



OpenShift Container Platform 4.1

Applications

Creating and managing applications on OpenShift Container Platform 4.1

OpenShift Container Platform 4.1 Applications

Creating and managing applications on OpenShift Container Platform 4.1

Legal Notice

Copyright © 2020 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[®] is a registered trademark of Oracle and/or its affiliates.

XFS[®] 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[®] 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 instructions for the various ways to create and manage instances of user-provisioned applications running on OpenShift Container Platform. It introduces concepts and tasks related to building applications using the Operator Framework, as well as provisioning applications using the Open Service Broker API.

Table of Contents

CHAPTER 1. PROJECTS	8
1.1. WORKING WITH PROJECTS	8
1.1.1. Creating a project using the web console	8
1.1.2. Creating a project using the CLI	8
1.1.3. Viewing a project using the web console	9
1.1.4. Viewing a project using the CLI	9
1.1.5. Adding to a project	9
1.1.6. Checking project status using the web console	9
1.1.7. Checking project status using the CLI	10
1.1.8. Deleting a project using the web console	10
1.1.9. Deleting a project using the CLI	10
1.2. CREATING A PROJECT AS ANOTHER USER	10
1.2.1. API impersonation	10
1.2.2. Impersonating a user when you create a project	10
1.3. CONFIGURING PROJECT CREATION	11
1.3.1. About project creation	11
1.3.2. Modifying the template for new projects	11
1.3.3. Disabling project self-provisioning	12
1.3.4. Customizing the project request message	14
CHAPTER 2. OPERATORS	16
2.1. UNDERSTANDING OPERATORS	16
2.1.1. Why use Operators?	16
2.1.2. Operator Framework	16
2.1.3. Operator maturity model	17
2.2. UNDERSTANDING THE OPERATOR LIFECYCLE MANAGER	17
2.2.1. Overview of the Operator Lifecycle Manager	18
2.2.2. ClusterServiceVersions (CSVs)	18
2.2.3. Operator Lifecycle Manager architecture	19
2.2.3.1. OLM Operator	20
2.2.3.2. Catalog Operator	20
2.2.3.3. Catalog Registry	21
2.2.4. OperatorGroups	21
2.2.4.1. OperatorGroup membership	21
2.2.4.1.1. Troubleshooting OperatorGroup membership	22
2.2.4.2. Target namespace selection	22
2.2.4.3. OperatorGroup CSV annotations	23
2.2.4.4. Provided APIs annotation	23
2.2.4.5. Role-based access control	24
2.2.4.6. Copied CSVs	27
2.2.4.7. Static OperatorGroups	27
2.2.4.8. OperatorGroup intersection	28
2.2.4.8.1. Rules for intersection	28
2.2.5. Metrics	29
2.3. UNDERSTANDING THE OPERATORHUB	29
2.3.1. Overview of the OperatorHub	29
2.3.2. OperatorHub architecture	30
2.3.2.1. OperatorSource	30
2.3.2.2. CatalogSourceConfig	31
2.4. ADDING OPERATORS TO A CLUSTER	31
2.4.1. Installing Operators from the OperatorHub	31

2.4.1.1. Installing from the OperatorHub using the web console	32
2.4.1.2. Installing from the OperatorHub using the CLI	35
2.5. DELETING OPERATORS FROM A CLUSTER	38
2.5.1. Deleting Operators from a cluster using the web console	38
2.5.2. Deleting Operators from a cluster using the CLI	39
2.6. CREATING APPLICATIONS FROM INSTALLED OPERATORS	40
2.6.1. Creating an etcd cluster using an Operator	40
2.7. MANAGING RESOURCES FROM CUSTOM RESOURCE DEFINITIONS	42
2.7.1. Custom Resource Definitions	43
2.7.2. Creating Custom Resources from a file	43
2.7.3. Inspecting Custom Resources	44
CHAPTER 3. APPLICATION LIFE CYCLE MANAGEMENT	46
3.1. CREATING APPLICATIONS	46
3.1.1. Creating an application by using the CLI	46
3.1.1.1. Creating an application from source code	46
3.1.1.1.1. Local	46
3.1.1.1.2. Remote	46
3.1.1.1.3. Build strategy detection	47
3.1.1.1.4. Language Detection	47
3.1.1.2. Creating an application from an image	48
3.1.1.2.1. DockerHub MySQL image	49
3.1.1.2.2. Image in a private registry	49
3.1.1.2.3. Existing imagestream and optional imagestreamtag	49
3.1.1.3. Creating an application from a template	49
3.1.1.3.1. Template Parameters	49
3.1.1.4. Modifying application creation	50
3.1.1.4.1. Specifying environment variables	50
3.1.1.4.2. Specifying build environment variables	51
3.1.1.4.3. Specifying labels	51
3.1.1.4.4. Viewing the output without creation	51
3.1.1.4.5. Creating objects with different names	51
3.1.1.4.6. Creating objects in a different project	52
3.1.1.4.7. Creating multiple objects	52
3.1.1.4.8. Grouping images and source in a single Pod	52
3.1.1.4.9. Searching for images, templates, and other inputs	52
CHAPTER 4. SERVICE BROKERS	53
4.1. INSTALLING THE SERVICE CATALOG	53
4.1.1. About the service catalog	53
4.1.2. Installing service catalog	53
4.1.3. Uninstalling service catalog	54
4.2. INSTALLING THE TEMPLATE SERVICE BROKER	55
4.2.1. About the Template Service Broker	55
4.2.2. Installing the Template Service Broker Operator	55
4.2.3. Starting the Template Service Broker	56
4.3. PROVISIONING TEMPLATE APPLICATIONS	57
4.3.1. Provisioning template applications	57
4.4. UNINSTALLING THE TEMPLATE SERVICE BROKER	58
4.4.1. Uninstalling the Template Service Broker	58
4.5. INSTALLING THE OPENSIFT ANSIBLE BROKER	59
4.5.1. About the OpenShift Ansible Broker	59
4.5.1.1. Ansible playbook bundles	60

4.5.2. Installing the OpenShift Ansible Service Broker Operator	60
4.5.3. Starting the OpenShift Ansible Broker	61
4.5.3.1. OpenShift Ansible Broker configuration options	62
4.6. CONFIGURING THE OPENSIFT ANSIBLE BROKER	64
4.6.1. Configuring the OpenShift Ansible Broker	64
4.6.1.1. OpenShift Ansible Broker configuration options	64
4.6.2. Configuring monitoring for the OpenShift Ansible Broker	66
4.7. PROVISIONING SERVICE BUNDLES	67
4.7.1. Provisioning service bundles	67
4.8. UNINSTALLING THE OPENSIFT ANSIBLE BROKER	68
4.8.1. Uninstalling the OpenShift Ansible Broker	68
CHAPTER 5. DEPLOYMENTS	70
5.1. UNDERSTANDING DEPLOYMENTS AND DEPLOYMENTCONFIGS	70
5.1.1. Building blocks of a deployment	70
5.1.1.1. ReplicationControllers	70
5.1.1.2. ReplicaSets	71
5.1.2. DeploymentConfigs	72
5.1.3. Deployments	74
5.1.4. Comparing Deployments and DeploymentConfigs	74
5.1.4.1. Design	74
5.1.4.2. DeploymentConfigs-specific features	75
Automatic rollbacks	75
Triggers	75
Lifecycle hooks	75
Custom strategies	75
5.1.4.3. Deployments-specific features	75
Rollover	75
Proportional scaling	75
Pausing mid-rollout	75
5.2. MANAGING DEPLOYMENT PROCESSES	76
5.2.1. Managing DeploymentConfigs	76
5.2.1.1. Starting a deployment	76
5.2.1.2. Viewing a deployment	76
5.2.1.3. Retrying a deployment	76
5.2.1.4. Rolling back a deployment	77
5.2.1.5. Executing commands inside a container	77
5.2.1.6. Viewing deployment logs	78
5.2.1.7. Deployment triggers	78
ConfigChange deployment triggers	79
ImageChange deployment triggers	79
5.2.1.7.1. Setting deployment triggers	80
5.2.1.8. Setting deployment resources	80
5.2.1.9. Scaling manually	81
5.2.1.10. Accessing private repositories from DeploymentConfigs	81
5.2.1.11. Assigning pods to specific nodes	82
5.2.1.12. Running a Pod with a different service account	82
5.3. USING DEPLOYMENTCONFIG STRATEGIES	82
5.3.1. Rolling strategy	83
5.3.1.1. Canary deployments	85
5.3.1.2. Creating a Rolling deployment	85
5.3.2. Recreate strategy	86
5.3.3. Custom strategy	87

5.3.4. Lifecycle hooks	88
Pod-based lifecycle hook	88
5.3.4.1. Setting lifecycle hooks	90
5.4. USING ROUTE-BASED DEPLOYMENT STRATEGIES	90
5.4.1. Proxy shards and traffic splitting	90
5.4.2. N-1 compatibility	90
5.4.3. Graceful termination	91
5.4.4. Blue-green deployments	91
5.4.4.1. Setting up a blue-green deployment	91
5.4.5. A/B deployments	92
5.4.5.1. Load balancing for A/B testing	92
5.4.5.1.1. Managing weights using the web console	94
5.4.5.1.2. Managing weights using the CLI	94
5.4.5.1.3. One service, multiple DeploymentConfigs	95
CHAPTER 6. CRDS	97
6.1. EXTENDING THE KUBERNETES API WITH CUSTOM RESOURCE DEFINITIONS	97
6.1.1. Custom Resource Definitions	97
6.1.2. Creating a Custom Resource Definition	97
6.1.3. Creating cluster roles for Custom Resource Definitions	99
6.1.4. Creating Custom Resources from a file	100
6.1.5. Inspecting Custom Resources	101
6.2. MANAGING RESOURCES FROM CUSTOM RESOURCE DEFINITIONS	102
6.2.1. Custom Resource Definitions	102
6.2.2. Creating Custom Resources from a file	102
6.2.3. Inspecting Custom Resources	103
CHAPTER 7. QUOTAS	105
7.1. RESOURCE QUOTAS PER PROJECT	105
7.1.1. Resources managed by quotas	105
7.1.2. Quota scopes	107
7.1.3. Quota enforcement	108
7.1.4. Requests versus limits	108
7.1.5. Sample resource quota definitions	108
7.1.6. Creating a quota	112
7.1.6.1. Creating object count quotas	112
7.1.6.2. Setting resource quota for extended resources	113
7.1.7. Viewing a quota	115
7.1.8. Configuring quota synchronization period	115
7.2. RESOURCE QUOTAS ACROSS MULTIPLE PROJECTS	116
7.2.1. Selecting multiple projects during quota creation	116
7.2.2. Viewing applicable ClusterResourceQuotas	118
7.2.3. Selection granularity	118
CHAPTER 8. MONITORING APPLICATION HEALTH	119
8.1. UNDERSTANDING HEALTH CHECKS	119
8.1.1. Understanding the types of health checks	120
8.2. CONFIGURING HEALTH CHECKS	121
CHAPTER 9. IDLING APPLICATIONS	124
9.1. IDLING APPLICATIONS	124
9.1.1. Idling a single service	124
9.1.2. Idling multiple services	124
9.2. UNIDLING APPLICATIONS	124

