\_image\_name>:<tag> 3

- The **run bundle** command creates a valid file-based catalog and installs the Operator bundle on your cluster using OLM.
- Optional: By default, the command installs the Operator in the currently active project in your ~/.kube/config file. You can add the -n flag to set a different namespace scope for the installation.
- If you do not specify an image, the command uses **quay.io/operator-framework/opm:latest** as the default index image. If you specify an image, the command uses the bundle image itself as the index image.



## **IMPORTANT**

As of OpenShift Container Platform 4.11, the **run bundle** command supports the file-based catalog format for Operator catalogs by default. The deprecated SQLite database format for Operator catalogs continues to be supported; however, it will be removed in a future release. It is recommended that Operator authors migrate their workflows to the file-based catalog format.

This command performs the following actions:

- Create an index image referencing your bundle image. The index image is opaque and ephemeral, but accurately reflects how a bundle would be added to a catalog in production.
- Create a catalog source that points to your new index image, which enables OperatorHub to discover your Operator.
- Deploy your Operator to your cluster by creating an OperatorGroup, Subscription, InstallPlan, and all other required resources, including RBAC.

#### Additional resources

- File-based catalogs in Operator Framework packaging format
- File-based catalogs in Managing custom catalogs
- Bundle format

# 5.8.3. Publishing a catalog containing a bundled Operator

To install and manage Operators, Operator Lifecycle Manager (OLM) requires that Operator bundles are listed in an index image, which is referenced by a catalog on the cluster. As an Operator author, you can use the Operator SDK to create an index containing the bundle for your Operator and all of its dependencies. This is useful for testing on remote clusters and publishing to container registries.



#### **NOTE**

The Operator SDK uses the **opm** CLI to facilitate index image creation. Experience with the **opm** command is not required. For advanced use cases, the **opm** command can be used directly instead of the Operator SDK.

#### **Prerequisites**

- Operator SDK CLI installed on a development workstation
- Operator bundle image built and pushed to a registry
- OLM installed on a Kubernetes-based cluster (v1.16.0 or later if you use **apiextensions.k8s.io/v1** CRDs, for example OpenShift Container Platform 4.13)
- Logged in to the cluster with **oc** using an account with **cluster-admin** permissions

## Procedure

1. Run the following **make** command in your Operator project directory to build an index image containing your Operator bundle:

\$ make catalog-build CATALOG\_IMG=<registry>/<user>/<index\_image\_name>:<tag>

where the **CATALOG\_IMG** argument references a repository that you have access to. You can obtain an account for storing containers at repository sites such as Quay.io.

2. Push the built index image to a repository:

\$ make catalog-push CATALOG\_IMG=<registry>/<user>/<index\_image\_name>:<tag>

#### **TIP**

You can use Operator SDK **make** commands together if you would rather perform multiple actions in sequence at once. For example, if you had not yet built a bundle image for your Operator project, you can build and push both a bundle image and an index image with the following syntax:

```
$ make bundle-build bundle-push catalog-build catalog-push \
BUNDLE_IMG=<bundle_image_pull_spec> \
CATALOG_IMG=<index_image_pull_spec>
```

Alternatively, you can set the **IMAGE\_TAG\_BASE** field in your **Makefile** to an existing repository:

```
IMAGE_TAG_BASE=quay.io/example/my-operator
```

You can then use the following syntax to build and push images with automatically-generated names, such as **quay.io/example/my-operator-bundle:v0.0.1** for the bundle image and **quay.io/example/my-operator-catalog:v0.0.1** for the index image:

\$ make bundle-build bundle-push catalog-build catalog-push

3. Define a **CatalogSource** object that references the index image you just generated, and then create the object by using the **oc apply** command or web console:

## Example CatalogSource YAML

```
apiVersion: operators.coreos.com/v1alpha1
kind: CatalogSource
metadata:
name: cs-memcached
namespace: <operator_namespace>
spec:
displayName: My Test
publisher: Company
sourceType: grpc
grpcPodConfig:
securityContextConfig: <security_mode>
image: quay.io/example/memcached-catalog:v0.0.1
updateStrategy:
registryPoll:
interval: 10m
```

- 1 Specify the value of **legacy** or **restricted**. If the field is not set, the default value is **legacy**. In a future OpenShift Container Platform release, it is planned that the default value will be
- Set **image** to the image pull spec you used previously with the **CATALOG\_IMG** argument.
- 4. Check the catalog source:

\$ oc get catalogsource

# Example output

NAME DISPLAY TYPE PUBLISHER AGE cs-memcached My Test grpc Company 4h31m

#### Verification

- 1. Install the Operator using your catalog:
  - a. Define an **OperatorGroup** object and create it by using the **oc apply** command or web console:

# **Example Operator Group YAML**

apiVersion: operators.coreos.com/v1
kind: OperatorGroup
metadata:
 name: my-test
 namespace: <operator\_namespace>
spec:
 targetNamespaces:
 - <operator\_namespace>

b. Define a **Subscription** object and create it by using the **oc apply** command or web console:

## **Example Subscription YAML**

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

metadata:

name: catalogtest

namespace: <catalog\_namespace>

spec:

channel: "alpha"

installPlanApproval: Manual

name: catalog

source: cs-memcached

sourceNamespace: <operator\_namespace> startingCSV: memcached-operator.v0.0.1

- 2. Verify the installed Operator is running:
  - a. Check the Operator group:

\$ oc get og

# **Example output**

NAME AGE my-test 4h40m

b. Check the cluster service version (CSV):

\$ oc get csv

# Example output

NAME DISPLAY VERSION REPLACES PHASE memcached-operator.v0.0.1 Test 0.0.1 Succeeded

c. Check the pods for the Operator:

\$ oc get pods

# Example output

 NAME
 READY
 STATUS
 RESTARTS
 AGE

 9098d908802769fbde8bd45255e69710a9f8420a8f3d814abe88b68f8ervdj6
 0/1

 Completed
 0
 4h33m

 catalog-controller-manager-7fd5b7b987-69s4n
 2/2
 Running
 0

 4h32m

 cs-memcached-7622r
 1/1
 Running
 0
 4h33m

#### Additional resources

 See Managing custom catalogs for details on direct usage of the opm CLI for more advanced use cases.

# 5.8.4. Testing an Operator upgrade on Operator Lifecycle Manager

You can quickly test upgrading your Operator by using Operator Lifecycle Manager (OLM) integration in the Operator SDK, without requiring you to manually manage index images and catalog sources.

The **run bundle-upgrade** subcommand automates triggering an installed Operator to upgrade to a later version by specifying a bundle image for the later version.

## **Prerequisites**

- Operator installed with OLM either by using the **run bundle** subcommand or with traditional OLM installation
- A bundle image that represents a later version of the installed Operator

#### **Procedure**

1. If your Operator has not already been installed with OLM, install the earlier version either by using the **run bundle** subcommand or with traditional OLM installation.



#### **NOTE**

If the earlier version of the bundle was installed traditionally using OLM, the newer bundle that you intend to upgrade to must not exist in the index image referenced by the catalog source. Otherwise, running the **run bundle-upgrade** subcommand will cause the registry pod to fail because the newer bundle is already referenced by the index that provides the package and cluster service version (CSV).

For example, you can use the following **run bundle** subcommand for a Memcached Operator by specifying the earlier bundle image:

\$ operator-sdk run bundle <registry>/<user>/memcached-operator:v0.0.1

# **Example output**

INFO[0006] Creating a File-Based Catalog of the bundle "quay.io/demo/memcached-operator:v0.0.1"

INFO[0008] Generated a valid File-Based Catalog

INFO[0012] Created registry pod: quay-io-demo-memcached-operator-v1-0-1

INFO[0012] Created CatalogSource: memcached-operator-catalog

INFO[0012] OperatorGroup "operator-sdk-og" created

INFO[0012] Created Subscription: memcached-operator-v0-0-1-sub

INFO[0015] Approved InstallPlan install-h9666 for the Subscription: memcached-operator-v0-0-1-sub

INFO[0015] Waiting for ClusterServiceVersion "my-project/memcached-operator.v0.0.1" to reach 'Succeeded' phase

INFO[0015] Waiting for ClusterServiceVersion ""my-project/memcached-operator.v0.0.1" to appear

INFO[0026] Found ClusterServiceVersion "my-project/memcached-operator.v0.0.1" phase: Pending

INFO[0028] Found ClusterServiceVersion "my-project/memcached-operator.v0.0.1" phase: Installing

INFO[0059] Found ClusterServiceVersion "my-project/memcached-operator.v0.0.1" phase: Succeeded

INFO[0059] OLM has successfully installed "memcached-operator.v0.0.1"

2. Upgrade the installed Operator by specifying the bundle image for the later Operator version:

\$ operator-sdk run bundle-upgrade <registry>/<user>/memcached-operator:v0.0.2

# Example output

INFO[0002] Found existing subscription with name memcached-operator-v0-0-1-sub and namespace my-project

INFO[0002] Found existing catalog source with name memcached-operator-catalog and namespace my-project

INFO[0008] Generated a valid Upgraded File-Based Catalog

INFO[0009] Created registry pod: quay-io-demo-memcached-operator-v0-0-2

INFO[0009] Updated catalog source memcached-operator-catalog with address and annotations

INFO[0010] Deleted previous registry pod with name "quay-io-demo-memcached-operator-v0-0-1"

INFO[0041] Approved InstallPlan install-gvcjh for the Subscription: memcached-operator-v0-0-1-sub

INFO[0042] Waiting for ClusterServiceVersion "my-project/memcached-operator.v0.0.2" to reach 'Succeeded' phase

INFO[0019] Found ClusterServiceVersion "my-project/memcached-operator.v0.0.2" phase: Pending

INFO[0042] Found ClusterServiceVersion "my-project/memcached-operator.v0.0.2" phase: InstallReady

INFO[0043] Found ClusterServiceVersion "my-project/memcached-operator.v0.0.2" phase: Installing

INFO[0044] Found ClusterServiceVersion "my-project/memcached-operator.v0.0.2" phase: Succeeded

INFO[0044] Successfully upgraded to "memcached-operator.v0.0.2"

3. Clean up the installed Operators:

\$ operator-sdk cleanup memcached-operator

#### Additional resources

Traditional Operator installation with OLM

# 5.8.5. Controlling Operator compatibility with OpenShift Container Platform versions



#### **IMPORTANT**

Kubernetes periodically deprecates certain APIs that are removed in subsequent releases. If your Operator is using a deprecated API, it might no longer work after the OpenShift Container Platform cluster is upgraded to the Kubernetes version where the API has been removed.

As an Operator author, it is strongly recommended that you review the Deprecated API Migration Guide in Kubernetes documentation and keep your Operator projects up to date to avoid using deprecated and removed APIs. Ideally, you should update your Operator before the release of a future version of OpenShift Container Platform that would make the Operator incompatible.

When an API is removed from an OpenShift Container Platform version, Operators running on that cluster version that are still using removed APIs will no longer work properly. As an Operator author, you should plan to update your Operator projects to accommodate API deprecation and removal to avoid interruptions for users of your Operator.

#### TIP

You can check the event alerts of your Operators to find whether there are any warnings about APIs currently in use. The following alerts fire when they detect an API in use that will be removed in the next release:

## **APIRemovedInNextReleaseInUse**

APIs that will be removed in the next OpenShift Container Platform release.

#### **APIRemovedInNextEUSReleaseInUse**

APIs that will be removed in the next OpenShift Container Platform Extended Update Support (EUS) release.

If a cluster administrator has installed your Operator, before they upgrade to the next version of OpenShift Container Platform, they must ensure a version of your Operator is installed that is compatible with that next cluster version. While it is recommended that you update your Operator projects to no longer use deprecated or removed APIs, if you still need to publish your Operator bundles with removed APIs for continued use on earlier versions of OpenShift Container Platform, ensure that the bundle is configured accordingly.

The following procedure helps prevent administrators from installing versions of your Operator on an incompatible version of OpenShift Container Platform. These steps also prevent administrators from upgrading to a newer version of OpenShift Container Platform that is incompatible with the version of your Operator that is currently installed on their cluster.

This procedure is also useful when you know that the current version of your Operator will not work well, for any reason, on a specific OpenShift Container Platform version. By defining the cluster versions where the Operator should be distributed, you ensure that the Operator does not appear in a catalog of a cluster version which is outside of the allowed range.