CHAPTER 10. PRUNING OBJECTS TO RECLAIM RESOURCES	126
10.1. BASIC PRUNING OPERATIONS	126
10.2. PRUNING GROUPS	126
10.3. PRUNING DEPLOYMENTS	126
10.4. PRUNING BUILDS	127
10.5. PRUNING IMAGES	128
10.5.1. Image prune conditions	130
10.5.2. Running the image prune operation	131
10.5.3. Using secure or insecure connections	131
10.5.4. Image pruning problems	132
Images not being pruned	132
Using a secure connection against insecure registry	133
Using an insecure connection against a secured registry	133
Using the wrong certificate authority	133
10.6. HARD PRUNING THE REGISTRY	134
10.7. PRUNING CRON JOBS	136
CHAPTER 11. OPERATOR SDK	138
11.1. GETTING STARTED WITH THE OPERATOR SDK	138
11.1.1. Architecture of the Operator SDK	138
11.1.1.1. Workflow	138
11.1.1.2. Manager file	139
11.1.1.3. Prometheus Operator support	139
11.1.2. Installing the Operator SDK CLI	139
11.1.2.1. Installing from GitHub release	140
11.1.2.2. Installing from Homebrew	141
11.1.2.3. Compiling and installing from source	142
11.1.3. Building a Go-based Memcached Operator using the Operator SDK	143
11.1.4. Managing a Memcached Operator using the Operator Lifecycle Manager	148
11.1.5. Additional resources	150
11.2. CREATING ANSIBLE-BASED OPERATORS	151
11.2.1. Ansible support in the Operator SDK	151
11.2.1.1. Custom Resource files	151
11.2.1.2. Watches file	152
11.2.1.2.1. Advanced options	153
11.2.1.3. Extra variables sent to Ansible	154
11.2.1.4. Ansible Runner directory	155
11.2.2. Installing the Operator SDK CLI	155
11.2.2.1. Installing from GitHub release	155
11.2.2.2. Installing from Homebrew	157
11.2.2.3. Compiling and installing from source	158
11.2.3. Building an Ansible-based Operator using the Operator SDK	158
11.2.4. Managing application lifecycle using the k8s Ansible module	164
11.2.4.1. Installing the k8s Ansible module	164
11.2.4.2. Testing the k8s Ansible module locally	165
11.2.4.3. Testing the k8s Ansible module inside an Operator	166
11.2.4.3.1. Testing an Ansible-based Operator locally	167
11.2.4.3.2. Testing an Ansible-based Operator on a cluster	168
11.2.5. Managing Custom Resource status using the k8s_status Ansible module	169
11.2.5.1. Using the k8s_status Ansible module when testing locally	170
11.2.6. Additional resources	170
11.3. CREATING HELM-BASED OPERATORS	170
11.3.1. Helm chart support in the Operator SDK	171

11.3.2. Installing the Operator SDK CLI	171
11.3.2.1. Installing from GitHub release	172
11.3.2.2. Installing from Homebrew	173
11.3.2.3. Compiling and installing from source	174
11.3.3. Building a Helm-based Operator using the Operator SDK	175
11.3.4. Additional resources	180
11.4. GENERATING A CLUSTERSERVICEVERSION (CSV)	180
11.4.1. How CSV generation works	180
Workflow	181
11.4.2. CSV composition configuration	181
11.4.3. Manually-defined CSV fields	182
11.4.4. Generating a CSV	183
11.4.5. Understanding your Custom Resource Definitions (CRDs)	184
11.4.5.1. Owned CRDs	184
11.4.5.2. Required CRDs	186
11.4.5.3. CRD templates	187
11.4.6. Understanding your API services	188
11.4.6.1. Owned APIServices	188
11.4.6.1.1. APIService Resource Creation	189
11.4.6.1.2. APIService Serving Certs	189
11.4.6.2. Required APIServices	189
11.5. CONFIGURING BUILT-IN MONITORING WITH PROMETHEUS	190
11.5.1. Prometheus Operator support	190
11.5.2. Metrics helper	190
11.5.2.1. Modifying the metrics port	191
11.5.3. ServiceMonitor resources	191
11.5.3.1. Creating ServiceMonitor resources	191
11.6. CONFIGURING LEADER ELECTION	192
11.6.1. Using Leader-for-life election	193
11.6.2. Using Leader-with-lease election	193
11.7. OPERATOR SDK CLI REFERENCE	194
11.7.1. build	194
11.7.2. completion	195
11.7.3. print-deps	195
11.7.4. generate	196
11.7.5. olm-catalog	196
11.7.5.1. gen-csv	197
11.7.6. new	197
11.7.7. add	198
11.7.8. test	200
11.7.8.1. local	200
11.7.9. up	201
11.7.9.1. local	201
11.8. APPENDICES	202
11.8.1. Operator project scaffolding layout	202
11.8.1.1. Go-based projects	202
11.8.1.2. Helm-based projects	203

CHAPTER 1. PROJECTS

1.1. WORKING WITH PROJECTS

A *project* allows a community of users to organize and manage their content in isolation from other communities.



NOTE

Projects starting with **openshift-** and **kube-** are [default projects](#). These projects host cluster components that run as Pods and other infrastructure components. As such, OpenShift Container Platform does not allow you to create Projects starting with **openshift-** or **kube-** using the **oc new-project** command. Cluster administrators can create these Projects using the [oc adm new-project command](#).

1.1.1. Creating a project using the web console

If allowed by your cluster administrator, you can create a new project.



NOTE

Projects starting with **openshift-** and **kube-** are considered critical by OpenShift Container Platform. As such, OpenShift Container Platform does not allow you to create Projects starting with **openshift-** using the web console.

Procedure

1. Navigate to **Home** → **Projects**.
2. Click **Create Project**.
3. Enter your project details.
4. Click **Create**.

1.1.2. Creating a project using the CLI

If allowed by your cluster administrator, you can create a new project.



NOTE

Projects starting with **openshift-** and **kube-** are considered critical by OpenShift Container Platform. As such, OpenShift Container Platform does not allow you to create Projects starting with **openshift-** or **kube-** using the **oc new-project** command. Cluster administrators can create these Projects using the [oc adm new-project command](#).

Procedure

1. Run:

```
$ oc new-project <project_name> \  
--description="<description>" --display-name="<display_name>"
```

For example:

```
$ oc new-project hello-openshift \
  --description="This is an example project" \
  --display-name="Hello OpenShift"
```



NOTE

The number of projects you are allowed to create may be limited by the system administrator. After your limit is reached, you might have to delete an existing project in order to create a new one.

1.1.3. Viewing a project using the web console

Procedure

1. Navigate to **Home** → **Projects**.
2. Select a project to view.
On this page, click the **Resources** button to see workloads in the project and click the **Dashboard** button to see metrics and details about the project.

1.1.4. Viewing a project using the CLI

When viewing projects, you are restricted to seeing only the projects you have access to view based on the authorization policy.

Procedure

1. To view a list of projects, run:

```
$ oc get projects
```

2. You can change from the current project to a different project for CLI operations. The specified project is then used in all subsequent operations that manipulate project-scoped content:

```
$ oc project <project_name>
```

1.1.5. Adding to a project

Procedure

1. Navigate to **Home** → **Projects**.
2. Select a project.
3. In the upper right-hand corner of the **Project Status** menu, click **Add**, then choose from the provided options.

1.1.6. Checking project status using the web console

Procedure

1. Navigate to **Home** → **Projects**.
2. Select a project to see its status.

1.1.7. Checking project status using the CLI

Procedure

1. Run:

```
$ oc status
```

This command provides a high-level overview of the current project, with its components and their relationships.

1.1.8. Deleting a project using the web console

Procedure

1. Navigate to **Home** → **Projects**.
2. Locate the project that you want to delete from the list of projects.
3. On the far right side of the project listing, select **Delete Project** from the menu. If you do not have permissions to delete the project, the **Delete Project** option is grayed out and the option is not clickable.

1.1.9. Deleting a project using the CLI

When you delete a project, the server updates the project status to **Terminating** from **Active**. Then, the server clears all content from a project that is in the **Terminating** state before finally removing the project. While a project is in **Terminating** status, you cannot add new content to the project. Projects can be deleted from the CLI or the web console.

Procedure

1. Run:

```
$ oc delete project <project_name>
```

1.2. CREATING A PROJECT AS ANOTHER USER

Impersonation allows you to create a project as a different user.

1.2.1. API impersonation

You can configure a request to the OpenShift Container Platform API to act as though it originated from another user. For more information, see [User impersonation](#) in the Kubernetes documentation.

1.2.2. Impersonating a user when you create a project

You can impersonate a different user when you create a project request. Because **system:authenticated:oauth** is the only bootstrap group that can create project requests, you must impersonate that group.

Procedure

- To create a project request on behalf of a different user:

```
$ oc new-project <project> --as=<user> \
  --as-group=system:authenticated --as-group=system:authenticated:oauth
```

1.3. CONFIGURING PROJECT CREATION

In OpenShift Container Platform, *projects* are used to group and isolate related objects. When a request is made to create a new project using the web console or **oc new-project** command, an endpoint in OpenShift Container Platform is used to provision the project according to a template, which can be customized.

As a cluster administrator, you can allow and configure how developers and service accounts can create, or *self-provision*, their own projects.

1.3.1. About project creation

The OpenShift Container Platform API server automatically provisions new projects based on the project template that is identified by the **projectRequestTemplate** parameter in the cluster's project configuration resource. If the parameter is not defined, the API server creates a default template that creates a project with the requested name, and assigns the requesting user to the **admin** role for that project.

When a project request is submitted, the API substitutes the following parameters into the template:

Table 1.1. Default project template parameters

Parameter	Description
PROJECT_NAME	The name of the project. Required.
PROJECT_DISPLAYNAME	The display name of the project. May be empty.
PROJECT_DESCRIPTION	The description of the project. May be empty.
PROJECT_ADMIN_USER	The user name of the administrating user.
PROJECT_REQUESTING_USER	The user name of the requesting user.

Access to the API is granted to developers with the **self-provisioner** role and the **self-provisioners** cluster role binding. This role is available to all authenticated developers by default.

1.3.2. Modifying the template for new projects

As a cluster administrator, you can modify the default project template so that new projects are created using your custom requirements.

To create your own custom project template:

Procedure

1. Log in as a user with **cluster-admin** privileges.
2. Generate the default project template:

```
$ oc adm create-bootstrap-project-template -o yaml > template.yaml
```

3. Use a text editor to modify the generated **template.yaml** file by adding objects or modifying existing objects.
4. The project template must be created in the **openshift-config** namespace. Load your modified template:

```
$ oc create -f template.yaml -n openshift-config
```

5. Edit the project configuration resource using the web console or CLI.
 - Using the web console:
 - i. Navigate to the **Administration** → **Cluster Settings** page.
 - ii. Click **Global Configuration** to view all configuration resources.
 - iii. Find the entry for **Project** and click **Edit YAML**.
 - Using the CLI:
 - i. Edit the **project.config.openshift.io/cluster** resource:

```
$ oc edit project.config.openshift.io/cluster
```

6. Update the **spec** section to include the **projectRequestTemplate** and **name** parameters, and set the name of your uploaded project template. The default name is **project-request**.

Project configuration resource with custom project template

```
apiVersion: config.openshift.io/v1
kind: Project
metadata:
  ...
spec:
  projectRequestTemplate:
    name: <template_name>
```

7. After you save your changes, create a new project to verify that your changes were successfully applied.

1.3.3. Disabling project self-provisioning

You can prevent an authenticated user group from self-provisioning new projects.

Procedure

1. Log in as a user with **cluster-admin** privileges.
2. View the **self-provisioners** cluster role binding usage by running the following command:

```
$ oc describe clusterrolebinding.rbac self-provisioners

Name: self-provisioners
Labels: <none>
Annotations: rbac.authorization.kubernetes.io/autoupdate=true
Role:
  Kind: ClusterRole
  Name: self-provisioner
Subjects:
  Kind Name   Namespace
  ----
  Group system:authenticated:oauth
```

Review the subjects in the **self-provisioners** section.

3. Remove the **self-provisioner** cluster role from the group **system:authenticated:oauth**.
 - If the **self-provisioners** cluster role binding binds only the **self-provisioner** role to the **system:authenticated:oauth** group, run the following command:

```
$ oc patch clusterrolebinding.rbac self-provisioners -p '{"subjects": null}'
```

- If the **self-provisioners** cluster role binding binds the **self-provisioner** role to more users, groups, or service accounts than the **system:authenticated:oauth** group, run the following command:

```
$ oc adm policy \
  remove-cluster-role-from-group self-provisioner \
  system:authenticated:oauth
```

4. Edit the **self-provisioners** cluster role binding to prevent automatic updates to the role. Automatic updates reset the cluster roles to the default state.

- To update the role binding using the CLI:
 - i. Run the following command:

```
$ oc edit clusterrolebinding.rbac self-provisioners
```

- ii. In the displayed role binding, set the **rbac.authorization.kubernetes.io/autoupdate** parameter value to **false**, as shown in the following example:

```
apiVersion: authorization.openshift.io/v1
kind: ClusterRoleBinding
metadata:
```

```

annotations:
  rbac.authorization.kubernetes.io/autoupdate: "false"
...

```

- To update the role binding by using a single command:

```
$ oc patch clusterrolebinding.rbac self-provisioners -p '{"metadata": {"annotations": {"rbac.authorization.kubernetes.io/autoupdate": "false"} } }'
```

5. Login as an authenticated user and verify that it can no longer self-provision a project:

```
$ oc new-project test
```

Error from server (Forbidden): You may not request a new project via this API.

Consider customizing this project request message to provide more helpful instructions specific to your organization.

1.3.4. Customizing the project request message

When a developer or a service account that is unable to self-provision projects makes a project creation request using the web console or CLI, the following error message is returned by default:

You may not request a new project via this API.

Cluster administrators can customize this message. Consider updating it to provide further instructions on how to request a new project specific to your organization. For example:

- To request a project, contact your system administrator at **projectname@example.com**.
- To request a new project, fill out the project request form located at **<https://internal.example.com/openshift-project-request>**.

To customize the project request message:

Procedure

1. Edit the project configuration resource using the web console or CLI.
 - Using the web console:
 - i. Navigate to the **Administration** → **Cluster Settings** page.
 - ii. Click **Global Configuration** to view all configuration resources.
 - iii. Find the entry for **Project** and click **Edit YAML**.
 - Using the CLI:
 - i. Log in as a user with **cluster-admin** privileges.
 - ii. Edit the **project.config.openshift.io/cluster** resource:

```
$ oc edit project.config.openshift.io/cluster
```

2. Update the **spec** section to include the **projectRequestMessage** parameter and set the value to your custom message:

Project configuration resource with custom project request message

```
apiVersion: config.openshift.io/v1
kind: Project
metadata:
  ...
spec:
  projectRequestMessage: <message_string>
```

For example:

```
apiVersion: config.openshift.io/v1
kind: Project
metadata:
  ...
spec:
  projectRequestMessage: To request a project, contact your system administrator at
  projectname@example.com.
```

3. After you save your changes, attempt to create a new project as a developer or service account that is unable to self-provision projects to verify that your changes were successfully applied.

CHAPTER 2. OPERATORS

2.1. UNDERSTANDING OPERATORS

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

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

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

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

2.1.1. Why use Operators?

Operators provide:

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

Why deploy on Kubernetes?

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

Why manage your app with Kubernetes APIs and **kubectrl** tooling?

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

How do Operators compare with Service Brokers?

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

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

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

Controls the installation, upgrade, and role-based access control (RBAC) of Operators in a cluster. Deployed by default in OpenShift Container Platform 4.1.

Operator Metering

Collects operational metrics about Operators on the cluster for Day 2 management and aggregating usage metrics.

OperatorHub

Web console for discovering and installing Operators on your cluster. Deployed by default in OpenShift Container Platform 4.1.

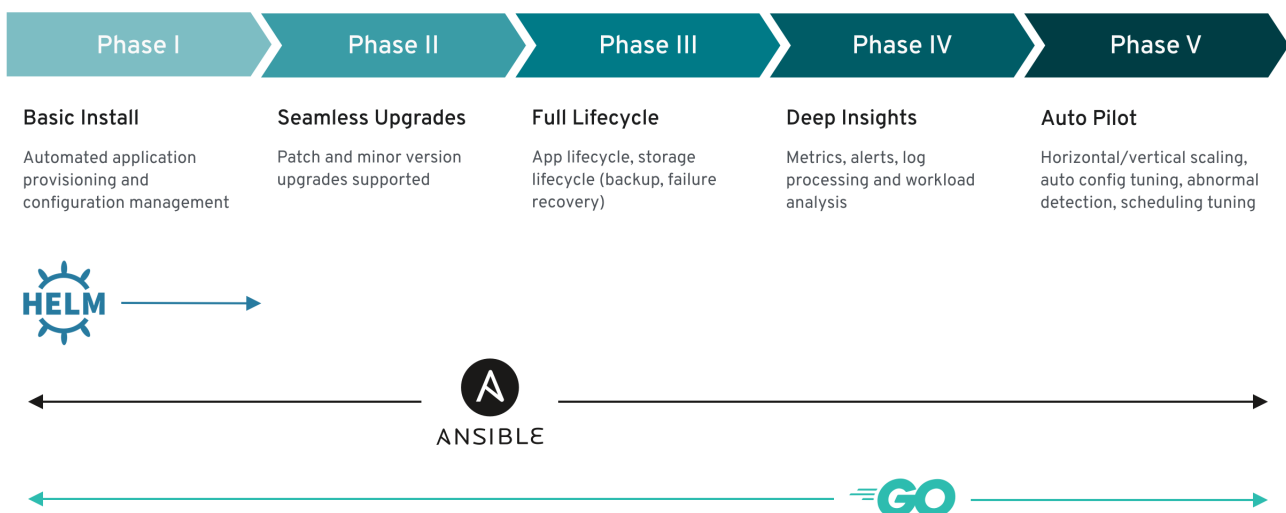
These tools are designed to be composable, so you can use any that are useful to you.

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

Figure 2.1. Operator maturity model



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

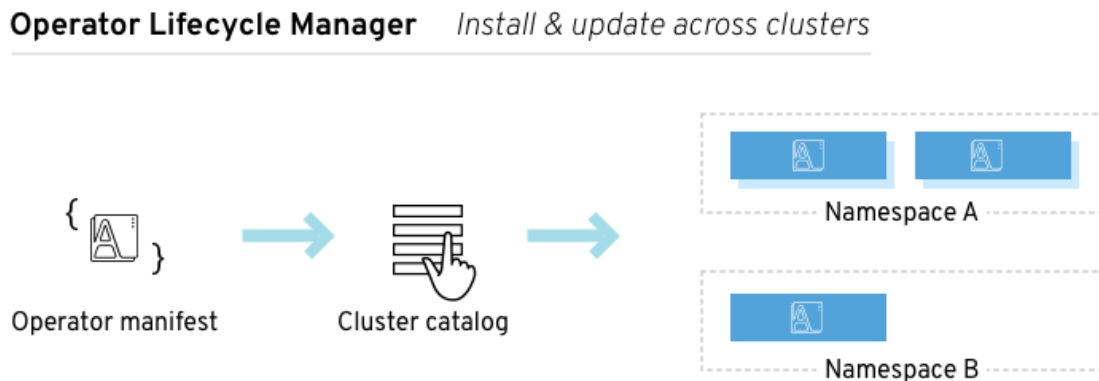
2.2. UNDERSTANDING THE OPERATOR LIFECYCLE MANAGER

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

2.2.1. Overview of the Operator Lifecycle Manager

In OpenShift Container Platform 4.1, the *Operator Lifecycle Manager* (OLM) helps users install, update, and manage the lifecycle of all Operators and their associated services running across their clusters. It is part of the [Operator Framework](#), an open source toolkit designed to manage Kubernetes native applications (Operators) in an effective, automated, and scalable way.

Figure 2.2. Operator Lifecycle Manager workflow



The OLM runs by default in OpenShift Container Platform 4.1, 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.

2.2.2. ClusterServiceVersions (CSVs)

A *ClusterServiceVersion* (CSV) is a YAML manifest created from Operator metadata that assists the Operator Lifecycle Manager (OLM) in running the Operator in a cluster. It is the metadata that accompanies an Operator container image, used to populate user interfaces with information like its logo, description, and version. It is also a source of technical information needed to run the Operator, like the RBAC rules it requires and which Custom Resources (CRs) it manages or depends on.

A CSV is composed of:

Metadata

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

Install strategy

- Type: Deployment

- Set of service accounts and required permissions
- Set of Deployments.

Custom Resource Definitions (CRDs)

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

2.2.3. Operator Lifecycle Manager architecture

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

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

Table 2.1. CRDs managed by OLM and Catalog Operators

Resource	Short name	Owner	Description
ClusterServiceVersion	csv	OLM	Application metadata: name, version, icon, required resources, installation, etc.
InstallPlan	ip	Catalog	Calculated list of resources to be created in order to automatically install or upgrade a CSV.
CatalogSource	catalog	Catalog	A repository of CSVs, CRDs, and packages that define an application.
Subscription	sub	Catalog	Keeps CSVs up to date by tracking a channel in a package.
OperatorGroup	og	OLM	Configures all Operators deployed in the same namespace as the OperatorGroup object to watch for their Custom Resource (CR) in a list of namespaces or cluster-wide.

Each of these Operators are also responsible for creating resources:

Table 2.2. Resources created by OLM and Catalog Operators

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

2.2.3.1. OLM Operator

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

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

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

OLM Operator workflow

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



NOTE

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

2.2.3.2. Catalog Operator

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

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

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

Catalog Operator workflow

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

2.2.3.3. Catalog Registry

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

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

2.2.4. OperatorGroups

An *OperatorGroup* is an OLM resource that provides multitenant configuration to OLM-installed Operators. An OperatorGroup selects a set of target namespaces in which to generate required RBAC access for its member Operators. The set of target namespaces is provided by a comma-delimited string stored in the CSV's **olm.targetNamespaces** annotation. This annotation is applied to member Operator's CSV instances and is projected into their deployments.

2.2.4.1. OperatorGroup membership

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

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

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

Table 2.3. InstallModes and supported OperatorGroups

InstallModeType	Description
OwnNamespace	The Operator can be a member of an OperatorGroup that selects its own namespace.

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

**NOTE**

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

2.2.4.1.1. Troubleshooting OperatorGroup membership

- If more than one OperatorGroup exists in a single namespace, any CSV created in that namespace will transition to a failure state with the reason **TooManyOperatorGroups**. CSVs in a failed state for this reason will transition to pending once the number of OperatorGroups in their namespaces reaches one.
- If a CSV's InstallModes do not support the target namespace selection of the OperatorGroup in its namespace, the CSV will transition to a failure state with the reason **UnsupportedOperatorGroup**. CSVs in a failed state for this reason will transition to pending once either the OperatorGroup's target namespace selection changes to a supported configuration, or the CSV's InstallModes are modified to support the OperatorGroup's target namespace selection.

2.2.4.2. Target namespace selection

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

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

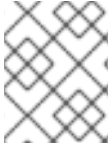
You can also explicitly name the target namespaces using the **spec.targetNamespaces** field:

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

```

name: my-group
namespace: my-namespace
spec:
  targetNamespaces:
  - my-namespace
  - my-other-namespace
  - my-other-other-namespace

```

**NOTE**

If both **spec.targetNamespaces** and **spec.selector** are defined, **spec.selector** is ignored.