#### **IMPORTANT**

Operators that use deprecated APIs can adversely impact critical workloads when cluster administrators upgrade to a future version of OpenShift Container Platform where the API is no longer supported. If your Operator is using deprecated APIs, you should configure the following settings in your Operator project as soon as possible.

## **Prerequisites**

• An existing Operator project

## Procedure

If you know that a specific bundle of your Operator is not supported and will not work correctly
on OpenShift Container Platform later than a certain cluster version, configure the maximum
version of OpenShift Container Platform that your Operator is compatible with. In your
Operator project's cluster service version (CSV), set the olm.maxOpenShiftVersion
annotation to prevent administrators from upgrading their cluster before upgrading the installed
Operator to a compatible version:



#### **IMPORTANT**

You must use **olm.maxOpenShiftVersion** annotation only if your Operator bundle version cannot work in later versions. Be aware that cluster admins cannot upgrade their clusters with your solution installed. If you do not provide later version and a valid upgrade path, administrators may uninstall your Operator and can upgrade the cluster version.

## Example CSV with olm.maxOpenShiftVersion annotation

apiVersion: operators.coreos.com/v1alpha1

kind: ClusterServiceVersion

metadata: annotations:

"olm.properties": "{{"type": "olm.maxOpenShiftVersion", "value": "<cluster version>"}| 1



- Specify the maximum cluster version of OpenShift Container Platform that your Operator is compatible with. For example, setting value to 4.9 prevents cluster upgrades to OpenShift Container Platform versions later than 4.9 when this bundle is installed on a cluster.
- 2. If your bundle is intended for distribution in a Red Hat-provided Operator catalog, configure the compatible versions of OpenShift Container Platform for your Operator by setting the following properties. This configuration ensures your Operator is only included in catalogs that target compatible versions of OpenShift Container Platform:



## NOTE

This step is only valid when publishing Operators in Red Hat-provided catalogs. If your bundle is only intended for distribution in a custom catalog, you can skip this step. For more details, see "Red Hat-provided Operator catalogs".

a. Set the **com.redhat.openshift.versions** annotation in your project's bundle/metadata/annotations.yaml file:

### Example bundle/metadata/annotations.yaml file with compatible versions

com.redhat.openshift.versions: "v4.7-v4.9" 1



b. To prevent your bundle from being carried on to an incompatible version of OpenShift Container Platform, ensure that the index image is generated with the proper com.redhat.openshift.versions label in your Operator's bundle image. For example, if your project was generated using the Operator SDK, update the bundle.Dockerfile file:

## Example bundle.Dockerfile with compatible versions

LABEL com.redhat.openshift.versions="<versions>" 1



Set to a range or single version, for example, **v4.7-v4.9**. This setting defines the cluster versions where the Operator should be distributed, and the Operator does not appear in a catalog of a cluster version which is outside of the range.

You can now bundle a new version of your Operator and publish the updated version to a catalog for distribution.

#### Additional resources

- Managing OpenShift Versions in the Certified Operator Build Guide
- Updating installed Operators
- Red Hat-provided Operator catalogs

#### 5.8.6. Additional resources

- See Operator Framework packaging format for details on the bundle format.
- See Managing custom catalogs for details on adding bundle images to index images by using the **opm** command.
- See Operator Lifecycle Manager workflow for details on how upgrades work for installed Operators.

## 5.9. COMPLYING WITH POD SECURITY ADMISSION

Pod security admission is an implementation of the Kubernetes pod security standards. Pod security admission restricts the behavior of pods. Pods that do not comply with the pod security admission defined globally or at the namespace level are not admitted to the cluster and cannot run.

If your Operator project does not require escalated permissions to run, you can ensure your workloads run in namespaces set to the **restricted** pod security level. If your Operator project requires escalated permissions to run, you must set the following security context configurations:

- The allowed pod security admission level for the Operator's namespace
- The allowed security context constraints (SCC) for the workload's service account

For more information, see Understanding and managing pod security admission .

# 5.9.1. Security context constraint synchronization with pod security standards

OpenShift Container Platform includes Kubernetes pod security admission. Globally, the **privileged** profile is enforced, and the **restricted** profile is used for warnings and audits.

In addition to the global pod security admission control configuration, a controller exists that applies pod security admission control **warn** and **audit** labels to namespaces according to the SCC permissions of the service accounts that are in a given namespace.



#### **IMPORTANT**

Namespaces that are defined as part of the cluster payload have pod security admission synchronization disabled permanently. You can enable pod security admission synchronization on other namespaces as necessary. If an Operator is installed in a user-created **openshift-\*** namespace, synchronization is turned on by default after a cluster service version (CSV) is created in the namespace.

The controller examines **ServiceAccount** object permissions to use security context constraints in each namespace. Security context constraints (SCCs) are mapped to pod security profiles based on their field values; the controller uses these translated profiles. Pod security admission **warn** and **audit** labels are set to the most privileged pod security profile found in the namespace to prevent warnings and audit logging as pods are created.

Namespace labeling is based on consideration of namespace-local service account privileges.

Applying pods directly might use the SCC privileges of the user who runs the pod. However, user privileges are not considered during automatic labeling.

# 5.9.2. Ensuring Operator workloads run in namespaces set to the restricted pod security level

To ensure your Operator project can run on a wide variety of deployments and environments, configure the Operator's workloads to run in namespaces set to the **restricted** pod security level.



#### **WARNING**

You must leave the **runAsUser** field empty. If your image requires a specific user, it cannot be run under restricted security context constraints (SCC) and restricted pod security enforcement.

#### Procedure

 To configure Operator workloads to run in namespaces set to the **restricted** pod security level, edit your Operator's namespace definition similar to the following examples:



### **IMPORTANT**

It is recommended that you set the seccomp profile in your Operator's namespace definition. However, setting the seccomp profile is not supported in OpenShift Container Platform 4.10.

• For Operator projects that must run in only OpenShift Container Platform 4.11 and later, edit your Operator's namespace definition similar to the following example:

#### Example config/manager/manager.yaml file

spec:

```
securityContext:
seccompProfile:
type: RuntimeDefault
runAsNonRoot: true
containers:
- name: <operator_workload_container>
securityContext:
allowPrivilegeEscalation: false
capabilities:
drop:
- ALL
...
```

- By setting the seccomp profile type to **RuntimeDefault**, the SCC defaults to the pod security profile of the namespace.
- For Operator projects that must also run in OpenShift Container Platform 4.10, edit your Operator's namespace definition similar to the following example:

## Example config/manager/manager.yaml file

```
spec:
securityContext: 1
runAsNonRoot: true
containers:
- name: <operator_workload_container>
securityContext:
allowPrivilegeEscalation: false
capabilities:
drop:
- ALL
...
```

Leaving the seccomp profile type unset ensures your Operator project can run in OpenShift Container Platform 4.10.

#### Additional resources

Managing security context constraints

# 5.9.3. Managing pod security admission for Operator workloads that require escalated permissions

If your Operator project requires escalated permissions to run, you must edit your Operator's cluster service version (CSV).

#### **Procedure**

1. Set the security context configuration to the required permission level in your Operator's CSV, similar to the following example:

Example < operator name > .clusterserviceversion.vaml file with network administrator

# privileges

```
containers:
- name: my-container
securityContext:
allowPrivilegeEscalation: false
capabilities:
add:
- "NET_ADMIN"
```

2. Set the service account privileges that allow your Operator's workloads to use the required security context constraints (SCC), similar to the following example:

## Example < operator name > .clusterserviceversion.yaml file

```
install:
spec:
clusterPermissions:
- rules:
- apiGroups:
- security.openshift.io
resourceNames:
- privileged
resources:
- securitycontextconstraints
verbs:
- use
serviceAccountName: default
```

3. Edit your Operator's CSV description to explain why your Operator project requires escalated permissions similar to the following example:

## Example < operator\_name > .clusterserviceversion.yaml file

```
spec:
apiservicedefinitions:{}
...
description: The <operator_name> requires a privileged pod security admission label set on the Operator's namespace. The Operator's agents require escalated permissions to restart the node if the node needs remediation.
```

## 5.9.4. Additional resources

• Understanding and managing pod security admission

# 5.10. VALIDATING OPERATORS USING THE SCORECARD TOOL

As an Operator author, you can use the scorecard tool in the Operator SDK to do the following tasks:

- Validate that your Operator project is free of syntax errors and packaged correctly
- Review suggestions about ways you can improve your Operator

#### 5.10.1. About the scorecard tool

While the Operator SDK **bundle validate** subcommand can validate local bundle directories and remote bundle images for content and structure, you can use the **scorecard** command to run tests on your Operator based on a configuration file and test images. These tests are implemented within test images that are configured and constructed to be executed by the scorecard.

The scorecard assumes it is run with access to a configured Kubernetes cluster, such as OpenShift Container Platform. The scorecard runs each test within a pod, from which pod logs are aggregated and test results are sent to the console. The scorecard has built-in basic and Operator Lifecycle Manager (OLM) tests and also provides a means to execute custom test definitions.

#### Scorecard workflow

- 1. Create all resources required by any related custom resources (CRs) and the Operator
- 2. Create a proxy container in the deployment of the Operator to record calls to the API server and run tests
- 3. Examine parameters in the CRs

The scorecard tests make no assumptions as to the state of the Operator being tested. Creating Operators and CRs for an Operators are beyond the scope of the scorecard itself. Scorecard tests can, however, create whatever resources they require if the tests are designed for resource creation.

#### scorecard command syntax

\$ operator-sdk scorecard <bundle\_dir\_or\_image> [flags]

The scorecard requires a positional argument for either the on-disk path to your Operator bundle or the name of a bundle image.

For further information about the flags, run:

\$ operator-sdk scorecard -h

# 5.10.2. Scorecard configuration

The scorecard tool uses a configuration that allows you to configure internal plugins, as well as several global configuration options. Tests are driven by a configuration file named **config.yaml**, which is generated by the **make bundle** command, located in your **bundle**/ directory:

```
./bundle
...
tests
scorecard
config.yaml
```

## Example scorecard configuration file

kind: Configuration

apiversion: scorecard.operatorframework.io/v1alpha3

metadata: name: config stages:

parallel: true tests:

- image: quay.io/operator-framework/scorecard-test:v1.28.0

entrypoint:

scorecard-testbasic-check-spec

labels:

suite: basic

test: basic-check-spec-test

- image: quay.io/operator-framework/scorecard-test:v1.28.0

entrypoint:

- scorecard-test

- olm-bundle-validation

labels: suite: olm

test: olm-bundle-validation-test

The configuration file defines each test that scorecard can execute. The following fields of the scorecard configuration file define the test as follows:

Configuration field	Description
image	Test container image name that implements a test
entrypoint	Command and arguments that are invoked in the test image to execute a test
labels	Scorecard-defined or custom labels that select which tests to run

# 5.10.3. Built-in scorecard tests

The scorecard ships with pre-defined tests that are arranged into suites: the basic test suite and the Operator Lifecycle Manager (OLM) suite.

Table 5.16. Basic test suite

Test	Description	Short name
Spec Block Exists	This test checks the custom resource (CR) created in the cluster to make sure that all CRs have a <b>spec</b> block.	basic-check-spec- test

## Table 5.17. OLM test suite

Test	Description	Short name

Test	Description	Short name
Bundle Validation	This test validates the bundle manifests found in the bundle that is passed into scorecard. If the bundle contents contain errors, then the test result output includes the validator log as well as error messages from the validation library.	olm-bundle- validation-test
Provided APIs Have Validation	This test verifies that the custom resource definitions (CRDs) for the provided CRs contain a validation section and that there is validation for each <b>spec</b> and <b>status</b> field detected in the CR.	olm-crds-have- validation-test
Owned CRDs Have Resources Listed	This test makes sure that the CRDs for each CR provided via the <b>cr-manifest</b> option have a <b>resources</b> subsection in the <b>owned</b> CRDs section of the ClusterServiceVersion (CSV). If the test detects used resources that are not listed in the resources section, it lists them in the suggestions at the end of the test. Users are required to fill out the resources section after initial code generation for this test to pass.	olm-crds-have- resources-test
Spec Fields With Descriptors	This test verifies that every field in the CRs <b>spec</b> sections has a corresponding descriptor listed in the CSV.	olm-spec- descriptors-test
Status Fields With Descriptors	This test verifies that every field in the CRs <b>status</b> sections have a corresponding descriptor listed in the CSV.	olm-status- descriptors-test

# 5.10.4. Running the scorecard tool

A default set of Kustomize files are generated by the Operator SDK after running the **init** command. The default **bundle/tests/scorecard/config.yaml** file that is generated can be immediately used to run the scorecard tool against your Operator, or you can modify this file to your test specifications.