Alternatively, you can omit both **spec.selector** and **spec.targetNamespaces** to specify a *global* OperatorGroup, which selects all namespaces:

```

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

```

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

2.2.4.3. OperatorGroup CSV annotations

Member CSVs of an OperatorGroup have the following annotations:

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

**NOTE**

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

2.2.4.4. Provided APIs annotation

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

CSVs of an OperatorGroup are included.

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

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

2.2.4.5. Role-based access control

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

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

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

- [ClusterRoles for each API resource from a CRD](#)
- [ClusterRoles for each API resource from an APIService](#)
- [Additional Roles and RoleBindings](#)

Table 2.4. ClusterRoles generated for each API resource from a CRD

ClusterRole	Settings
<kind>.<group>-<version>-admin	Verbs on <kind> : <ul style="list-style-type: none"> ● * Aggregation labels: <ul style="list-style-type: none"> ● rbac.authorization.k8s.io/aggregate-to-admin: true ● olm.opgroup.permissions/aggregate-to-admin: <operatorgroup_name>
<kind>.<group>-<version>-edit	Verbs on <kind> : <ul style="list-style-type: none"> ● create ● update ● patch ● delete Aggregation labels: <ul style="list-style-type: none"> ● rbac.authorization.k8s.io/aggregate-to-edit: true ● olm.opgroup.permissions/aggregate-to-edit: <operatorgroup_name>
<kind>.<group>-<version>-view	Verbs on <kind> : <ul style="list-style-type: none"> ● get ● list ● watch Aggregation labels: <ul style="list-style-type: none"> ● rbac.authorization.k8s.io/aggregate-to-view: true ● olm.opgroup.permissions/aggregate-to-view: <operatorgroup_name>

ClusterRole	Settings
<kind>.<group>-<version>-view-crdview	<p>Verbs on apiextensions.k8s.io customresourcedefinitions <crd-name>:</p> <ul style="list-style-type: none"> ● get <p>Aggregation labels:</p> <ul style="list-style-type: none"> ● rbac.authorization.k8s.io/aggregate-to-view: true ● olm.opgroup.permissions/aggregate-to-view: <operatorgroup_name>

Table 2.5. ClusterRoles generated for each API resource from an APIService

ClusterRole	Settings
<kind>.<group>-<version>-admin	<p>Verbs on <kind>:</p> <ul style="list-style-type: none"> ● * <p>Aggregation labels:</p> <ul style="list-style-type: none"> ● rbac.authorization.k8s.io/aggregate-to-admin: true ● olm.opgroup.permissions/aggregate-to-admin: <operatorgroup_name>
<kind>.<group>-<version>-edit	<p>Verbs on <kind>:</p> <ul style="list-style-type: none"> ● create ● update ● patch ● delete <p>Aggregation labels:</p> <ul style="list-style-type: none"> ● rbac.authorization.k8s.io/aggregate-to-edit: true ● olm.opgroup.permissions/aggregate-to-edit: <operatorgroup_name>

ClusterRole	Settings
<code><kind>.<group>-<version>-view</code>	Verbs on <code><kind></code> : <ul style="list-style-type: none"> • <code>get</code> • <code>list</code> • <code>watch</code> Aggregation labels: <ul style="list-style-type: none"> • <code>rbac.authorization.k8s.io/aggregate-to-view: true</code> • <code>olm.opgroup.permissions/aggregate-to-view: <operatorgroup_name></code>

Additional Roles and RoleBindings

- If the CSV defines exactly one target namespace that contains `*`, then a ClusterRole and corresponding ClusterRoleBinding are generated for each permission defined in the CSV's permissions field. All resources generated are given the `olm.owner: <csv_name>` and `olm.owner.namespace: <csv_namespace>` labels.
- If the CSV does *not* define exactly one target namespace that contains `*`, then all Roles and RoleBindings in the Operator namespace with the `olm.owner: <csv_name>` and `olm.owner.namespace: <csv_namespace>` labels are copied into the target namespace.

2.2.4.6. Copied CSVs

OLM creates copies of all active member CSVs of an OperatorGroup in each of that OperatorGroup's target namespaces. The purpose of a copied CSV is to tell users of a target namespace that a specific Operator is configured to watch resources created there. Copied CSVs have a status reason **Copied** and are updated to match the status of their source CSV. The `olm.targetNamespaces` annotation is stripped from copied CSVs before they are created on the cluster. Omitting the target namespace selection avoids the duplication of target namespaces between tenants. Copied CSVs are deleted when their source CSV no longer exists or the OperatorGroup that their source CSV belongs to no longer targets the copied CSV's namespace.

2.2.4.7. Static OperatorGroups

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

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

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

```

annotations:
  olm.providedAPIs:
Alertmanager.v1.monitoring.coreos.com,Prometheus.v1.monitoring.coreos.com,PrometheusRule.v1.mo
nitoring.coreos.com,ServiceMonitor.v1.monitoring.coreos.com
spec:
  staticProvidedAPIs: true
  selector:
    matchLabels:
      something.cool.io/cluster-monitoring: "true"

```

2.2.4.8. OperatorGroup intersection

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

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



NOTE

When checking intersection rules, an OperatorGroup's namespace is always included as part of its selected target namespaces.

2.2.4.8.1. Rules for intersection

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

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

- Clean up any deployments that belong to the CSV.
- Transition the CSV to a failed state with status reason **CannotModifyStaticOperatorGroupProvidedAPIs**.
- If the OperatorGroup is *not* static:
 - Replace the OperatorGroup's **olm.providedAPIs** annotation with the difference between itself and the CSV's provided APIs.

**NOTE**

Failure states caused by OperatorGroups are non-terminal.

The following actions are performed each time an OperatorGroup synchronizes:

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

2.2.5. Metrics

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

Table 2.6. Metrics exposed by OLM

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

2.3. UNDERSTANDING THE OPERATORHUB

This guide outlines the architecture of the OperatorHub.

2.3.1. Overview of the OperatorHub

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

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

Category	Description
Red Hat Operators	Red Hat products packaged and shipped by Red Hat. Supported by Red Hat.
Certified Operators	Products from leading independent software vendors (ISVs). Red Hat partners with ISVs to package and ship. Supported by the ISV.
Community Operators	Optionally-visible software maintained by relevant representatives in the operator-framework/community-operators GitHub repository. No official support.
Custom Operators	Operators you add to the cluster yourself. If you have not added any Custom Operators, the Custom category does not appear in the Web console on your OperatorHub.

The OperatorHub component is installed and run as an Operator by default on OpenShift Container Platform in the **openshift-marketplace** namespace.

2.3.2. OperatorHub architecture

The OperatorHub component's Operator manages two Custom Resource Definitions (CRDs): an [OperatorSource](#) and a [CatalogSourceConfig](#).



NOTE

Although some OperatorSource and CatalogSourceConfig information is exposed through the OperatorHub user interface, those files are only used directly by those who are creating their own Operators.

2.3.2.1. OperatorSource

For each Operator, the OperatorSource is used to define the external data store used to store Operator bundles. A [simple OperatorSource](#) includes:

Field	Description
type	To identify the data store as an application registry, type is set to appregistry .
endpoint	Currently, Quay is the external data store used by the OperatorHub, so the endpoint is set to https://quay.io/cnr for the Quay.io appregistry .
registryNamespace	For a Community Operator, this is set to community-operator .

Field	Description
displayName	Optionally set to a name that appears in the OperatorHub user interface for the Operator.
publisher	Optionally set to the person or organization publishing the Operator, so it can be displayed on the OperatorHub.

2.3.2.2. CatalogSourceConfig

An Operator's CatalogSourceConfig is used to enable an Operator present in the OperatorSource on the cluster.

A [simple CatalogSourceConfig](#) must identify:

Field	Description
targetNamespace	The location where the Operator would be deployed and updated, such as openshift-operators . This is a namespace that the OLM watches.
packages	A comma-separated list of packages that make up the content of the Operator.

2.4. ADDING OPERATORS TO A CLUSTER

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

2.4.1. Installing Operators from the OperatorHub

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

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

Installation Mode

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

Update Channel

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

Approval Strategy

You can choose Automatic or Manual updates. If you choose Automatic updates for an installed Operator, when a new version of that Operator is available, the Operator Lifecycle Manager (OLM) automatically upgrades the running instance of your Operator without human intervention. If you

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

2.4.1.1. Installing from the OperatorHub using the web console

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

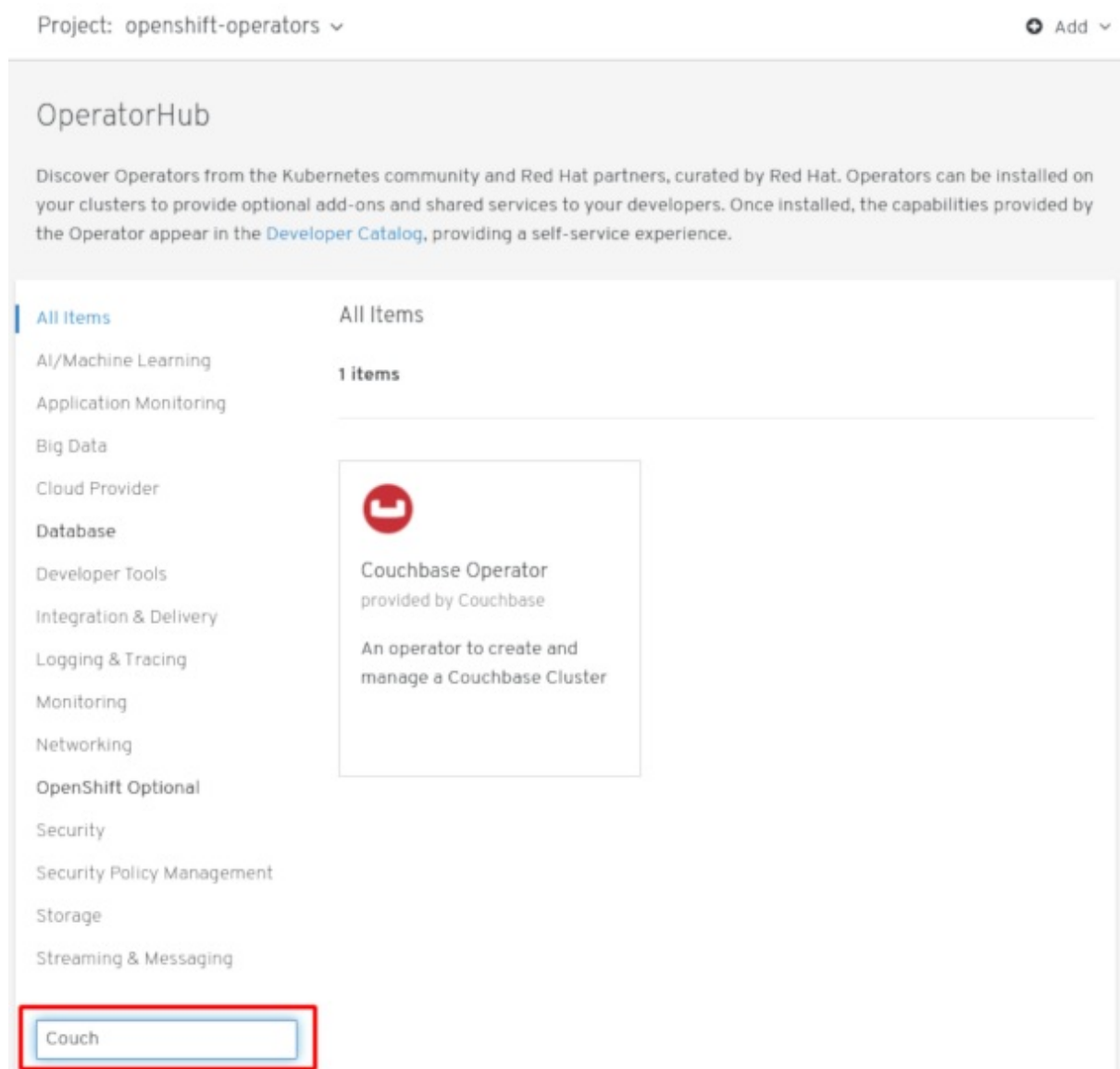
Prerequisites

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

Procedure

1. Navigate in the web console to the **Catalog → OperatorHub** page.
2. Scroll or type a keyword into the **Filter by keyword** box (in this case, **Couchbase**) to find the Operator you want.

Figure 2.3. Filter Operators by keyword



3. Select the Operator. For a Community Operator, you are warned that Red Hat does not certify those Operators. You must acknowledge that warning before continuing. Information about the Operator is displayed.
4. Read the information about the Operator and click **Install**.
5. On the **Create Operator Subscription** page:
 - a. Select one of the following:
 - **All namespaces on the cluster (default)** installs the Operator in the default **openshift-operators** namespace to watch and be made available to all namespaces in the cluster. This option is not always available.
 - **A specific namespace on the cluster** allows you to choose a specific, single namespace in which to install the Operator. The Operator will only watch and be made available for use in this single namespace.
 - b. Select an **Update Channel** (if more than one is available).
 - c. Select **Automatic** or **Manual** approval strategy, as described earlier.
6. Click **Subscribe** to make the Operator available to the selected namespaces on this OpenShift Container Platform cluster.
7. From the **Catalog → Operator Management** page, you can monitor an Operator Subscription's installation and upgrade progress.
 - a. If you selected a Manual approval strategy, the Subscription's upgrade status will remain **Upgrading** until you review and approve its Install Plan.

Figure 2.4. Manually approving from the Install Plan page

Project: openshift-operators ⌵ ➕ Add ⌵

[couchbase-enterprise-certified](#) > Install Plan Details

IP install-bqbms Actions ⌵

[Overview](#) [YAML](#) [Components](#)

Review Manual Install Plan

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

[Preview Install Plan](#)

Install Plan Overview

<p>NAME install-bqbms</p> <p>NAMESPACE NS openshift-operators</p> <p>LABELS No labels</p> <p>CREATED AT 🕒 a few seconds ago</p> <p>1 OWNER SUB couchbase-enterprise-certified</p>	<p>STATUS RequiresApproval</p> <p>COMPONENTS CSV couchbase-operator.v1.1.0</p> <p>CATALOG SOURCES CS installed-certified-openshift-operators</p>
--	---

After approving on the **Install Plan** page, the Subscription upgrade status moves to **Up to date**.

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

Figure 2.5. Subscription upgrade status Up to date

Project: openshift-operators ▾

SUB couchbase-enterprise-certified

Overview **YAML**

Subscription Overview

CHANNEL preview ✎	APPROVAL Automatic ✎	UPGRADE STATUS ✔ Up to date	1 installed 0 installing
----------------------	-------------------------	---------------------------------------	-----------------------------

NAME
couchbase-enterprise-certified

INSTALLED VERSION
CSV couchbase-operator.v1.1.0

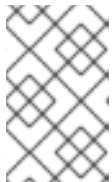
NAMESPACE
NS openshift-operators

STARTING VERSION
couchbase-operator.v1.1.0

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

CATALOG SOURCE
CS installed-certified-openshift-operators

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



NOTE

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

If it does not:

- Switch to the **Catalog → Operator Management** page and inspect the **Operator Subscriptions** and **Install Plans** tabs for any failure or errors under **Status**.
- Check the logs in any Pods in the **openshift-operators** project (or other relevant namespace if **A specific namespace...** Installation Mode was selected) on the **Workloads → Pods** page that are reporting issues to troubleshoot further.

2.4.1.2. Installing from the OperatorHub using the CLI

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

**NOTE**

The web console version of this procedure handles the creation of the CatalogSourceConfig and Subscription objects behind the scenes for you, appearing as if it was one step.

Prerequisites

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

Procedure

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

```
$ oc get packagemanifests -n openshift-marketplace
NAME                AGE
amq-streams         14h
packageserver       15h
couchbase-enterprise 14h
mongodb-enterprise  14h
etcd                 14h
myoperator           14h
...
```

2. To identify the Operators to enable on the cluster, create a CatalogSourceConfig object YAML file (for example, **csc.cr.yaml**). Include one or more packages listed in the previous step (such as couchbase-enterprise or etcd). For example:

Example CatalogSourceConfig

```
apiVersion: operators.coreos.com/v1
kind: CatalogSourceConfig
metadata:
  name: example
  namespace: openshift-marketplace
spec:
  targetNamespace: openshift-operators 1
  packages: myoperator 2
```

- 1** Set the **targetNamespace** to identify the namespace where you want the Operator to be available. The **openshift-operators** namespace is watched by the Operator Lifecycle Manager (OLM).
- 2** Set **packages** to a comma-separated list of Operators to which you want to subscribe.

The Operator generates a CatalogSource from your CatalogSourceConfig in the namespace specified in **targetNamespace**.

3. Create the CatalogSourceConfig to enable the specified Operators in the selected namespace:

```
$ oc apply -f csc.cr.yaml
```


4. Create a Subscription object YAML file (for example, **myoperator-sub.yaml**) to subscribe a namespace to an Operator. Note that the namespace you pick must have an OperatorGroup that matches the installMode (either AllNamespaces or SingleNamespace modes):

Example Subscription

```
apiVersion: operators.coreos.com/v1alpha1
kind: Subscription
metadata:
  name: myoperator
  namespace: openshift-operators
spec:
  channel: alpha
  name: myoperator 1
  source: example 2
  sourceNamespace: openshift-operators
```

- 1** Name of the Operator to subscribe to.
- 2** Name of the CatalogSource that was created.

5. Create the Subscription object:

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

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

6. Later, if you want to install more Operators:
 - a. Update your CatalogSourceConfig file (in this example, **csc.cr.yaml**) with more packages. For example:

Example updated CatalogSourceConfig

```
apiVersion: operators.coreos.com/v1
kind: CatalogSourceConfig
metadata:
  name: example
  namespace: openshift-marketplace
spec:
  targetNamespace: global
  packages: myoperator,another-operator 1
```

- 1** Add new packages to existing package list.

- b. Update the CatalogSourceConfig object:

```
$ oc apply -f csc.cr.yaml
```

- c. Create additional Subscription objects for the new Operators.

Additional resources

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

2.5. DELETING OPERATORS FROM A CLUSTER


To delete (uninstall) an Operator from your cluster, you can simply delete the subscription to remove it from the subscribed namespace. If you want a clean slate, you can also remove the operator CSV and deployment, then delete Operator's entry in the CatalogSourceConfig. The following text describes how to delete Operators from a cluster using either the web console or the command line.

2.5.1. Deleting Operators from a cluster using the web console

To delete an installed Operator from the selected namespace through the web console, follow these steps:

Procedure

- Select the Operator to delete. There are two paths to do this:
 - From the **Catalog** → **OperatorHub** page:
 - Scroll or type a keyword into the **Filter by keyword box** (in this case, **jaeger**) to find the Operator you want and click on it.
 - Click **Uninstall**.
 - From the **Catalog** → **Operator Management** page:
 - Select the namespace where the Operator is installed from the **Project** list. For cluster-wide Operators, the default is **openshift-operators**.
 - From the **Operator Subscriptions** tab, find the Operator you want to delete (in this

example, **jaeger**) and click the Options menu  at the end of its entry.

Project: openshift-operators ▼ Add ▼

Operator Management

Operator Catalogs **Operator Subscriptions** Install Plans

[Create Subscription](#) Filter Subscriptions by package...

NAME ↑	NAMESPACE	STATUS	CHANNEL
SUB jaeger	NS openshift-operators	✔ Up to date	alpha ⋮

Edit Subscription
 Remove Subscription...
 View ClusterServiceVersion...

3. Click **Remove Subscription**.
2. When prompted by the **Remove Subscription** window, optionally select the **Also completely remove the jaeger Operator from the selected namespace** check box if you want all components related to the installation to be removed. This removes the CSV, which in turn removes the Pods, Deployments, CRDs, and CRs associated with the Operator.
3. Select **Remove**. This Operator will stop running and no longer receive updates.



NOTE

Although the Operator is no longer installed or receiving updates, that Operator will still appear on the Operator Catalogs list, ready to re-subscribe. To remove the Operator from that listing, you can delete the Operator's entry in the CatalogSourceConfig from the command line (as shown in last step of "Deleting operators from a cluster using the CLI").

2.5.2. Deleting Operators from a cluster using the CLI

Instead of using the OpenShift Container Platform web console, you can delete an Operator from your cluster by using the CLI. You do this by deleting the Subscription and ClusterServiceVersion from the **targetNamespace**, then editing the CatalogSourceConfig to remove the Operator's package name.

Prerequisites

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

Procedure

In this example, there are two Operators (Jaeger and Descheduler) installed in the **openshift-operators** namespace. The goal is to remove Jaeger without removing Descheduler.

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

```
$ oc get subscription jaeger -n openshift-operators -o yaml | grep currentCSV
currentCSV: jaeger-operator.v1.8.2
```

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

```
$ oc delete subscription jaeger -n openshift-operators
subscription.operators.coreos.com "jaeger" deleted
```

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

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

4. Display the contents of the **CatalogSourceConfig** resource and review the list of packages in the **spec** section:

■

```
$ oc get catalogsourceconfig -n openshift-marketplace \
  installed-community-openshift-operators -o yaml
```

For example, the spec section might appear as follows:

Example of CatalogSourceConfig

```
spec:
  csDisplayName: Community Operators
  csPublisher: Community
  packages: jaeger,descheduler
  targetNamespace: openshift-operators
```

5. Remove the Operator from the CatalogSourceConfig in one of two ways:

- If you have multiple Operators, edit the CatalogSourceConfig resource and remove the Operator's package:

```
$ oc edit catalogsourceconfig -n openshift-marketplace \
  installed-community-openshift-operators
```

Remove the package from the **packages** line, as shown:

Example of modified packages in CatalogSourceConfig

```
packages: descheduler
```

Save the change and the **marketplace-operator** will reconcile the CatalogSourceConfig.

- If there is only one Operator in the CatalogSourceConfig, you can remove it by deleting the entire CatalogSourceConfig as follows:

```
$ oc delete catalogsourceconfig -n openshift-marketplace \
  installed-community-openshift-operators
```

2.6. CREATING APPLICATIONS FROM INSTALLED OPERATORS

This guide walks developers through an example of creating applications from an installed Operator using the OpenShift Container Platform 4.1 web console.

2.6.1. Creating an etcd cluster using an Operator

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

Prerequisites

- Access to an OpenShift Container Platform 4.1 cluster.
- The etcd Operator already installed cluster-wide by an administrator.