# **Prerequisites**

• Operator project generated by using the Operator SDK

## Procedure

1. Generate or regenerate your bundle manifests and metadata for your Operator:

\$ make bundle

This command automatically adds scorecard annotations to your bundle metadata, which is used by the **scorecard** command to run tests.

2. Run the scorecard against the on-disk path to your Operator bundle or the name of a bundle image:

\$ operator-sdk scorecard <bundle\_dir\_or\_image>

# 5.10.5. Scorecard output

The **--output** flag for the **scorecard** command specifies the scorecard results output format: either **text** or **json**.

## Example 5.15. Example JSON output snippet

```
"apiVersion": "scorecard.operatorframework.io/v1alpha3",
 "kind": "TestList",
 "items": [
  {
   "kind": "Test",
   "apiVersion": "scorecard.operatorframework.io/v1alpha3",
   "spec": {
    "image": "quay.io/operator-framework/scorecard-test:v1.28.0",
     "entrypoint": [
      "scorecard-test",
      "olm-bundle-validation"
     "labels": {
      "suite": "olm",
      "test": "olm-bundle-validation-test"
    }
   },
   "status": {
    "results": [
       "name": "olm-bundle-validation",
       "log": "time=\"2020-06-10T19:02:49Z\" level=debug msg=\"Found manifests directory\"
name=bundle-test\ntime=\"2020-06-10T19:02:49Z\" level=debug msg=\"Found metadata
directory\" name=bundle-test\ntime=\"2020-06-10T19:02:49Z\" level=debug msg=\"Getting
mediaType info from manifests directory\" name=bundle-test\ntime=\"2020-06-10T19:02:49Z\"
level=info msg=\"Found annotations file\" name=bundle-test\ntime=\"2020-06-10T19:02:49Z\"
level=info msg=\"Could not find optional dependencies file\" name=bundle-test\n",
       "state": "pass"
    1
```

# Example 5.16. Example text output snippet

```
Image: quay.io/operator-framework/scorecard-test:v1.28.0
Entrypoint: [scorecard-test olm-bundle-validation]
Labels:
"suite":"olm"
"test":"olm-bundle-validation-test"
```

Results:

Name: olm-bundle-validation

State: pass

time="2020-07-15T03:19:02Z" level=debug msg="Found manifests directory" name=bundle-test time="2020-07-15T03:19:02Z" level=debug msg="Found metadata directory" name=bundle-test time="2020-07-15T03:19:02Z" level=debug msg="Getting mediaType info from manifests

directory" name=bundle-test

time="2020-07-15T03:19:02Z" level=info msg="Found annotations file" name=bundle-test time="2020-07-15T03:19:02Z" level=info msg="Could not find optional dependencies file" name=bundle-test



#### **NOTE**

The output format spec matches the **Test** type layout.

# 5.10.6. Selecting tests

Scorecard tests are selected by setting the --selector CLI flag to a set of label strings. If a selector flag is not supplied, then all of the tests within the scorecard configuration file are run.

Tests are run serially with test results being aggregated by the scorecard and written to standard output, or stdout.

#### Procedure

1. To select a single test, for example **basic-check-spec-test**, specify the test by using the -selector flag:

```
$ operator-sdk scorecard <bundle_dir_or_image> \
```

- --selector=test=basic-check-spec-test
- 2. To select a suite of tests, for example **olm**, specify a label that is used by all of the OLM tests:

```
$ operator-sdk scorecard <bundle dir or image> \
```

- -o text \
- --selector=suite=olm
- 3. To select multiple tests, specify the test names by using the **selector** flag using the following syntax:

```
$ operator-sdk scorecard <bundle_dir_or_image> \
```

- -o text \
- --selector='test in (basic-check-spec-test,olm-bundle-validation-test)'

# 5.10.7. Enabling parallel testing

As an Operator author, you can define separate stages for your tests using the scorecard configuration file. Stages run sequentially in the order they are defined in the configuration file. A stage contains a list of tests and a configurable parallel setting.

By default, or when a stage explicitly sets **parallel** to **false**, tests in a stage are run sequentially in the order they are defined in the configuration file. Running tests one at a time is helpful to guarantee that no two tests interact and conflict with each other.

However, if tests are designed to be fully isolated, they can be parallelized.

#### Procedure

• To run a set of isolated tests in parallel, include them in the same stage and set **parallel** to **true**:

apiVersion: scorecard.operatorframework.io/v1alpha3

kind: Configuration metadata:

name: config

stages:

- parallel: true 1

tests:

- entrypoint:
- scorecard-test
- basic-check-spec

image: quay.io/operator-framework/scorecard-test:v1.28.0

labels:

suite: basic

test: basic-check-spec-test

- entrypoint:
  - scorecard-test
  - olm-bundle-validation

image: quay.io/operator-framework/scorecard-test:v1.28.0

labels: suite: olm

test: olm-bundle-validation-test

Enables parallel testing

All tests in a parallel stage are executed simultaneously, and scorecard waits for all of them to finish before proceding to the next stage. This can make your tests run much faster.

## 5.10.8. Custom scorecard tests

The scorecard tool can run custom tests that follow these mandated conventions:

- Tests are implemented within a container image
- Tests accept an entrypoint which include a command and arguments
- Tests produce v1alpha3 scorecard output in JSON format with no extraneous logging in the test output
- Tests can obtain the bundle contents at a shared mount point of /bundle
- Tests can access the Kubernetes API using an in-cluster client connection

Writing custom tests in other programming languages is possible if the test image follows the above guidelines.

The following example shows of a custom test image written in Go:

### Example 5.17. Example custom scorecard test

```
// Copyright 2020 The Operator-SDK Authors
//
// Licensed under the Apache License, Version 2.0 (the "License");
// you may not use this file except in compliance with the License.
// You may obtain a copy of the License at
//
    http://www.apache.org/licenses/LICENSE-2.0
//
//
// Unless required by applicable law or agreed to in writing, software
// distributed under the License is distributed on an "AS IS" BASIS,
// WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
// See the License for the specific language governing permissions and
// limitations under the License.
package main
import (
"encoding/json"
"fmt"
"log"
"os"
scapiv1alpha3 "github.com/operator-framework/api/pkg/apis/scorecard/v1alpha3"
apimanifests "github.com/operator-framework/api/pkg/manifests"
)
// This is the custom scorecard test example binary
// As with the Redhat scorecard test image, the bundle that is under
// test is expected to be mounted so that tests can inspect the
// bundle contents as part of their test implementations.
// The actual test is to be run is named and that name is passed
// as an argument to this binary. This argument mechanism allows
// this binary to run various tests all from within a single
// test image.
const PodBundleRoot = "/bundle"
func main() {
entrypoint := os.Args[1:]
if len(entrypoint) == 0 {
 log.Fatal("Test name argument is required")
}
// Read the pod's untar'd bundle from a well-known path.
cfg, err := apimanifests.GetBundleFromDir(PodBundleRoot)
if err != nil {
 log.Fatal(err.Error())
var result scapiv1alpha3.TestStatus
// Names of the custom tests which would be passed in the
```

```
//`operator-sdk` command.
switch entrypoint[0] {
case CustomTest1Name:
 result = CustomTest1(cfg)
case CustomTest2Name:
 result = CustomTest2(cfg)
default:
 result = printValidTests()
// Convert scapiv1alpha3.TestResult to json.
prettyJSON, err := json.MarshalIndent(result, "", " ")
if err != nil {
 log.Fatal("Failed to generate json", err)
fmt.Printf("%s\n", string(prettyJSON))
// printValidTests will print out full list of test names to give a hint to the end user on what the valid
tests are.
func printValidTests() scapiv1alpha3.TestStatus {
result := scapiv1alpha3.TestResult{}
result.State = scapiv1alpha3.FailState
result.Errors = make([]string, 0)
result.Suggestions = make([]string, 0)
str := fmt.Sprintf("Valid tests for this image include: %s %s",
 CustomTest1Name.
 CustomTest2Name)
result.Errors = append(result.Errors, str)
return scapiv1alpha3.TestStatus{
 Results: []scapiv1alpha3.TestResult{result},
}
}
const (
CustomTest1Name = "customtest1"
CustomTest2Name = "customtest2"
// Define any operator specific custom tests here.
// CustomTest1 and CustomTest2 are example test functions. Relevant operator specific
// test logic is to be implemented in similarly.
func CustomTest1(bundle *apimanifests.Bundle) scapiv1alpha3.TestStatus {
r := scapiv1alpha3.TestResult{}
r.Name = CustomTest1Name
r.State = scapiv1alpha3.PassState
r.Errors = make([]string, 0)
r.Suggestions = make([]string, 0)
almExamples := bundle.CSV.GetAnnotations()["alm-examples"]
if almExamples == "" {
 fmt.Println("no alm-examples in the bundle CSV")
```

```
return wrapResult(r)
}
func CustomTest2(bundle *apimanifests.Bundle) scapiv1alpha3.TestStatus {
r := scapiv1alpha3.TestResult{}
r.Name = CustomTest2Name
r.State = scapiv1alpha3.PassState
r.Errors = make([]string, 0)
r.Suggestions = make([]string, 0)
almExamples := bundle.CSV.GetAnnotations()["alm-examples"]
if almExamples == "" {
 fmt.Println("no alm-examples in the bundle CSV")
}
return wrapResult(r)
func wrapResult(r scapiv1alpha3.TestResult) scapiv1alpha3.TestStatus {
return scapiv1alpha3.TestStatus{
 Results: []scapiv1alpha3.TestResult{r},
```

## 5.11. VALIDATING OPERATOR BUNDLES

As an Operator author, you can run the **bundle validate** command in the Operator SDK to validate the content and format of an Operator bundle. You can run the command on a remote Operator bundle image or a local Operator bundle directory.

## 5.11.1. About the bundle validate command

While the Operator SDK **scorecard** command can run tests on your Operator based on a configuration file and test images, the **bundle validate** subcommand can validate local bundle directories and remote bundle images for content and structure.

# bundle validate command syntax

\$ operator-sdk bundle validate <bundle\_dir\_or\_image> <flags>



## NOTE

The **bundle validate** command runs automatically when you build your bundle using the **make bundle** command.

Bundle images are pulled from a remote registry and built locally before they are validated. Local bundle directories must contain Operator metadata and manifests. The bundle metadata and manifests must have a structure similar to the following bundle layout:

## Example bundle layout

Bundle tests pass validation and finish with an exit code of 0 if no errors are detected.

## **Example output**

INFO[0000] All validation tests have completed successfully

Tests fail validation and finish with an exit code of 1 if errors are detected.

## **Example output**

ERRO[0000] Error: Value cache.example.com/v1alpha1, Kind=Memcached: CRD "cache.example.com/v1alpha1, Kind=Memcached" is present in bundle "" but not defined in CSV

Bundle tests that result in warnings can still pass validation with an exit code of **0** as long as no errors are detected. Tests only fail on errors.

## Example output

WARN[0000] Warning: Value : (memcached-operator.v0.0.1) annotations not found INFO[0000] All validation tests have completed successfully

For further information about the **bundle validate** subcommand, run:

\$ operator-sdk bundle validate -h

## 5.11.2. Built-in bundle validate tests

The Operator SDK ships with pre-defined validators arranged into suites. If you run the **bundle validate** command without specifying a validator, the default test runs. The default test verifies that a bundle adheres to the specifications defined by the Operator Framework community. For more information, see "Bundle format".

You can run optional validators to test for issues such as OperatorHub compatibility or deprecated Kubernetes APIs. Optional validators always run in addition to the default test.

# bundle validate command syntax for optional test suites

\$ operator-sdk bundle validate <bundle\_dir\_or\_image>
--select-optional <test\_label>

#### Table 5.18. Addtional bundle validate validators

Name	Description	Label
Operator Framework	This validator tests an Operator bundle against the entire suite of validators provided by the Operator Framework.	suite=operatorfra mework

Name	Description	Label
OperatorHub	This validator tests an Operator bundle for compatibility with OperatorHub.	name=operatorhu b
Good Practices	This validator tests whether an Operator bundle complies with good practices as defined by the Operator Framework. It checks for issues, such as an empty CRD description or unsupported Operator Lifecycle Manager (OLM) resources.	name=good- practices

#### Additional resources

Bundle format

# 5.11.3. Running the bundle validate command

The default validator runs a test every time you enter the **bundle validate** command. You can run optional validators using the **--select-optional** flag. Optional validators run tests in addition to the default test.