Procedure

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

TIP

You can get this list from the CLI using:

```
$ oc get csv
```

3. On the **Installed Operators** page, click **Copied**, and then click the etcd Operator to view more details and available actions:

Figure 2.6. etcd Operator overview

etcd
0.9.2 provided by CoreOS, Inc

Actions ▾

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

PROVIDER
CoreOS, Inc

CREATED AT
Feb 4, 3:10 pm

LINKS
Blog
<https://coreos.com/etcd>

Documentation
<https://coreos.com/operator/s/etcd/docs/latest/>

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

MAINTAINERS
CoreOS, Inc
support@coreos.com

Provided APIs

EC etcd Cluster
Represents a cluster of etcd nodes.
[Create New](#)

EB etcd Backup
Represents the intent to backup an etcd cluster.
[Create New](#)

ER etcd Restore
Represents the intent to restore an etcd cluster from a backup.
[Create New](#)

Description

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

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


4. Create a new etcd cluster:
 - a. In the **etcd Cluster** API box, click **Create New**.
 - b. The next screen allows you to make any modifications to the minimal starting template of an **EtcdCluster** object, such as the size of the cluster. For now, click **Create** to finalize. This

triggers the Operator to start up the Pods, Services, and other components of the new etcd cluster.

- Click the **Resources** tab to see that your project now contains a number of resources created and configured automatically by the Operator.

Figure 2.7. etcd Operator resources











etcdoperator.v0.9.2 > EtcdCluster Details

 example Actions ▾

Overview YAML **Resources**

Filter Resources by name...

2 Service 3 Pod Select All Filters 5 Items

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

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

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

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

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

2.7. MANAGING RESOURCES FROM CUSTOM RESOURCE DEFINITIONS

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

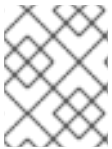
2.7.1. Custom Resource Definitions

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

A *Custom Resource Definition* (CRD) object defines a new, unique object **Kind** in the cluster and lets the Kubernetes API server handle its entire lifecycle.

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

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



NOTE

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

2.7.2. Creating Custom Resources from a file

After a Custom Resource Definition (CRD) has been added to the cluster, Custom Resources (CRs) can be created with the CLI from a file using the CR specification.

Prerequisites

- CRD added to the cluster by a cluster administrator.

Procedure

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

Example YAML file for a CR

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

- 1 Specify the group name and API version (name/version) from the Custom Resource Definition.
- 2 Specify the type in the CRD.
- 3 Specify a name for the object.

- 4 Specify the [finalizers](#) for the object, if any. Finalizers allow controllers to implement conditions that must be completed before the object can be deleted.
- 5 Specify conditions specific to the type of object.

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

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

2.7.3. Inspecting Custom Resources

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

Prerequisites

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

Procedure

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

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

For example:

```
$ oc get crontab

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

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

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

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

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

$ oc get ct -o yaml

apiVersion: v1
items:
- apiVersion: stable.example.com/v1
  kind: CronTab
  metadata:
    clusterName: ""
    creationTimestamp: 2017-05-31T12:56:35Z
    deletionGracePeriodSeconds: null
    deletionTimestamp: null
```



```
name: my-new-cron-object
namespace: default
resourceVersion: "285"
selfLink: /apis/stable.example.com/v1/namespaces/default/crontabs/my-new-cron-object
uid: 9423255b-4600-11e7-af6a-28d2447dc82b
spec:
  cronSpec: '* * * * /5' 1
  image: my-awesome-cron-image 2
```

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

CHAPTER 3. APPLICATION LIFE CYCLE MANAGEMENT

3.1. CREATING APPLICATIONS

You can create an OpenShift Container Platform application from components that include source or binary code, images, and templates by using the OpenShift Container Platform CLI.

The set of objects created by **new-app** depends on the artifacts passed as input: source repositories, images, or templates.

3.1.1. Creating an application by using the CLI

3.1.1.1. Creating an application from source code

With the **new-app** command you can create applications from source code in a local or remote Git repository.

The **new-app** command creates a build configuration, which itself creates a new application image from your source code. The **new-app** command typically also creates a deployment configuration to deploy the new image, and a service to provide load-balanced access to the deployment running your image.

OpenShift Container Platform automatically detects whether the **Pipeline** or **Source** build strategy should be used, and in the case of **Source** builds, detects an appropriate language builder image.

3.1.1.1.1. Local

To create an application from a Git repository in a local directory:

```
$ oc new-app /<path to source code>
```



NOTE

If you use a local Git repository, the repository must have a remote named **origin** that points to a URL that is accessible by the OpenShift Container Platform cluster. If there is no recognized remote, running the **new-app** command will create a binary build.

3.1.1.1.2. Remote

To create an application from a remote Git repository:

```
$ oc new-app https://github.com/sclorg/cakephp-ex
```

To create an application from a private remote Git repository:

```
$ oc new-app https://github.com/youruser/yourprivaterepo --source-secret=yoursecret
```



NOTE

If you use a private remote Git repository, you can use the **--source-secret** flag to specify an existing source clone secret that will get injected into your **BuildConfig** to access the repository.

You can use a subdirectory of your source code repository by specifying a **--context-dir** flag. To create an application from a remote Git repository and a context subdirectory:

```
$ oc new-app https://github.com/sclorg/s2i-ruby-container.git \
  --context-dir=2.0/test/puma-test-app
```

Also, when specifying a remote URL, you can specify a Git branch to use by appending **#<branch_name>** to the end of the URL:

```
$ oc new-app https://github.com/openshift/ruby-hello-world.git#beta4
```

3.1.1.1.3. Build strategy detection

If a **Jenkinsfile** exists in the root or specified context directory of the source repository when creating a new application, OpenShift Container Platform generates a Pipeline build strategy.

Otherwise, it generates a Source build strategy.

Override the build strategy by setting the **--strategy** flag to either **pipeline** or **source**.

```
$ oc new-app /home/user/code/myapp --strategy=docker
```



NOTE

The **oc** command requires that files containing build sources are available in a remote Git repository. For all source builds, you must use **git remote -v**.

3.1.1.1.4. Language Detection

If you use the **Source** build strategy, **new-app** attempts to determine the language builder to use by the presence of certain files in the root or specified context directory of the repository:

Table 3.1. Languages Detected by **new-app**

Language	Files
dotnet	project.json, *.csproj
jee	pom.xml
nodejs	app.json, package.json
perl	cpanfile, index.pl
php	composer.json, index.php
python	requirements.txt, setup.py
ruby	Gemfile, Rakefile, config.ru

Language	Files
scala	build.sbt
golang	Godeps, main.go

After a language is detected, **new-app** searches the OpenShift Container Platform server for imagestreamtags that have a **supports** annotation matching the detected language, or an imagestream that matches the name of the detected language. If a match is not found, **new-app** searches the [Docker Hub registry](#) for an image that matches the detected language based on name.

You can override the image the builder uses for a particular source repository by specifying the image, either an imagestream or container specification, and the repository with a ~ as a separator. Note that if this is done, build strategy detection and language detection are not carried out.

For example, to use the **myproject/my-ruby** imagestream with the source in a remote repository:

```
$ oc new-app myproject/my-ruby~https://github.com/openshift/ruby-hello-world.git
```

To use the `openshift/ruby-20-centos7:latest` container imagestream with the source in a local repository:

```
$ oc new-app openshift/ruby-20-centos7:latest~/home/user/code/my-ruby-app
```



NOTE

Language detection requires the Git client to be locally installed so that your repository can be cloned and inspected. If Git is not available, you can avoid the language detection step by specifying the builder image to use with your repository with the **<image>~<repository>** syntax.

The **-i <image> <repository>** invocation requires that **new-app** attempt to clone **repository** in order to determine what type of artifact it is, so this will fail if Git is not available.

The **-i <image> --code <repository>** invocation requires **new-app** clone **repository** in order to determine whether **image** should be used as a builder for the source code, or deployed separately, as in the case of a database image.

3.1.1.2. Creating an application from an image

You can deploy an application from an existing image. Images can come from imagestreams in the OpenShift Container Platform server, images in a specific registry, or images in the local Docker server.

The **new-app** command attempts to determine the type of image specified in the arguments passed to it. However, you can explicitly tell **new-app** whether the image is a container image using the **--docker-image** argument or an imagestream using the **-i|--image** argument.



NOTE

If you specify an image from your local Docker repository, you must ensure that the same image is available to the OpenShift Container Platform cluster nodes.

3.1.1.2.1. DockerHub MySQL image

Create an application from the DockerHub MySQL image, for example:

```
$ oc new-app mysql
```

3.1.1.2.2. Image in a private registry

Create an application using an image in a private registry, specify the full container image specification:

```
$ oc new-app myregistry:5000/example/myimage
```

3.1.1.2.3. Existing imagestream and optional imagestreamtag

Create an application from an existing imagestream and optional imagestreamtag:

```
$ oc new-app my-stream:v1
```

3.1.1.3. Creating an application from a template

You can create an application from a previously stored template or from a template file, by specifying the name of the template as an argument. For example, you can store a sample application template and use it to create an application.

Create an application from a stored template, for example:

```
$ oc create -f examples/sample-app/application-template-stibuild.json
$ oc new-app ruby-helloworld-sample
```

To directly use a template in your local file system, without first storing it in OpenShift Container Platform, use the **-f|--file** argument. For example:

```
$ oc new-app -f examples/sample-app/application-template-stibuild.json
```

3.1.1.3.1. Template Parameters

When creating an application based on a template, use the **-p|--param** argument to set parameter values that are defined by the template:

```
$ oc new-app ruby-helloworld-sample \
  -p ADMIN_USERNAME=admin -p ADMIN_PASSWORD=mypassword
```

You can store your parameters in a file, then use that file with **--param-file** when instantiating a template. If you want to read the parameters from standard input, use **--param-file=-**:

```
$ cat helloworld.params
ADMIN_USERNAME=admin
ADMIN_PASSWORD=mypassword
$ oc new-app ruby-helloworld-sample --param-file=helloworld.params
$ cat helloworld.params | oc new-app ruby-helloworld-sample --param-file=-
```

3.1.1.4. Modifying application creation

The **new-app** command generates OpenShift Container Platform objects that build, deploy, and run the application that is created. Normally, these objects are created in the current project and assigned names that are derived from the input source repositories or the input images. However, with **new-app** you can modify this behavior.

Table 3.2. **new-app** output objects

Object	Description
BuildConfig	A BuildConfig is created for each source repository that is specified in the command line. The BuildConfig specifies the strategy to use, the source location, and the build output location.
ImageStreams	For BuildConfig , two ImageStreams are usually created. One represents the input image. With Source builds, this is the builder image. With Docker builds, this is the FROM image. The second one represents the output image. If a container image was specified as input to new-app , then an imagestream is created for that image as well.
DeploymentConfig	A DeploymentConfig is created either to deploy the output of a build, or a specified image. The new-app command creates emptyDir volumes for all Docker volumes that are specified in containers included in the resulting DeploymentConfig .
Service	The new-app command attempts to detect exposed ports in input images. It uses the lowest numeric exposed port to generate a service that exposes that port. In order to expose a different port, after new-app has completed, simply use the oc expose command to generate additional services.
Other	Other objects can be generated when instantiating templates, according to the template.

3.1.1.4.1. Specifying environment variables

When generating applications from a template, source, or an image, you can use the **-e|--env** argument to pass environment variables to the application container at run time:

```
$ oc new-app openshift/postgresql-92-centos7 \
  -e POSTGRESQL_USER=user \
  -e POSTGRESQL_DATABASE=db \
  -e POSTGRESQL_PASSWORD=password
```

The variables can also be read from file using the **--env-file** argument:

```
$ cat postgresql.env
POSTGRESQL_USER=user
POSTGRESQL_DATABASE=db
POSTGRESQL_PASSWORD=password
$ oc new-app openshift/postgresql-92-centos7 --env-file=postgresql.env
```

Additionally, environment variables can be given on standard input by using **--env-file=-:**

```
$ cat postgresql.env | oc new-app openshift/postgresql-92-centos7 --env-file=-
```



NOTE

Any **BuildConfig** objects created as part of **new-app** processing are not updated with environment variables passed with the **-e|--env** or **--env-file** argument.