## **Prerequisites**

• Operator project generated by using the Operator SDK

#### Procedure

- 1. If you want to run the default validator against a local bundle directory, enter the following command from your Operator project directory:
  - \$ operator-sdk bundle validate ./bundle
- 2. If you want to run the default validator against a remote Operator bundle image, enter the following command:

```
$ operator-sdk bundle validate \
  <bundle_registry>/<bundle_image_name>:<tag>
```

where:

## <bundle\_registry>

Specifies the registry where the bundle is hosted, such as **quay.io/example**.

## <bundle\_image\_name>

Specifies the name of the bundle image, such as **memcached-operator**.

#### <tag>

Specifies the tag of the bundle image, such as v1.28.0.



#### **NOTE**

If you want to validate an Operator bundle image, you must host your image in a remote registry. The Operator SDK pulls the image and builds it locally before running tests. The **bundle validate** command does not support testing local bundle images.

3. If you want to run an additional validator against an Operator bundle, enter the following command:

```
$ operator-sdk bundle validate \
    <bundle_dir_or_image> \
    --select-optional <test_label>
```

where:

### <bundle\_dir\_or\_image>

Specifies the local bundle directory or remote bundle image, such as

~/projects/memcached or quay.io/example/memcached-operator:v1.28.0.

### <test\_label>

Specifies the name of the validator you want to run, such as name=good-practices.

## Example output

ERRO[0000] Error: Value apiextensions.k8s.io/v1, Kind=CustomResource: unsupported media type registry+v1 for bundle object WARN[0000] Warning: Value k8sevent.v0.0.1: owned CRD "k8sevents.k8s.k8sevent.com" has an empty description

# 5.12. HIGH-AVAILABILITY OR SINGLE-NODE CLUSTER DETECTION AND SUPPORT

An OpenShift Container Platform cluster can be configured in high-availability (HA) mode, which uses multiple nodes, or in non-HA mode, which uses a single node. A single-node cluster, also known as single-node OpenShift, is likely to have more conservative resource constraints. Therefore, it is important that Operators installed on a single-node cluster can adjust accordingly and still run well.

By accessing the cluster high-availability mode API provided in OpenShift Container Platform, Operator authors can use the Operator SDK to enable their Operator to detect a cluster's infrastructure topology, either HA or non-HA mode. Custom Operator logic can be developed that uses the detected cluster topology to automatically switch the resource requirements, both for the Operator and for any Operands or workloads it manages, to a profile that best fits the topology.

# 5.12.1. About the cluster high-availability mode API

OpenShift Container Platform provides a cluster high-availability mode API that can be used by Operators to help detect infrastructure topology. The Infrastructure API holds cluster-wide information regarding infrastructure. Operators managed by Operator Lifecycle Manager (OLM) can use the Infrastructure API if they need to configure an Operand or managed workload differently based on the high-availability mode.

In the Infrastructure API, the **infrastructureTopology** status expresses the expectations for infrastructure services that do not run on control plane nodes, usually indicated by a node selector for a **role** value other than **master**. The **controlPlaneTopology** status expresses the expectations for Operands that normally run on control plane nodes.

The default setting for either status is **HighlyAvailable**, which represents the behavior Operators have in multiple node clusters. The **SingleReplica** setting is used in single-node clusters, also known as single-node OpenShift, and indicates that Operators should not configure their Operands for high-availability operation.

The OpenShift Container Platform installer sets the **controlPlaneTopology** and **infrastructureTopology** status fields based on the replica counts for the cluster when it is created, according to the following rules:

- When the control plane replica count is less than 3, the **controlPlaneTopology** status is set to **SingleReplica**. Otherwise, it is set to **HighlyAvailable**.
- When the worker replica count is 0, the control plane nodes are also configured as workers. Therefore, the **infrastructureTopology** status will be the same as the **controlPlaneTopology** status.
- When the worker replica count is 1, the **infrastructureTopology** is set to **SingleReplica**. Otherwise, it is set to **HighlyAvailable**.

# 5.12.2. Example API usage in Operator projects

As an Operator author, you can update your Operator project to access the Infrastructure API by using normal Kubernetes constructs and the **controller-runtime** library, as shown in the following examples:

## controller-runtime library example

```
// Simple query
nn := types.NamespacedName{
Name: "cluster",
}
infraConfig := &configv1.Infrastructure{}
err = crClient.Get(context.Background(), nn, infraConfig)
if err != nil {
return err
}
fmt.Printf("using crclient: %v\n", infraConfig.Status.ControlPlaneTopology)
fmt.Printf("using crclient: %v\n", infraConfig.Status.InfrastructureTopology)
```

## Kubernetes constructs example

```
operatorConfigInformer := configinformer.NewSharedInformerFactoryWithOptions(configClient,
2*time.Second)
infrastructureLister = operatorConfigInformer.Config().V1().Infrastructures().Lister()
infraConfig, err := configClient.ConfigV1().Infrastructures().Get(context.Background(), "cluster",
metav1.GetOptions{})
if err != nil {
    return err
}
```

```
// fmt.Printf("%v\n", infraConfig)
fmt.Printf("%v\n", infraConfig.Status.ControlPlaneTopology)
fmt.Printf("%v\n", infraConfig.Status.InfrastructureTopology)
```

# 5.13. CONFIGURING BUILT-IN MONITORING WITH PROMETHEUS

This guide describes the built-in monitoring support provided by the Operator SDK using the Prometheus Operator and details usage for authors of Go-based and Ansible-based Operators.

## 5.13.1. Prometheus Operator support

Prometheus is an open-source systems monitoring and alerting toolkit. The Prometheus Operator creates, configures, and manages Prometheus clusters running on Kubernetes-based clusters, such as OpenShift Container Platform.

Helper functions exist in the Operator SDK by default to automatically set up metrics in any generated Go-based Operator for use on clusters where the Prometheus Operator is deployed.

# 5.13.2. Exposing custom metrics for Go-based Operators

As an Operator author, you can publish custom metrics by using the global Prometheus registry from the **controller-runtime/pkg/metrics** library.

## **Prerequisites**

- Go-based Operator generated using the Operator SDK
- Prometheus Operator, which is deployed by default on OpenShift Container Platform clusters

#### Procedure

- In your Operator SDK project, uncomment the following line in the config/default/kustomization.yaml file:
  - ../prometheus
- 2. Create a custom controller class to publish additional metrics from the Operator. The following example declares the **widgets** and **widgetFailures** collectors as global variables, and then registers them with the **init()** function in the controller's package:

## Example 5.18. controllers/memcached controller test metrics.go file

```
package controllers

import (
   "github.com/prometheus/client_golang/prometheus"
   "sigs.k8s.io/controller-runtime/pkg/metrics"
)

var (
   widgets = prometheus.NewCounter(
     prometheus.CounterOpts{
        Name: "widgets_total",
```

```
Help: "Number of widgets processed",
},
)
widgetFailures = prometheus.NewCounter(
prometheus.CounterOpts{
Name: "widget_failures_total",
Help: "Number of failed widgets",
},
)
)

func init() {
// Register custom metrics with the global prometheus registry
metrics.Registry.MustRegister(widgets, widgetFailures)
}
```

3. Record to these collectors from any part of the reconcile loop in the **main** controller class, which determines the business logic for the metric:

# Example 5.19. controllers/memcached\_controller.go file

```
func (r *MemcachedReconciler) Reconcile(ctx context.Context, req ctrl.Request)
(ctrl.Result, error) {
...
// Add metrics
widgets.Inc()
widgetFailures.Inc()

return ctrl.Result{}, nil
}
```

4. Build and push the Operator:

\$ make docker-build docker-push IMG=<registry>/<user>/<image\_name>:<tag>

5. Deploy the Operator:

\$ make deploy IMG=<registry>/<user>/<image\_name>:<tag>

6. Create role and role binding definitions to allow the service monitor of the Operator to be scraped by the Prometheus instance of the OpenShift Container Platform cluster.

Roles must be assigned so that service accounts have the permissions to scrape the metrics of the namespace:

#### Example 5.20. config/prometheus/role.yaml role

```
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRole
metadata:
name: prometheus-k8s-role
namespace: memcached-operator-system
```

#### rules:

- apiGroups:

- "

#### resources:

- endpoints
- pods
- services
- nodes
- secrets

#### verbs:

- get
- list
- watch

## Example 5.21. config/prometheus/rolebinding.yaml role binding

apiVersion: rbac.authorization.k8s.io/v1

kind: ClusterRoleBinding

metadata:

name: prometheus-k8s-rolebinding

namespace: memcached-operator-system

roleRef:

apiGroup: rbac.authorization.k8s.io

kind: ClusterRole

name: prometheus-k8s-role

subjects:

 kind: ServiceAccount name: prometheus-k8s

namespace: openshift-monitoring

7. Apply the roles and role bindings for the deployed Operator:

\$ oc apply -f config/prometheus/role.yaml

\$ oc apply -f config/prometheus/rolebinding.yaml

8. Set the labels for the namespace that you want to scrape, which enables OpenShift cluster monitoring for that namespace:

\$ oc label namespace <operator\_namespace> openshift.io/cluster-monitoring="true"

#### Verification

 Query and view the metrics in the OpenShift Container Platform web console. You can use the names that were set in the custom controller class, for example widgets\_total and widget\_failures\_total.

# 5.13.3. Exposing custom metrics for Ansible-based Operators

As an Operator author creating Ansible-based Operators, you can use the Operator SDK's **osdk\_metrics** module to expose custom Operator and Operand metrics, emit events, and support logging.

## **Prerequisites**

- Ansible-based Operator generated using the Operator SDK
- Prometheus Operator, which is deployed by default on OpenShift Container Platform clusters

#### Procedure

1. Generate an Ansible-based Operator. This example uses a **testmetrics.com** domain:

```
$ operator-sdk init \
   --plugins=ansible \
   --domain=testmetrics.com
```

2. Create a **metrics** API. This example uses a **kind** named **Testmetrics**:

```
$ operator-sdk create api \
   --group metrics \
   --version v1 \
   --kind Testmetrics \
   --generate-role
```

3. Edit the **roles/testmetrics/tasks/main.yml** file and use the **osdk\_metrics** module to create custom metrics for your Operator project:

Example 5.22. Example roles/testmetrics/tasks/main.yml file

```
# tasks file for Memcached
- name: start k8sstatus
 k8s:
  definition:
   kind: Deployment
   apiVersion: apps/v1
   metadata:
    name: '{{ ansible_operator_meta.name }}-memcached'
    namespace: '{{ ansible_operator_meta.namespace }}'
   spec:
    replicas: "{{size}}"
    selector:
     matchLabels:
       app: memcached
    template:
     metadata:
       labels:
        app: memcached
     spec:
       containers:
       - name: memcached
        command:
        - memcached
```

```
-m=64
        - -0
        - modern
        image: "docker.io/memcached:1.4.36-alpine"
        ports:
         - containerPort: 11211
- osdk metric:
  name: my_thing_counter
  description: This metric counts things
  counter: {}
- osdk_metric:
  name: my_counter_metric
  description: Add 3.14 to the counter
  counter:
   increment: yes
- osdk metric:
  name: my_gauge_metric
  description: Create my gauge and set it to 2.
  gauge:
   set: 2
osdk_metric:
  name: my_histogram_metric
  description: Observe my histogram
  histogram:
   observe: 2
osdk_metric:
  name: my_summary_metric
  description: Observe my summary
  summary:
   observe: 2
```

## Verification

- 1. Run your Operator on a cluster. For example, to use the "run as a deployment" method:
  - a. Build the Operator image and push it to a registry:
    - \$ make docker-build docker-push IMG=<registry>/<user>/<image\_name>:<tag>
  - b. Install the Operator on a cluster:
    - \$ make install
  - c. Deploy the Operator:
    - \$ make deploy IMG=<registry>/<user>/<image\_name>:<tag>

- 2. Create a **Testmetrics** custom resource (CR):
  - a. Define the CR spec:

## Example 5.23. Example config/samples/metrics\_v1\_testmetrics.yaml file

apiVersion: metrics.testmetrics.com/v1

kind: Testmetrics

metadata:

name: testmetrics-sample

spec: size: 1

b. Create the object:

\$ oc create -f config/samples/metrics\_v1\_testmetrics.yaml

3. Get the pod details:

\$ oc get pods

# **Example output**

NAME READY STATUS RESTARTS AGE ansiblemetrics-controller-manager-<id> 2/2 Running 0 149m testmetrics-sample-memcached-<id> 1/1 Running 0 147m

4. Get the endpoint details:

\$ oc get ep

## **Example output**

NAME ENDPOINTS AGE ansiblemetrics-controller-manager-metrics-service 10.129.2.70:8443 150m

5. Request a custom metrics token:

\$ token=`oc create token prometheus-k8s -n openshift-monitoring`

- 6. Check the metrics values:
  - a. Check the my counter metric value:

\$ oc exec ansiblemetrics-controller-manager-<id> -- curl -k -H "Authoriza tion: Bearer \$token" 'https://10.129.2.70:8443/metrics' | grep my\_counter

## Example output

HELP my\_counter\_metric Add 3.14 to the counter TYPE my\_counter\_metric counter my\_counter\_metric 2 b. Check the my\_gauge\_metric value:

\$ oc exec ansiblemetrics-controller-manager-<id> -- curl -k -H "Authoriza tion: Bearer \$token" 'https://10.129.2.70:8443/metrics' | grep\_gauge

## **Example output**

HELP my\_gauge\_metric Create my gauge and set it to 2.

c. Check the my\_histogram\_metric and my\_summary\_metric values:

\$ oc exec ansiblemetrics-controller-manager-<id> -- curl -k -H "Authoriza tion: Bearer \$token" 'https://10.129.2.70:8443/metrics' | grep Observe

## **Example output**

HELP my\_histogram\_metric Observe my histogram HELP my\_summary\_metric Observe my summary

# 5.14. CONFIGURING LEADER ELECTION

During the lifecycle of an Operator, it is possible that there may be more than one instance running at any given time, for example when rolling out an upgrade for the Operator. In such a scenario, it is necessary to avoid contention between multiple Operator instances using leader election. This ensures only one leader instance handles the reconciliation while the other instances are inactive but ready to take over when the leader steps down.

There are two different leader election implementations to choose from, each with its own trade-off:

#### Leader-for-life

The leader pod only gives up leadership, using garbage collection, when it is deleted. This implementation precludes the possibility of two instances mistakenly running as leaders, a state also known as split brain. However, this method can be subject to a delay in electing a new leader. For example, when the leader pod is on an unresponsive or partitioned node, the **pod-eviction-timeout** dictates long how it takes for the leader pod to be deleted from the node and step down, with a default of **5m**. See the Leader-for-life Go documentation for more.

#### Leader-with-lease

The leader pod periodically renews the leader lease and gives up leadership when it cannot renew the lease. This implementation allows for a faster transition to a new leader when the existing leader is isolated, but there is a possibility of split brain in certain situations. See the Leader-with-lease Go documentation for more.

By default, the Operator SDK enables the Leader-for-life implementation. Consult the related Go documentation for both approaches to consider the trade-offs that make sense for your use case.

## 5.14.1. Operator leader election examples

The following examples illustrate how to use the two leader election options for an Operator, Leader-for-life and Leader-with-lease.

#### 5.14.1.1. Leader-for-life election

With the Leader-for-life election implementation, a call to **leader.Become()** blocks the Operator as it retries until it can become the leader by creating the config map named **memcached-operator-lock**:

```
import (
...
  "github.com/operator-framework/operator-sdk/pkg/leader"
)

func main() {
...
  err = leader.Become(context.TODO(), "memcached-operator-lock")
  if err != nil {
    log.Error(err, "Failed to retry for leader lock")
    os.Exit(1)
  }
  ...
}
```

If the Operator is not running inside a cluster, **leader.Become()** simply returns without error to skip the leader election since it cannot detect the name of the Operator.

#### 5.14.1.2. Leader-with-lease election

The Leader-with-lease implementation can be enabled using the Manager Options for leader election:

```
import (
    ...
    "sigs.k8s.io/controller-runtime/pkg/manager"
)

func main() {
    ...
    opts := manager.Options{
    ...
    LeaderElection: true,
    LeaderElectionID: "memcached-operator-lock"
    }
    mgr, err := manager.New(cfg, opts)
    ...
}
```

When the Operator is not running in a cluster, the Manager returns an error when starting because it cannot detect the namespace of the Operator to create the config map for leader election. You can override this namespace by setting the **Leader Election Namespace** option for the Manager.

## 5.15. OBJECT PRUNING UTILITY FOR GO-BASED OPERATORS

The **operator-lib** pruning utility lets Go-based Operators clean up, or prune, objects when they are no longer needed. Operator authors can also use the utility to create custom hooks and strategies.

## 5.15.1. About the operator-lib pruning utility

Objects, such as jobs or pods, are created as a normal part of the Operator life cycle. If the cluster administrator or the Operator does not remove these object, they can stay in the cluster and consume resources.

Previously, the following options were available for pruning unnecessary objects:

- Operator authors had to create a unique pruning solution for their Operators.
- Cluster administrators had to clean up objects on their own.

The **operator-lib** pruning utility removes objects from a Kubernetes cluster for a given namespace. The library was added in version **0.9.0** of the **operator-lib** library as part of the Operator Framework.

# 5.15.2. Pruning utility configuration

The **operator-lib** pruning utility is written in Go and includes common pruning strategies for Go-based Operators.

## **Example configuration**

```
cfg = Config{
    log:
               logf.Log.WithName("prune"),
    DryRun:
                 false.
    Clientset:
                client.
    LabelSelector: "app=<operator name>",
    Resources: []schema.GroupVersionKind{
         {Group: "", Version: "", Kind: PodKind},
    },
    Namespaces: []string{"<operator_namespace>"},
    Strategy: StrategyConfig{
         Mode:
                      MaxCountStrategy,
         MaxCountSetting: 1,
    PreDeleteHook: myhook,
```

The pruning utility configuration file defines pruning actions by using the following fields:

Configuration field	Description
log	Logger used to handle library log messages.
DryRun	Boolean that determines whether resources should be removed. If set to <b>true</b> , the utility runs but does not to remove resources.
Clientset	Client-go Kubernetes ClientSet used for Kubernetes API calls.
LabelSelector	Kubernetes label selector expression used to find resources to prune.
Resources	Kubernetes resource kinds. <b>PodKind</b> and <b>JobKind</b> are currently supported.

Configuration field	Description
Namespaces	List of Kubernetes namespaces to search for resources.
Strategy	Pruning strategy to run.
Strategy.Mode	MaxCountStrategy, MaxAgeStrategy, or CustomStrategy are currently supported.
Strategy.MaxCountSettin	Integer value for <b>MaxCountStrategy</b> that specifies how many resources should remain after the pruning utility runs.
Strategy.MaxAgeSetting	Go <b>time.Duration</b> string value, such as <b>48h</b> , that specifies the age of resources to prune.
Strategy.CustomSettings	Go map of values that can be passed into a custom strategy function.
PreDeleteHook	Optional: Go function to call before pruning a resource.
CustomStrategy	Optional: Go function that implements a custom pruning strategy.

# **Pruning execution**

You can call the pruning action by running the execute function on the pruning configuration.

err := cfg.Execute(ctx)

You can also call a pruning action by using a cron package or by calling the pruning utility with a triggering event.

# 5.16. MIGRATING PACKAGE MANIFEST PROJECTS TO BUNDLE FORMAT

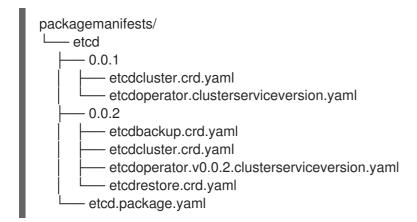
Support for the legacy *package manifest format* for Operators is removed in OpenShift Container Platform 4.8 and later. If you have an Operator project that was initially created using the package manifest format, you can use the Operator SDK to migrate the project to the bundle format. The bundle format is the preferred packaging format for Operator Lifecycle Manager (OLM) starting in OpenShift Container Platform 4.6.

# 5.16.1. About packaging format migration

The Operator SDK **pkgman-to-bundle** command helps in migrating Operator Lifecycle Manager (OLM) package manifests to bundles. The command takes an input package manifest directory and generates bundles for each of the versions of manifests present in the input directory. You can also then build bundle images for each of the generated bundles.

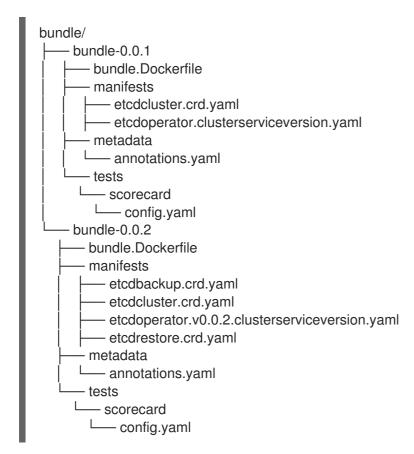
For example, consider the following **packagemanifests**/ directory for a project in the package manifest format:

# Example package manifest format layout



After running the migration, the following bundles are generated in the **bundle**/ directory:

# Example bundle format layout



Based on this generated layout, bundle images for both of the bundles are also built with the following names:

- quay.io/example/etcd:0.0.1
- quay.io/example/etcd:0.0.2

## Additional resources

Operator Framework packaging format

# 5.16.2. Migrating a package manifest project to bundle format

Operator authors can use the Operator SDK to migrate a package manifest format Operator project to a bundle format project.

# **Prerequisites**

- Operator SDK CLI installed
- Operator project initially generated using the Operator SDK in package manifest format

#### Procedure

• Use the Operator SDK to migrate your package manifest project to the bundle format and generate bundle images:

\$ operator-sdk pkgman-to-bundle <package\_manifests\_dir> \ 1 [--output-dir <directory>] \ 2 | --image-tag-base <image\_name\_base> 3

- Specify the location of the package manifests directory for the project, such as **packagemanifests**/ or **manifests**/.
- Optional: By default, the generated bundles are written locally to disk to the **bundle**/ directory. You can use the **--output-dir** flag to specify an alternative location.
- Set the **--image-tag-base** flag to provide the base of the image name, such as **quay.io/example/etcd**, that will be used for the bundles. Provide the name without a tag, because the tag for the images will be set according to the bundle version. For example, the full bundle image names are generated in the format **<image\_name\_base>: <bundle\_version>**.

#### Verification

• Verify that the generated bundle image runs successfully:

\$ operator-sdk run bundle <bundle\_image\_name>:<tag>

## **Example output**

INFO[0025] Successfully created registry pod: quay-io-my-etcd-0-9-4
INFO[0025] Created CatalogSource: etcd-catalog
INFO[0026] OperatorGroup "operator-sdk-og" created
INFO[0026] Created Subscription: etcdoperator-v0-9-4-sub
INFO[0031] Approved InstallPlan install-5t58z for the Subscription: etcdoperator-v0-9-4-sub
INFO[0031] Waiting for ClusterServiceVersion "default/etcdoperator.v0.9.4" to reach
'Succeeded' phase
INFO[0032] Waiting for ClusterServiceVersion "default/etcdoperator.v0.9.4" to appear
INFO[0048] Found ClusterServiceVersion "default/etcdoperator.v0.9.4" phase: Pending
INFO[0049] Found ClusterServiceVersion "default/etcdoperator.v0.9.4" phase: Installing

INFO[0064] Found ClusterServiceVersion "default/etcdoperator.v0.9.4" phase: Succeeded

INFO[0065] OLM has successfully installed "etcdoperator.v0.9.4"

## 5.17. OPERATOR SDK CLI REFERENCE

The Operator SDK command-line interface (CLI) is a development kit designed to make writing Operators easier.

# Operator SDK CLI syntax

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

Operator authors with cluster administrator access to a Kubernetes-based cluster (such as OpenShift Container Platform) can use the Operator SDK CLI to develop their own Operators based on Go, Ansible, or Helm. Kubebuilder is embedded into the Operator SDK as the scaffolding solution for Gobased Operators, which means existing Kubebuilder projects can be used as is with the Operator SDK and continue to work.

## 5.17.1. bundle

The **operator-sdk bundle** command manages Operator bundle metadata.

#### 5.17.1.1. validate

The **bundle validate** subcommand validates an Operator bundle.