3.1.1.4.2. Specifying build environment variables

When generating applications from a template, source, or an image, you can use the **--build-env** argument to pass environment variables to the build container at run time:

```
$ oc new-app openshift/ruby-23-centos7 \
  --build-env HTTP_PROXY=http://myproxy.net:1337/ \
  --build-env GEM_HOME=~/.gem
```

The variables can also be read from a file using the **--build-env-file** argument:

```
$ cat ruby.env
HTTP_PROXY=http://myproxy.net:1337/
GEM_HOME=~/.gem
$ oc new-app openshift/ruby-23-centos7 --build-env-file=ruby.env
```

Additionally, environment variables can be given on standard input by using **--build-env-file=-**:

```
$ cat ruby.env | oc new-app openshift/ruby-23-centos7 --build-env-file=-
```

3.1.1.4.3. Specifying labels

When generating applications from source, images, or templates, you can use the **-l|--label** argument to add labels to the created objects. Labels make it easy to collectively select, configure, and delete objects associated with the application.

```
$ oc new-app https://github.com/openshift/ruby-hello-world -l name=hello-world
```

3.1.1.4.4. Viewing the output without creation

To see a dry-run of running the **new-app** command, you can use the **-o|--output** argument with a **yaml** or **json** value. You can then use the output to preview the objects that are created or redirect it to a file that you can edit. After you are satisfied, you can use **oc create** to create the OpenShift Container Platform objects.

To output **new-app** artifacts to a file, edit them, then create them:

```
$ oc new-app https://github.com/openshift/ruby-hello-world \
  -o yaml > myapp.yaml
$ vi myapp.yaml
$ oc create -f myapp.yaml
```

3.1.1.4.5. Creating objects with different names

Objects created by **new-app** are normally named after the source repository, or the image used to generate them. You can set the name of the objects produced by adding a **--name** flag to the command:

```
$ oc new-app https://github.com/openshift/ruby-hello-world --name=myapp
```

3.1.1.4.6. Creating objects in a different project

Normally, **new-app** creates objects in the current project. However, you can create objects in a different project by using the **-n|--namespace** argument:

```
$ oc new-app https://github.com/openshift/ruby-hello-world -n myproject
```

3.1.1.4.7. Creating multiple objects

The **new-app** command allows creating multiple applications specifying multiple parameters to **new-app**. Labels specified in the command line apply to all objects created by the single command. Environment variables apply to all components created from source or images.

To create an application from a source repository and a Docker Hub image:

```
$ oc new-app https://github.com/openshift/ruby-hello-world mysql
```



NOTE

If a source code repository and a builder image are specified as separate arguments, **new-app** uses the builder image as the builder for the source code repository. If this is not the intent, specify the required builder image for the source using the **~** separator.

3.1.1.4.8. Grouping images and source in a single Pod

The **new-app** command allows deploying multiple images together in a single Pod. In order to specify which images to group together, use the **+** separator. The **--group** command line argument can also be used to specify the images that should be grouped together. To group the image built from a source repository with other images, specify its builder image in the group:

```
$ oc new-app ruby+mysql
```

To deploy an image built from source and an external image together:

```
$ oc new-app \  
  ruby~https://github.com/openshift/ruby-hello-world \  
  mysql \  
  --group=ruby+mysql
```

3.1.1.4.9. Searching for images, templates, and other inputs

To search for images, templates, and other inputs for the **oc new-app** command, add the **--search** and **--list** flags. For example, to find all of the images or templates that include PHP:

```
$ oc new-app --search php
```


CHAPTER 4. SERVICE BROKERS

4.1. INSTALLING THE SERVICE CATALOG



IMPORTANT

The service catalog is deprecated in OpenShift Container Platform 4. Equivalent and better functionality is present in the Operator Framework and Operator Lifecycle Manager (OLM).

4.1.1. About the service catalog

When developing microservices-based applications to run on cloud native platforms, there are many ways to provision different resources and share their coordinates, credentials, and configuration, depending on the service provider and the platform.

To give developers a more seamless experience, OpenShift Container Platform includes a *service catalog*, an implementation of the [Open Service Broker API](#) (OSB API) for Kubernetes. This allows users to connect any of their applications deployed in OpenShift Container Platform to a wide variety of service brokers.

The service catalog allows cluster administrators to integrate multiple platforms using a single API specification. The OpenShift Container Platform web console displays the cluster service classes offered by service brokers in the service catalog, allowing users to discover and instantiate those services for use with their applications.

As a result, service users benefit from ease and consistency of use across different types of services from different providers, while service providers benefit from having one integration point that gives them access to multiple platforms.

The service catalog is not installed by default in OpenShift Container Platform 4.

4.1.2. Installing service catalog

If you plan on using any of the services from the OpenShift Ansible Broker or Template Service Broker, you must install the service catalog by completing the following steps.

The custom resources for the service catalog's API server and controller manager are created by default in OpenShift Container Platform, but initially have a **managementState** of **Removed**. To install the service catalog, you must change the **managementState** for these resources to **Managed**.

Procedure

1. Enable the service catalog API server:
 - a. Use the following command to edit the service catalog API server resource:

```
$ oc edit servicecatalogapiservers
```

- b. Under **spec**, set the **managementState** field to **Managed**:

```
spec:
  logLevel: Normal
  managementState: Managed
```

- c. Save the file to apply the changes.
The Operator installs the service catalog API server component. As of OpenShift Container Platform 4, this component is installed into the **openshift-service-catalog-apiserver** namespace.

2. Enable the service catalog controller manager:

- a. Use the following command to edit the service catalog controller manager resource:

```
$ oc edit servicecatalogcontrollermanagers
```

- b. Under **spec**, set the **managementState** field to **Managed**:

```
spec:
  logLevel: Normal
  managementState: Managed
```

- c. Save the file to apply the changes.
The Operator installs the service catalog controller manager component. As of OpenShift Container Platform 4, this component is installed into the **openshift-service-catalog-controller-manager** namespace.

4.1.3. Uninstalling service catalog

To uninstall the service catalog, you must change the **managementState** for the service catalog's API server and controller manager resources from **Managed** to **Removed**.

Procedure

1. Disable the service catalog API server:

- a. Use the following command to edit the service catalog API server resource:

```
$ oc edit servicecatalogapiservers
```

- b. Under **spec**, set the **managementState** field to **Removed**:

```
spec:
  logLevel: Normal
  managementState: Removed
```

- c. Save the file to apply the changes.

2. Disable the service catalog controller manager:

- a. Use the following command to edit the service catalog controller manager resource:

```
$ oc edit servicecatalogcontrollermanagers
```

- b. Under **spec**, set the **managementState** field to **Removed**:

```
spec:
  logLevel: Normal
  managementState: Removed
```

- c. Save the file to apply the changes.

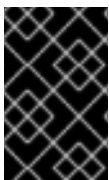


IMPORTANT

There is a known issue related to projects getting stuck in a "Terminating" state when attempting to delete them after disabling the service catalog. See the [OpenShift Container Platform 4.1 Release Notes](#) for a workaround. ([BZ#1746174](#))

4.2. INSTALLING THE TEMPLATE SERVICE BROKER

You can install the Template Service Broker to gain access to the template applications that it provides.



IMPORTANT

The Template Service Broker is deprecated in OpenShift Container Platform 4. Equivalent and better functionality is present in the Operator Framework and Operator Lifecycle Manager (OLM).

Prerequisites

- [Install the service catalog](#)

4.2.1. About the Template Service Broker

The *Template Service Broker* gives the service catalog visibility into the default Instant App and Quickstart templates that have shipped with OpenShift Container Platform since its initial release. The Template Service Broker can also make available as a service anything for which an OpenShift Container Platform template has been written, whether provided by Red Hat, a cluster administrator or user, or a third-party vendor.

By default, the Template Service Broker shows objects that are globally available from the **openshift** project. It can also be configured to watch any other project that a cluster administrator chooses.

The Template Service Broker is not installed by default in OpenShift Container Platform 4.

4.2.2. Installing the Template Service Broker Operator

Prerequisites

- You have installed the service catalog.

Procedure

The following procedure installs the Template Service Broker Operator using the web console.

1. Create a namespace.
 - a. Navigate in the web console to **Administration** → **Namespaces** and click **Create**

Namespace.

- b. Enter **openshift-template-service-broker** in the **Name** field and click **Create**.

**NOTE**

The namespace must start with **openshift-**.

2. Navigate to the **Catalog** → **OperatorHub** page. Verify that the **openshift-template-service-broker** project is selected.
3. Select **Template Service Broker Operator**.
4. Read the information about the Operator and click **Install**.
5. Review the default selections and click **Subscribe**.

Next, you must start the Template Service Broker in order to access the template applications it provides.

4.2.3. Starting the Template Service Broker

After you have installed the Template Service Broker Operator, you can start the Template Service Broker using the following procedure.

Prerequisites

- You have installed the service catalog.
- You have installed the Template Service Broker Operator.

Procedure

1. Navigate in the web console to **Catalog** → **Installed Operators** and select the **openshift-template-service-broker** project.
2. Select the **Template Service Broker Operator**.
3. Under **Provided APIs**, click **Create New** for **Template Service Broker**.
4. Review the default YAML and click **Create**.
5. Verify that the Template Service Broker has started.

After the Template Service Broker has started, you can view the available template applications by navigating to **Catalog** → **Developer Catalog** and selecting the **Service Class** checkbox. Note that it may take a few minutes for the Template Service Broker to start and the template applications to be available.

If you do not yet see these Service classes, you can check the status of the following items:

- Template Service Broker Pod status
 - From the **Workloads** → **Pods** page for the **openshift-template-service-broker** project, verify that the Pod that starts with **apiserver-** has a status of **Running** and readiness of **Ready**.

- Cluster service broker status
 - From the **Catalog → Broker Management → Service Brokers** page, verify that the **template-service-broker** service broker has a status of **Ready**.
- Service catalog controller manager Pod logs
 - From the **Workloads → Pods** page for the **openshift-service-catalog-controller-manager** project, review the logs for each of the Pods and verify that you see a log entry with the message **Successfully fetched catalog entries from broker**.

4.3. PROVISIONING TEMPLATE APPLICATIONS

4.3.1. Provisioning template applications

The following procedure provisions an example PostgreSQL template application that was made available by the Template Service Broker.

Prerequisites

- The service catalog is installed.
- The Template Service Broker is installed.

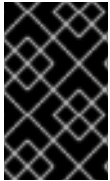
Procedure

1. Create a project.
 - a. Navigate in the web console to **Home → Projects** and click **Create Project**.
 - b. Enter **test-postgresql** in the **Name** field and click **Create**.
2. Create a service instance.
 - a. Navigate to the **Catalog → Developer Catalog** page.
 - b. Select the **PostgreSQL (Ephemeral)** template application and click **Create Service Instance**.
 - c. Review the default selections and set any other required fields, and click **Create**.
 - d. Go to **Catalog → Provisioned Services** and verify that the **postgresql-ephemeral** service instance is created and has a status of **Ready**.
You can check the progress on the **Home → Events** page. After a few moments, you should see an event for **postgresql-ephemeral** with the message "The instance was provisioned successfully".
3. Create a service binding.
 - a. From the **Provisioned Services** page, click **postgresql-ephemeral** and click **Create Service Binding**.
 - b. Review the default service binding name and click **Create**.
This creates a new secret for binding using the name provided.
4. Review the secret that was created.

- a. Navigate to **Workloads** → **Secrets** and verify that a secret named **postgresql-ephemeral** was created.
- b. Click **postgresql-ephemeral** and review the key-value pairs in the **Data** section, which are used for binding to other apps.