Table 5.19. bundle validate flags

Flag	Description
-h,help	Help output for the <b>bundle validate</b> subcommand.
index-builder (string)	Tool to pull and unpack bundle images. Only used when validating a bundle image.  Available options are <b>docker</b> , which is the default, <b>podman</b> , or <b>none</b> .
list-optional	List all optional validators available. When set, no validators are run.
select-optional (string)	Label selector to select optional validators to run. When run with the <b>list-optional</b> flag, lists available optional validators.

# 5.17.2. cleanup

The **operator-sdk cleanup** command destroys and removes resources that were created for an Operator that was deployed with the **run** command.

Table 5.20. cleanup flags

Flag	Description
-h,help	Help output for the <b>run bundle</b> subcommand.
kubeconfig (string)	Path to the <b>kubeconfig</b> file to use for CLI requests.
-n,namespace (string)	If present, namespace in which to run the CLI request.

Flag	Description
timeout <duration></duration>	Time to wait for the command to complete before failing. The default value is <b>2m0s</b> .

# 5.17.3. completion

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

# Table 5.21. completion subcommands

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

# Table 5.22. completion flags

Flag	Description
-h,help	Usage help output.

## For example:

\$ operator-sdk completion bash

# Example output

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

#### 5.17.4. create

The operator-sdk create command is used to create, or scaffold, a Kubernetes API.

# 5.17.4.1. api

The **create api** subcommand scaffolds a Kubernetes API. The subcommand must be run in a project that was initialized with the **init** command.

# Table 5.23. create api flags

Flag	Description
-h,help	Help output for the <b>run bundle</b> subcommand.

# 5.17.5. generate

The **operator-sdk generate** command invokes a specific generator to generate code or manifests.

#### 5.17.5.1. bundle

The **generate bundle** subcommand generates a set of bundle manifests, metadata, and a **bundle.Dockerfile** file for your Operator project.



## **NOTE**

Typically, you run the **generate kustomize manifests** subcommand first to generate the input Kustomize bases that are used by the **generate bundle** subcommand. However, you can use the **make bundle** command in an initialized project to automate running these commands in sequence.

Table 5.24. generate bundle flags

Flag	Description
channels (string)	Comma-separated list of channels to which the bundle belongs. The default value is <b>alpha</b> .
crds-dir (string)	Root directory for <b>CustomResoureDefinition</b> manifests.
default-channel (string)	The default channel for the bundle.
deploy-dir (string)	Root directory for Operator manifests, such as deployments and RBAC. This directory is different from the directory passed to the <b>input-dir</b> flag.
-h,help	Help for <b>generate bundle</b>
input-dir (string)	Directory from which to read an existing bundle. This directory is the parent of your bundle <b>manifests</b> directory and is different from the <b>deploy-dir</b> directory.
kustomize-dir (string)	Directory containing Kustomize bases and a <b>kustomization.yaml</b> file for bundle manifests. The default path is <b>config/manifests</b> .
manifests	Generate bundle manifests.
metadata	Generate bundle metadata and Dockerfile.
output-dir (string)	Directory to write the bundle to.

Flag	Description
overwrite	Overwrite the bundle metadata and Dockerfile if they exist. The default value is <b>true</b> .
package (string)	Package name for the bundle.
-q,quiet	Run in quiet mode.
stdout	Write bundle manifest to standard out.
version (string)	Semantic version of the Operator in the generated bundle. Set only when creating a new bundle or upgrading the Operator.

#### Additional resources

• See Bundling an Operator for a full procedure that includes using the **make bundle** command to call the **generate bundle** subcommand.

# 5.17.5.2. kustomize

The **generate kustomize** subcommand contains subcommands that generate Kustomize data for the Operator.

#### 5.17.5.2.1. manifests

The **generate kustomize manifests** subcommand generates or regenerates Kustomize bases and a **kustomization.yaml** file in the **config/manifests** directory, which are used to build bundle manifests by other Operator SDK commands. This command interactively asks for UI metadata, an important component of manifest bases, by default unless a base already exists or you set the **--interactive=false** flag.

Table 5.25. generate kustomize manifests flags

Flag	Description
apis-dir (string)	Root directory for API type definitions.
-h,help	Help for <b>generate kustomize manifests</b> .
input-dir (string)	Directory containing existing Kustomize files.
interactive	When set to <b>false</b> , if no Kustomize base exists, an interactive command prompt is presented to accept custom metadata.
output-dir (string)	Directory where to write Kustomize files.
package (string)	Package name.

Flag	Description
-q,quiet	Run in quiet mode.

## 5.17.6. init

The **operator-sdk init** command initializes an Operator project and generates, or *scaffolds*, a default project directory layout for the given plugin.

This command writes the following files:

- Boilerplate license file
- **PROJECT** file with the domain and repository
- Makefile to build the project
- **go.mod** file with project dependencies
- kustomization.yaml file for customizing manifests
- Patch file for customizing images for manager manifests
- Patch file for enabling Prometheus metrics
- main.go file to run

## Table 5.26. init flags

Flag	Description
help, -h	Help output for the <b>init</b> command.
plugins (string)	Name and optionally version of the plugin to initialize the project with. Available plugins are ansible.sdk.operatorframework.io/v1, go.kubebuilder.io/v2, go.kubebuilder.io/v3, and helm.sdk.operatorframework.io/v1.
project-version	Project version. Available values are <b>2</b> and <b>3-alpha</b> , which is the default.

## 5.17.7. run

The **operator-sdk run** command provides options that can launch the Operator in various environments.

## 5.17.7.1. bundle

The **run bundle** subcommand deploys an Operator in the bundle format with Operator Lifecycle Manager (OLM).

# Table 5.27. run bundle flags

Flag	Description
index-image (string)	Index image in which to inject a bundle. The default image is <b>quay.io/operator-framework/upstream-opm-builder:latest</b> .
install-mode <install_mode_value &gt;</install_mode_value 	Install mode supported by the cluster service version (CSV) of the Operator, for example <b>AllNamespaces</b> or <b>SingleNamespace</b> .
timeout <duration></duration>	Install timeout. The default value is <b>2m0s</b> .
kubeconfig (string)	Path to the <b>kubeconfig</b> file to use for CLI requests.
-n,namespace (string)	If present, namespace in which to run the CLI request.
security-context- config <security_context></security_context>	Specifies the security context to use for the catalog pod. Allowed values include <b>restricted</b> and <b>legacy</b> . The default value is <b>legacy</b> . [1]
-h,help	Help output for the <b>run bundle</b> subcommand.

 The **restricted** security context is not compatible with the **default** namespace. To configure your Operator's pod security admission in your production environment, see "Complying with pod security admission". For more information about pod security admission, see "Understanding and managing pod security admission".

#### **Additional resources**

- See Operator group membership for details on possible install modes.
- Complying with pod security admission
- Understanding and managing pod security admission

# 5.17.7.2. bundle-upgrade

The **run bundle-upgrade** subcommand upgrades an Operator that was previously installed in the bundle format with Operator Lifecycle Manager (OLM).

Table 5.28. run bundle-upgrade flags

Flag	Description
timeout <duration></duration>	Upgrade timeout. The default value is <b>2m0s</b> .
kubeconfig (string)	Path to the <b>kubeconfig</b> file to use for CLI requests.
-n,namespace (string)	If present, namespace in which to run the CLI request.

Flag	Description	
security-context- config <security_context></security_context>	Specifies the security context to use for the catalog pod. Allowed values include <b>restricted</b> and <b>legacy</b> . The default value is <b>legacy</b> . [1]	
-h,help	Help output for the <b>run bundle</b> subcommand.	

 The **restricted** security context is not compatible with the **default** namespace. To configure your Operator's pod security admission in your production environment, see "Complying with pod security admission". For more information about pod security admission, see "Understanding and managing pod security admission".

### Additional resources

- Complying with pod security admission
- Understanding and managing pod security admission

### 5.17.8. scorecard

The **operator-sdk scorecard** command runs the scorecard tool to validate an Operator bundle and provide suggestions for improvements. The command takes one argument, either a bundle image or directory containing manifests and metadata. If the argument holds an image tag, the image must be present remotely.

Table 5.29. scorecard flags

Flag	Description		
-c,config (string)	Path to scorecard configuration file. The default path is <b>bundle/tests/scorecard/config.yaml</b> .		
-h,help	Help output for the <b>scorecard</b> command.		
kubeconfig (string)	Path to <b>kubeconfig</b> file.		
-L,list	List which tests are available to run.		
<b>-n</b> ,namespace (string)	Namespace in which to run the test images.		
-o,output (string)	Output format for results. Available values are <b>text</b> , which is the default, and <b>json</b> .		
pod-security <security_context></security_context>	Option to run scorecard with the specified security context. Allowed values include <b>restricted</b> and <b>legacy</b> . The default value is <b>legacy</b> . [1]		
-I,selector (string)	Label selector to determine which tests are run.		

Flag	Description	
-s,service-account (string)	Service account to use for tests. The default value is <b>default</b> .	
-x,skip-cleanup	Disable resource cleanup after tests are run.	
-w,wait-time <duration></duration>	Seconds to wait for tests to complete, for example <b>35s</b> . The default value is <b>30s</b> .	

1. The **restricted** security context is not compatible with the **default** namespace. To configure your Operator's pod security admission in your production environment, see "Complying with pod security admission". For more information about pod security admission, see "Understanding and managing pod security admission".

### Additional resources

- See Validating Operators using the scorecard tool for details about running the scorecard tool.
- Complying with pod security admission
- Understanding and managing pod security admission

# **CHAPTER 6. CLUSTER OPERATORS REFERENCE**

This reference guide indexes the *cluster Operators* shipped by Red Hat that serve as the architectural foundation for OpenShift Container Platform. Cluster Operators are installed by default, unless otherwise noted, and are managed by the Cluster Version Operator (CVO). For more details on the control plane architecture, see Operators in OpenShift Container Platform.

Cluster administrators can view cluster Operators in the OpenShift Container Platform web console from the **Administration** → **Cluster Settings** page.



#### **NOTE**

Cluster Operators are not managed by Operator Lifecycle Manager (OLM) and OperatorHub. OLM and OperatorHub are part of the Operator Framework used in OpenShift Container Platform for installing and running optional add-on Operators.

Some of the following cluster Operators can be disabled prior to installation. For more information see cluster capabilities.

### 6.1. CLUSTER BAREMETAL OPERATOR



#### NOTE

The Cluster Baremetal Operator is an optional cluster capability that can be disabled by cluster administrators during installation. For more information about optional cluster capabilities, see "Cluster capabilities" in *Installing*.

#### **Purpose**

The Cluster Baremetal Operator (CBO) deploys all the components necessary to take a bare-metal server to a fully functioning worker node ready to run OpenShift Container Platform compute nodes. The CBO ensures that the metal3 deployment, which consists of the Bare Metal Operator (BMO) and Ironic containers, runs on one of the control plane nodes within the OpenShift Container Platform cluster. The CBO also listens for OpenShift Container Platform updates to resources that it watches and takes appropriate action.

# **Project**

cluster-baremetal-operator

#### Additional resources

Bare-metal capability

# 6.2. BARE METAL EVENT RELAY

#### **Purpose**

The OpenShift Bare Metal Event Relay manages the life-cycle of the Bare Metal Event Relay. The Bare Metal Event Relay enables you to configure the types of cluster event that are monitored using Redfish hardware events.

### Configuration objects

You can use this command to edit the configuration after installation: for example, the webhook port. You can edit configuration objects with:

\$ oc -n [namespace] edit cm hw-event-proxy-operator-manager-config

apiVersion: controller-runtime.sigs.k8s.io/v1alpha1

kind: ControllerManagerConfig

health:

healthProbeBindAddress: :8081

metrics:

bindAddress: 127.0.0.1:8080

webhook: port: 9443 leaderElection: leaderElect: true

resourceName: 6e7a703c.redhat-cne.org

### **Project**

hw-event-proxy-operator

#### **CRD**

The proxy enables applications running on bare-metal clusters to respond quickly to Redfish hardware changes and failures such as breaches of temperature thresholds, fan failure, disk loss, power outages, and memory failure, reported using the HardwareEvent CR.

### hardwareevents.event.redhat-cne.org:

Scope: Namespaced

CR: HardwareEvent

• Validation: Yes

# 6.3. CLOUD CREDENTIAL OPERATOR

#### **Purpose**

The Cloud Credential Operator (CCO) manages cloud provider credentials as Kubernetes custom resource definitions (CRDs). The CCO syncs on **CredentialsRequest** custom resources (CRs) to allow OpenShift Container Platform components to request cloud provider credentials with the specific permissions that are required for the cluster to run.

By setting different values for the **credentialsMode** parameter in the **install-config.yaml** file, the CCO can be configured to operate in several different modes. If no mode is specified, or the **credentialsMode** parameter is set to an empty string (""), the CCO operates in its default mode.

### Project

openshift-cloud-credential-operator

#### **CRDs**

- credentialsrequests.cloudcredential.openshift.io
  - Scope: Namespaced
  - CR: CredentialsRequest
  - Validation: Yes

# Configuration objects

No configuration required.

#### Additional resources

- About the Cloud Credential Operator
- CredentialsRequest custom resource

### 6.4. CLUSTER AUTHENTICATION OPERATOR

### **Purpose**

The Cluster Authentication Operator installs and maintains the **Authentication** custom resource in a cluster and can be viewed with:

\$ oc get clusteroperator authentication -o yaml

### **Project**

cluster-authentication-operator

# 6.5. CLUSTER AUTOSCALER OPERATOR

## **Purpose**

The Cluster Autoscaler Operator manages deployments of the OpenShift Cluster Autoscaler using the **cluster-api** provider.

## **Project**

cluster-autoscaler-operator

### **CRDs**

- ClusterAutoscaler: This is a singleton resource, which controls the configuration autoscaler
  instance for the cluster. The Operator only responds to the ClusterAutoscaler resource named
  default in the managed namespace, the value of the WATCH\_NAMESPACE environment
  variable.
- MachineAutoscaler: This resource targets a node group and manages the annotations to enable and configure autoscaling for that group, the min and max size. Currently only MachineSet objects can be targeted.

# 6.6. CLUSTER CLOUD CONTROLLER MANAGER OPERATOR

# **Purpose**



### NOTE

This Operator is General Availability for Microsoft Azure Stack Hub, IBM Cloud, Nutanix, Red Hat OpenStack Platform (RHOSP), and VMware vSphere.

It is available as a Technology Preview for Alibaba Cloud, Amazon Web Services (AWS), Google Cloud Platform (GCP), IBM Cloud Power VS, and Microsoft Azure.

The Cluster Cloud Controller Manager Operator manages and updates the cloud controller managers deployed on top of OpenShift Container Platform. The Operator is based on the Kubebuilder framework and **controller-runtime** libraries. It is installed via the Cluster Version Operator (CVO).

It contains the following components:

- Operator
- Cloud configuration observer

By default, the Operator exposes Prometheus metrics through the **metrics** service.

# **Project**

cluster-cloud-controller-manager-operator

# 6.7. CLUSTER CAPI OPERATOR



#### **NOTE**

This Operator is available as a Technology Preview for Amazon Web Services (AWS) and Google Cloud Platform (GCP).

### **Purpose**

The Cluster CAPI Operator maintains the lifecycle of Cluster API resources. This Operator is responsible for all administrative tasks related to deploying the Cluster API project within an OpenShift Container Platform cluster.

### **Project**

cluster-capi-operator

#### **CRDs**

- awsmachines.infrastructure.cluster.x-k8s.io
  - Scope: Namespaced
  - CR: awsmachine
  - Validation: No
- gcpmachines.infrastructure.cluster.x-k8s.io
  - Scope: Namespaced
  - CR: gcpmachine
  - Validation: No
- awsmachinetemplates.infrastructure.cluster.x-k8s.io
  - Scope: Namespaced
  - CR: awsmachinetemplate
  - Validation: No
- gcpmachinetemplates.infrastructure.cluster.x-k8s.io
  - Scope: Namespaced

- CR: gcpmachinetemplate
- Validation: No

### 6.8. CLUSTER CONFIG OPERATOR

### **Purpose**

The Cluster Config Operator performs the following tasks related to config.openshift.io:

- Creates CRDs.
- Renders the initial custom resources.
- Handles migrations.

### **Project**

cluster-config-operator

# 6.9. CLUSTER CSI SNAPSHOT CONTROLLER OPERATOR



#### **NOTE**

The Cluster CSI Snapshot Controller Operator is an optional cluster capability that can be disabled by cluster administrators during installation. For more information about optional cluster capabilities, see "Cluster capabilities" in *Installing*.

### **Purpose**

The Cluster CSI Snapshot Controller Operator installs and maintains the CSI Snapshot Controller. The CSI Snapshot Controller is responsible for watching the **VolumeSnapshot** CRD objects and manages the creation and deletion lifecycle of volume snapshots.

#### **Project**

cluster-csi-snapshot-controller-operator

#### Additional resources

CSI snapshot controller capability

# 6.10. CLUSTER IMAGE REGISTRY OPERATOR

### **Purpose**

The Cluster Image Registry Operator manages a singleton instance of the OpenShift image registry. It manages all configuration of the registry, including creating storage.

On initial start up, the Operator creates a default **image-registry** resource instance based on the configuration detected in the cluster. This indicates what cloud storage type to use based on the cloud provider.

If insufficient information is available to define a complete **image-registry** resource, then an incomplete resource is defined and the Operator updates the resource status with information about what is missing.

The Cluster Image Registry Operator runs in the **openshift-image-registry** namespace and it also manages the registry instance in that location. All configuration and workload resources for the registry reside in that namespace.

### **Project**

cluster-image-registry-operator

# 6.11. CLUSTER MACHINE APPROVER OPERATOR

# **Purpose**

The Cluster Machine Approver Operator automatically approves the CSRs requested for a new worker node after cluster installation.



#### NOTE

For the control plane node, the **approve-csr** service on the bootstrap node automatically approves all CSRs during the cluster bootstrapping phase.

### **Project**

cluster-machine-approver-operator

# 6.12. CLUSTER MONITORING OPERATOR

## **Purpose**

The Cluster Monitoring Operator manages and updates the Prometheus-based cluster monitoring stack deployed on top of OpenShift Container Platform.

### **Project**

openshift-monitoring

## **CRDs**

- alertmanagers.monitoring.coreos.com
  - Scope: Namespaced
  - CR: alertmanager
  - Validation: Yes
- prometheuses.monitoring.coreos.com
  - Scope: Namespaced
  - CR: prometheus
  - Validation: Yes
- prometheusrules.monitoring.coreos.com
  - Scope: Namespaced
  - CR: prometheusrule
  - Validation: Yes

# servicemonitors.monitoring.coreos.com

Scope: Namespaced

CR: servicemonitor

Validation: Yes

## **Configuration objects**

\$ oc -n openshift-monitoring edit cm cluster-monitoring-config

# 6.13. CLUSTER NETWORK OPERATOR

## **Purpose**

The Cluster Network Operator installs and upgrades the networking components on an OpenShift Container Platform cluster.

## 6.14. CLUSTER SAMPLES OPERATOR



#### NOTE

The Cluster Samples Operator is an optional cluster capability that can be disabled by cluster administrators during installation. For more information about optional cluster capabilities, see "Cluster capabilities" in *Installing*.

### **Purpose**

The Cluster Samples Operator manages the sample image streams and templates stored in the **openshift** namespace.

On initial start up, the Operator creates the default samples configuration resource to initiate the creation of the image streams and templates. The configuration object is a cluster scoped object with the key **cluster** and type **configs.samples**.

The image streams are the Red Hat Enterprise Linux CoreOS (RHCOS)-based OpenShift Container Platform image streams pointing to images on **registry.redhat.io**. Similarly, the templates are those categorized as OpenShift Container Platform templates.

The Cluster Samples Operator deployment is contained within the **openshift-cluster-samples-operator** namespace. On start up, the install pull secret is used by the image stream import logic in the OpenShift image registry and API server to authenticate with **registry.redhat.io**. An administrator can create any additional secrets in the **openshift** namespace if they change the registry used for the sample image streams. If created, those secrets contain the content of a **config.json** for **docker** needed to facilitate image import.

The image for the Cluster Samples Operator contains image stream and template definitions for the associated OpenShift Container Platform release. After the Cluster Samples Operator creates a sample, it adds an annotation that denotes the OpenShift Container Platform version that it is compatible with. The Operator uses this annotation to ensure that each sample matches the compatible release version. Samples outside of its inventory are ignored, as are skipped samples.

Modifications to any samples that are managed by the Operator are allowed as long as the version annotation is not modified or deleted. However, on an upgrade, as the version annotation will change, those modifications can get replaced as the sample will be updated with the newer version. The Jenkins

images are part of the image payload from the installation and are tagged into the image streams directly.

The samples resource includes a finalizer, which cleans up the following upon its deletion:

- Operator-managed image streams
- Operator-managed templates
- Operator-generated configuration resources
- Cluster status resources

Upon deletion of the samples resource, the Cluster Samples Operator recreates the resource using the default configuration.

## **Project**

cluster-samples-operator

#### Additional resources

OpenShift samples capability

# 6.15. CLUSTER STORAGE OPERATOR



#### NOTE

The Cluster Storage Operator is an optional cluster capability that can be disabled by cluster administrators during installation. For more information about optional cluster capabilities, see "Cluster capabilities" in *Installing*.

### **Purpose**

The Cluster Storage Operator sets OpenShift Container Platform cluster-wide storage defaults. It ensures a default **storageclass** exists for OpenShift Container Platform clusters. It also installs Container Storage Interface (CSI) drivers which enable your cluster to use various storage backends.

#### **Project**

cluster-storage-operator

# Configuration

No configuration is required.

### Notes

 The storage class that the Operator creates can be made non-default by editing its annotation, but this storage class cannot be deleted as long as the Operator runs.

#### Additional resources

Storage capability

### 6.16. CLUSTER VERSION OPERATOR

### **Purpose**

Cluster Operators manage specific areas of cluster functionality. The Cluster Version Operator (CVO) manages the lifecycle of cluster Operators, many of which are installed in OpenShift Container Platform by default.

The CVO also checks with the OpenShift Update Service to see the valid updates and update paths based on current component versions and information in the graph.

### **Project**

cluster-version-operator

### 6.17. CONSOLE OPERATOR



### **NOTE**

The Console Operator is an optional cluster capability that can be disabled by cluster administrators during installation. If you disable the Console Operator at installation, your cluster is still supported and upgradable. For more information about optional cluster capabilities, see "Cluster capabilities" in *Installing*.

### **Purpose**

The Console Operator installs and maintains the OpenShift Container Platform web console on a cluster. The Console Operator is installed by default and automatically maintains a console.

# **Project**

console-operator

### Additional resources

Web console capability

# 6.18. CONTROL PLANE MACHINE SET OPERATOR



#### **NOTE**

This Operator is available for Amazon Web Services (AWS), Google Cloud Platform (GCP), Microsoft Azure, and VMware vSphere.

#### **Purpose**

The Control Plane Machine Set Operator automates the management of control plane machine resources within an OpenShift Container Platform cluster.

### **Project**

cluster-control-plane-machine-set-operator

### **CRDs**

- controlplanemachineset.machine.openshift.io
  - Scope: Namespaced
  - CR: ControlPlaneMachineSet
  - Validation: Yes

#### Additional resources

- About control plane machine sets
- ControlPlaneMachineSet custom resource

# 6.19. DNS OPERATOR

### **Purpose**

The DNS Operator deploys and manages CoreDNS to provide a name resolution service to pods that enables DNS-based Kubernetes Service discovery in OpenShift Container Platform.

The Operator creates a working default deployment based on the cluster's configuration.

- The default cluster domain is **cluster.local**.
- Configuration of the CoreDNS Corefile or Kubernetes plugin is not yet supported.

The DNS Operator manages CoreDNS as a Kubernetes daemon set exposed as a service with a static IP. CoreDNS runs on all nodes in the cluster.

# **Project**

cluster-dns-operator

# 6.20. ETCD CLUSTER OPERATOR

### **Purpose**

The etcd cluster Operator automates etcd cluster scaling, enables etcd monitoring and metrics, and simplifies disaster recovery procedures.

### **Project**

cluster-etcd-operator

### **CRDs**

- etcds.operator.openshift.io
  - Scope: Cluster
  - o CR: etcd
  - Validation: Yes

### Configuration objects

\$ oc edit etcd cluster

### 6.21. INGRESS OPERATOR

#### Purpose

The Ingress Operator configures and manages the OpenShift Container Platform router.

### **Project**

openshift-ingress-operator

#### **CRDs**

• clusteringresses.ingress.openshift.io

Scope: Namespaced

CR: clusteringresses

• Validation: No

# Configuration objects

Cluster config

• Type Name: clusteringresses.ingress.openshift.io

Instance Name: default

• View Command:

\$ oc get clusteringresses.ingress.openshift.io -n openshift-ingress-operator default -o yaml

#### **Notes**

The Ingress Operator sets up the router in the **openshift-ingress** project and creates the deployment for the router:

\$ oc get deployment -n openshift-ingress

The Ingress Operator uses the **clusterNetwork[].cidr** from the **network/cluster** status to determine what mode (IPv4, IPv6, or dual stack) the managed Ingress Controller (router) should operate in. For example, if **clusterNetwork** contains only a v6 **cidr**, then the Ingress Controller operates in IPv6-only mode.

In the following example, Ingress Controllers managed by the Ingress Operator will run in IPv4-only mode because only one cluster network exists and the network is an IPv4 **cidr**:

\$ oc get network/cluster -o jsonpath='{.status.clusterNetwork[\*]}'

# Example output

map[cidr:10.128.0.0/14 hostPrefix:23]

### 6.22. INSIGHTS OPERATOR



### NOTE

The Insights Operator is an optional cluster capability that can be disabled by cluster administrators during installation. For more information about optional cluster capabilities, see "Cluster capabilities" in *Installing*.

#### **Purpose**

The Insights Operator gathers OpenShift Container Platform configuration data and sends it to Red

Hat. The data is used to produce proactive insights recommendations about potential issues that a cluster might be exposed to. These insights are communicated to cluster administrators through Insights Advisor on console.redhat.com.

### **Project**

insights-operator

# Configuration

No configuration is required.

#### **Notes**

Insights Operator complements OpenShift Container Platform Telemetry.

#### Additional resources

- Insights capability
- See About remote health monitoring for details about Insights Operator and Telemetry.

# 6.23. KUBERNETES API SERVER OPERATOR

### **Purpose**

The Kubernetes API Server Operator manages and updates the Kubernetes API server deployed on top of OpenShift Container Platform. The Operator is based on the OpenShift Container Platform **library-go** framework and it is installed using the Cluster Version Operator (CVO).

### **Project**

openshift-kube-apiserver-operator

#### **CRDs**

- kubeapiservers.operator.openshift.io
  - Scope: Cluster
  - CR: kubeapiserver
  - Validation: Yes

# **Configuration objects**

\$ oc edit kubeapiserver

### 6.24. KUBERNETES CONTROLLER MANAGER OPERATOR

#### Purpose

The Kubernetes Controller Manager Operator manages and updates the Kubernetes Controller Manager deployed on top of OpenShift Container Platform. The Operator is based on OpenShift Container Platform **library-go** framework and it is installed via the Cluster Version Operator (CVO).

It contains the following components:

- Operator
- Bootstrap manifest renderer

- Installer based on static pods
- Configuration observer

By default, the Operator exposes Prometheus metrics through the **metrics** service.

### **Project**

cluster-kube-controller-manager-operator

# 6.25. KUBERNETES SCHEDULER OPERATOR

### **Purpose**

The Kubernetes Scheduler Operator manages and updates the Kubernetes Scheduler deployed on top of OpenShift Container Platform. The Operator is based on the OpenShift Container Platform **library-go** framework and it is installed with the Cluster Version Operator (CVO).

The Kubernetes Scheduler Operator contains the following components:

- Operator
- Bootstrap manifest renderer
- Installer based on static pods
- Configuration observer

By default, the Operator exposes Prometheus metrics through the metrics service.

### **Project**

cluster-kube-scheduler-operator

### Configuration

The configuration for the Kubernetes Scheduler is the result of merging:

- a default configuration.
- an observed configuration from the spec **schedulers.config.openshift.io**.

All of these are sparse configurations, invalidated JSON snippets which are merged to form a valid configuration at the end.

### 6.26. KUBERNETES STORAGE VERSION MIGRATOR OPERATOR

### **Purpose**

The Kubernetes Storage Version Migrator Operator detects changes of the default storage version, creates migration requests for resource types when the storage version changes, and processes migration requests.

### **Project**

cluster-kube-storage-version-migrator-operator

# 6.27. MACHINE API OPERATOR

## **Purpose**

The Machine API Operator manages the lifecycle of specific purpose custom resource definitions (CRD), controllers, and RBAC objects that extend the Kubernetes API. This declares the desired state of machines in a cluster.

### **Project**

machine-api-operator

#### **CRDs**

- MachineSet
- Machine
- MachineHealthCheck

# 6.28. MACHINE CONFIG OPERATOR

### **Purpose**

The Machine Config Operator manages and applies configuration and updates of the base operating system and container runtime, including everything between the kernel and kubelet.

There are four components:

- machine-config-server: Provides Ignition configuration to new machines joining the cluster.
- machine-config-controller: Coordinates the upgrade of machines to the desired configurations defined by a MachineConfig object. Options are provided to control the upgrade for sets of machines individually.
- **machine-config-daemon**: Applies new machine configuration during update. Validates and verifies the state of the machine to the requested machine configuration.
- **machine-config**: Provides a complete source of machine configuration at installation, first start up, and updates for a machine.



### **IMPORTANT**

Currently, there is no supported way to block or restrict the machine config server endpoint. The machine config server must be exposed to the network so that newly-provisioned machines, which have no existing configuration or state, are able to fetch their configuration. In this model, the root of trust is the certificate signing requests (CSR) endpoint, which is where the kubelet sends its certificate signing request for approval to join the cluster. Because of this, machine configs should not be used to distribute sensitive information, such as secrets and certificates.

To ensure that the machine config server endpoints, ports 22623 and 22624, are secured in bare metal scenarios, customers must configure proper network policies.

#### Additional resources

• About the OpenShift SDN network plugin .

## **Project**

openshift-machine-config-operator

# 6.29. MARKETPLACE OPERATOR



#### **NOTE**

The Marketplace Operator is an optional cluster capability that can be disabled by cluster administrators if it is not needed. For more information about optional cluster capabilities, see "Cluster capabilities" in *Installing*.

## **Purpose**

The Marketplace Operator simplifies the process for bringing off-cluster Operators to your cluster by using a set of default Operator Lifecycle Manager (OLM) catalogs on the cluster. When the Marketplace Operator is installed, it creates the **openshift-marketplace** namespace. OLM ensures catalog sources installed in the **openshift-marketplace** namespace are available for all namespaces on the cluster.

### **Project**

operator-marketplace

#### Additional resources

Marketplace capability

# 6.30. NODE TUNING OPERATOR

### **Purpose**

The Node Tuning Operator helps you manage node-level tuning by orchestrating the TuneD daemon and achieves low latency performance by using the Performance Profile controller. The majority of high-performance applications require some level of kernel tuning. The Node Tuning Operator provides a unified management interface to users of node-level sysctls and more flexibility to add custom tuning specified by user needs.

The Operator manages the containerized TuneD daemon for OpenShift Container Platform as a Kubernetes daemon set. It ensures the custom tuning specification is passed to all containerized TuneD daemons running in the cluster in the format that the daemons understand. The daemons run on all nodes in the cluster, one per node.

Node-level settings applied by the containerized TuneD daemon are rolled back on an event that triggers a profile change or when the containerized TuneD daemon is terminated gracefully by receiving and handling a termination signal.

The Node Tuning Operator uses the Performance Profile controller to implement automatic tuning to achieve low latency performance for OpenShift Container Platform applications.

The cluster administrator configures a performance profile to define node-level settings such as the following:

- Updating the kernel to kernel-rt.
- Choosing CPUs for housekeeping.
- Choosing CPUs for running workloads.



#### **NOTE**

Currently, disabling CPU load balancing is not supported by cgroup v2. As a result, you might not get the desired behavior from performance profiles if you have cgroup v2 enabled. Enabling cgroup v2 is not recommended if you are using performance profiles.

The Node Tuning Operator is part of a standard OpenShift Container Platform installation in version 4.1 and later.



### **NOTE**

In earlier versions of OpenShift Container Platform, the Performance Addon Operator was used to implement automatic tuning to achieve low latency performance for OpenShift applications. In OpenShift Container Platform 4.11 and later, this functionality is part of the Node Tuning Operator.

### **Project**

cluster-node-tuning-operator

#### Additional resources

Low latency tuning of OCP nodes

# 6.31. OPENSHIFT API SERVER OPERATOR

# **Purpose**

The OpenShift API Server Operator installs and maintains the openshift-apiserver on a cluster.

#### **Project**

openshift-apiserver-operator

#### **CRDs**

- openshiftapiservers.operator.openshift.io
  - Scope: Cluster
  - CR: openshiftapiserver
  - Validation: Yes

# 6.32. OPENSHIFT CONTROLLER MANAGER OPERATOR

#### Purpose

The OpenShift Controller Manager Operator installs and maintains the **OpenShiftControllerManager** custom resource in a cluster and can be viewed with:

\$ oc get clusteroperator openshift-controller-manager -o yaml

The custom resource definition (CRD) **openshiftcontrollermanagers.operator.openshift.io** can be viewed in a cluster with:

\$ oc get crd openshiftcontrollermanagers.operator.openshift.io -o yaml

### **Project**

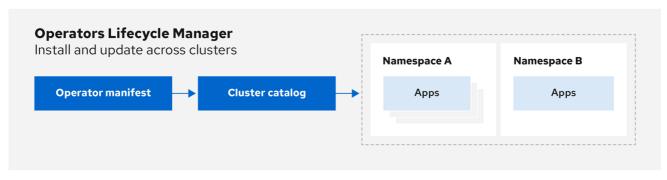
cluster-openshift-controller-manager-operator

# 6.33. OPERATOR LIFECYCLE MANAGER OPERATORS

# **Purpose**

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

Figure 6.1. Operator Lifecycle Manager workflow



OpenShift 43 1019

OLM runs by default in OpenShift Container Platform 4.13, which aids cluster administrators in installing, upgrading, and granting access to Operators running on their cluster. The OpenShift Container Platform web console provides management screens for cluster administrators to install Operators, as well as grant specific projects access to use the catalog of Operators available on the cluster.

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

### **CRDs**

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

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

Table 6.1. CRDs managed by OLM and Catalog Operators

Resource	Shor t nam e	Own er	Description
ClusterServic eVersion (CSV)	CSV	OLM	Application metadata: name, version, icon, required resources, installation, and so on.
InstallPlan	ip	Catal og	Calculated list of resources to be created to automatically install or upgrade a CSV.

Resource	Shor t nam e	Own er	Description
CatalogSour ce	cats rc	Catal og	A repository of CSVs, CRDs, and packages that define an application.
Subscription	sub	Catal og	Used to keep CSVs up to date by tracking a channel in a package.
OperatorGro up	og	OLM	Configures all Operators deployed in the same namespace as the <b>OperatorGroup</b> object to watch for their custom resource (CR) in a list of namespaces or cluster-wide.

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

Table 6.2. Resources created by OLM and Catalog Operators

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

# **OLM Operator**

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

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

The OLM Operator uses the following workflow:

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



### **NOTE**

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

### **Catalog Operator**

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

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

The Catalog Operator uses the following workflow:

- 1. Connect to each catalog source in the cluster.
- 2. Watch for unresolved install plans created by a user, and if found:
  - a. Find the CSV matching the name requested and add the CSV as a resolved resource.
  - b. For each managed or required CRD, add the CRD as a resolved resource.
  - c. For each required CRD, find the CSV that manages it.
- 3. Watch for resolved install plans and create all of the discovered resources for it, if approved by a user or automatically.
- 4. Watch for catalog sources and subscriptions and create install plans based on them.

### **Catalog Registry**

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

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

### Additional resources

• For more information, see the sections on understanding Operator Lifecycle Manager (OLM).

# 6.34. OPENSHIFT SERVICE CA OPERATOR

#### **Purpose**

The OpenShift Service CA Operator mints and manages serving certificates for Kubernetes services.

### **Project**

openshift-service-ca-operator

### 6.35. VSPHERE PROBLEM DETECTOR OPERATOR

# **Purpose**

The vSphere Problem Detector Operator checks clusters that are deployed on vSphere for common installation and misconfiguration issues that are related to storage.



### NOTE

The vSphere Problem Detector Operator is only started by the Cluster Storage Operator when the Cluster Storage Operator detects that the cluster is deployed on vSphere.

# Configuration

No configuration is required.

#### **Notes**

- The Operator supports OpenShift Container Platform installations on vSphere.
- The Operator uses the **vsphere-cloud-credentials** to communicate with vSphere.
- The Operator performs checks that are related to storage.

#### Additional resources

• For more details, see Using the vSphere Problem Detector Operator.