4.4. UNINSTALLING THE TEMPLATE SERVICE BROKER

You can uninstall the Template Service Broker if you no longer require access to the template applications that it provides.



IMPORTANT

The Template Service Broker is deprecated in OpenShift Container Platform 4. Equivalent and better functionality is present in the Operator Framework and Operator Lifecycle Manager (OLM).

4.4.1. Uninstalling the Template Service Broker

The following procedure uninstalls the Template Service Broker and its Operator using the web console.



WARNING

Do not uninstall the Template Service Broker if there are any provisioned services from it in your cluster, otherwise you might encounter errors when trying to manage the services.

Prerequisites

- The Template Service Broker is installed.

Procedure

This procedure assumes that you installed the Template Service Broker into the **openshift-template-service-broker** project.

1. Uninstall the Template Service Broker.
 - a. Navigate to **Catalog** → **Installed Operators** and select the **openshift-template-service-broker** project from the drop-down menu.
 - b. Click **Template Service Broker Operator**.
 - c. Select the **Template Service Broker** tab.
 - d. Click **template-service-broker**.
 - e. From the **Actions** drop-down menu, select **Delete Template Service Broker**.
 - f. Click **Delete** from the confirmation pop-up window.

The Template Service Broker is now uninstalled, and template applications will soon be removed from the Developer Catalog.

2. Uninstall the Template Service Broker Operator.
 - a. Navigate to **Catalog** → **Operator Management** and select the **openshift-template-service-broker** project from the drop-down menu.
 - b. Click **View subscription** for the **Template Service Broker Operator**.
 - c. Select **templateservicebroker**.
 - d. From the **Actions** drop-down menu, select **Remove Subscription**.
 - e. Verify that the checkbox is checked next to **Also completely remove the templateservicebroker Operator from the selected namespace** and click **Remove**. The Template Service Broker Operator is no longer installed in your cluster.

After the Template Service Broker is uninstalled, users will no longer have access to the template applications provided by the Template Service Broker.

4.5. INSTALLING THE OPENSIFT ANSIBLE BROKER

You can install the OpenShift Ansible Broker to gain access to the service bundles that it provides.



IMPORTANT

The OpenShift Ansible Broker is deprecated in OpenShift Container Platform 4. Equivalent and better functionality is present in the Operator Framework and Operator Lifecycle Manager (OLM).

Prerequisites

- [Install the service catalog](#)

4.5.1. About the OpenShift Ansible Broker

The *OpenShift Ansible Broker* is an implementation of the Open Service Broker (OSB) API that manages applications defined by *Ansible playbook bundles* (APBs). APBs provide a method for defining and distributing container applications in OpenShift Container Platform, and consist of a bundle of Ansible playbooks built into a container image with an Ansible runtime. APBs leverage Ansible to create a standard mechanism to automate complex deployments.

The OpenShift Ansible Broker follows this basic workflow:

1. A user requests the list of available applications from the service catalog using the OpenShift Container Platform web console.
2. The service catalog requests the list of available applications from the OpenShift Ansible Broker.
3. The OpenShift Ansible Broker communicates with a defined container image registry to learn which APBs are available.
4. The user issues a request to provision a specific APB.

5. The OpenShift Ansible Broker fulfills the user's provision request by invoking the provision method on the APB.

The OpenShift Ansible Broker is not installed by default in OpenShift Container Platform 4.

4.5.1.1. Ansible playbook bundles

An Ansible playbook bundle (APB) is a lightweight application definition that allows you to leverage existing investment in Ansible roles and playbooks.

APBs use a simple directory with named playbooks to perform OSB API actions, such as provision and bind. Metadata defined in the **apb.yml** file contains a list of required and optional parameters for use during deployment.

Additional resources

- [Ansible playbook bundle repository](#)

4.5.2. Installing the OpenShift Ansible Service Broker Operator

Prerequisites

- You have installed the service catalog.

Procedure

The following procedure installs the OpenShift Ansible Service Broker Operator using the web console.

1. Create a namespace.
 - a. Navigate in the web console to **Administration** → **Namespaces** and click **Create Namespace**.
 - b. Enter **openshift-ansible-service-broker** in the **Name** field and **openshift.io/cluster-monitoring=true** in the **Labels** field and click **Create**.



NOTE

The namespace must start with **openshift-**.

2. Create a cluster role binding.
 - a. Navigate to **Administration** → **Role Bindings** and click **Create Binding**.
 - b. For the **Binding Type**, select **Cluster-wide Role Binding (ClusterRoleBinding)**.
 - c. For the **Role Binding**, enter **ansible-service-broker** in the **Name** field.
 - d. For the **Role**, select **admin**.
 - e. For the **Subject**, choose the **Service Account** option, select the **openshift-ansible-service-broker** namespace, and enter **openshift-ansible-service-broker-operator** in the **Subject Name** field.
 - f. Click **Create**.

3. Create a secret to connect to the Red Hat Container Catalog.
 - a. Navigate to **Workloads** → **Secrets**. Verify that the **openshift-ansible-service-broker** project is selected.
 - b. Click **Create** → **Key/Value Secret**.
 - c. Enter **asb-registry-auth** as the **Secret Name**.
 - d. Add a **Key** of **username** and a **Value** of your Red Hat Container Catalog user name.
 - e. Click **Add Key/Value** and add a **Key** of **password** and a **Value** of your Red Hat Container Catalog password.
 - f. Click **Create**.
4. Navigate to the **Catalog** → **OperatorHub** page. Verify that the **openshift-ansible-service-broker** project is selected.
5. Select **OpenShift Ansible Service Broker Operator**.
6. Read the information about the Operator and click **Install**.
7. Review the default selections and click **Subscribe**.

Next, you must start the OpenShift Ansible Broker in order to access the service bundles it provides.

4.5.3. Starting the OpenShift Ansible Broker

After you have installed the OpenShift Ansible Service Broker Operator, you can start the OpenShift Ansible Broker using the following procedure.

Prerequisites

- You have installed the service catalog.
- You have installed the OpenShift Ansible Service Broker Operator.

Procedure

1. Navigate in the web console to **Catalog** → **Installed Operators** and select the **openshift-ansible-service-broker** project.
2. Select the **OpenShift Ansible Service Broker Operator**.
3. Under **Provided APIs**, click **Create New** for **Automation Broker**.
4. Add the following to the **spec** field in the default YAML provided:

```
registry:
  - name: rhcc
    type: rhcc
    url: https://registry.redhat.io
    auth_type: secret
    auth_name: asb-registry-auth
```

This references the secret that was created when installing the OpenShift Ansible Service Broker Operator, which allows you to connect to the Red Hat Container Catalog.

5. Set any additional OpenShift Ansible Broker configuration options and click **Create**.
6. Verify that the OpenShift Ansible Broker has started.
After the OpenShift Ansible Broker has started, you can view the available service bundles by navigating to **Catalog** → **Developer Catalog** and selecting the **Service Class** checkbox. Note that it may take a few minutes for the OpenShift Ansible Broker to start and the service bundles to be available.

If you do not yet see these Service classes, you can check the status of the following items:

- OpenShift Ansible Broker Pod status
 - From the **Workloads** → **Pods** page for the **openshift-ansible-service-broker** project, verify that the Pod that starts with **asb-** has a status of **Running** and readiness of **Ready**.
- Cluster service broker status
 - From the **Catalog** → **Broker Management** → **Service Brokers** page, verify that the **ansible-service-broker** service broker has a status of **Ready**.
- Service catalog controller manager Pod logs
 - From the **Workloads** → **Pods** page for the **openshift-service-catalog-controller-manager** project, review the logs for each of the Pods and verify that you see a log entry with the message **Successfully fetched catalog entries from broker**.

4.5.3.1. OpenShift Ansible Broker configuration options

You can set the following options for your OpenShift Ansible Broker.

Table 4.1. OpenShift Ansible Broker configuration options

YAML key	Description	Default value
brokerName	The name used to identify the broker instance.	ansible-service-broker
brokerNamespace	The namespace where the broker resides.	openshift-ansible-service-broker
brokerImage	The fully qualified image used for the broker.	docker.io/ansibleplaybookbundle/origin-ansible-service-broker:v4.0
brokerImagePullPolicy	The pull policy used for the broker image itself.	IfNotPresent
brokerNodeSelector	The node selector string used for the broker's deployment.	"

YAML key	Description	Default value
registries	Expressed as a yaml list of broker registry configs, allowing the user to configure the image registries the broker will discover and source its APBs from.	See the default registries array .
logLevel	The log level used for the broker's logs.	info
apbPullPolicy	The pull policy used for APB Pods.	IfNotPresent
sandboxRole	The role granted to the service account used to execute APBs.	edit
keepNamespace	Whether the transient namespace created to run the APB is deleted after the conclusion of the APB, regardless of the result.	false
keepNamespaceOnError	Whether the transient namespace created to run the APB is deleted after the conclusion of the APB, only in the event of an error result.	false
bootstrapOnStartup	Whether or not the broker should run its bootstrap routine on startup.	true
refreshInterval	The interval of time between broker bootstraps, refreshing its inventory of APBs.	600s
launchApbOnBind	<i>Experimental:</i> Toggles the broker executing APBs on bind operations.	false
autoEscalate	Whether the broker should escalate the permissions of a user while running the APB. This should typically remain false since the broker performs originating user authorization to ensure that the user has permissions granted to the APB sandbox.	false
outputRequest	Whether to output the low level HTTP requests that the broker receives.	false

Default array for registries

```
- type: rhcc
  name: rhcc
  url: https://registry.redhat.io
  white_list:
  - ".*-apb$"
  auth_type: secret
  auth_name: asb-registry-auth
```

4.6. CONFIGURING THE OPENSIFT ANSIBLE BROKER



IMPORTANT

The OpenShift Ansible Broker is deprecated in OpenShift Container Platform 4. Equivalent and better functionality is present in the Operator Framework and Operator Lifecycle Manager (OLM).

4.6.1. Configuring the OpenShift Ansible Broker

The following procedure customizes the settings for your OpenShift Ansible Broker.

Prerequisites

- You have installed and started the OpenShift Ansible Broker.

Procedure

This procedure assumes that you used **ansible-service-broker** both as the OpenShift Ansible Broker name and the project that it was installed into.

- Navigate in the web console to **Catalog** → **Installed Operators** and select the **ansible-service-broker** project.
- Select the **OpenShift Ansible Service Broker Operator**.
- On the **Automation Broker** tab, select **ansible-service-broker**.
- On the **YAML** tab, add or update any OpenShift Ansible Broker configuration options under the **spec** field.

For example:

```
spec:
  keepNamespace: true
  sandboxRole: edit
```

- Click **Save** to apply these changes.

4.6.1.1. OpenShift Ansible Broker configuration options

You can set the following options for your OpenShift Ansible Broker.

Table 4.2. OpenShift Ansible Broker configuration options

YAML key	Description	Default value
brokerName	The name used to identify the broker instance.	ansible-service-broker
brokerNamespace	The namespace where the broker resides.	openshift-ansible-service-broker

YAML key	Description	Default value
brokerImage	The fully qualified image used for the broker.	docker.io/ansibleplaybookbundle/origin-ansible-service-broker:v4.0
brokerImagePullPolicy	The pull policy used for the broker image itself.	IfNotPresent
brokerNodeSelector	The node selector string used for the broker's deployment.	"
registries	Expressed as a yaml list of broker registry configs, allowing the user to configure the image registries the broker will discover and source its APBs from.	See the default registries array .
logLevel	The log level used for the broker's logs.	info
apbPullPolicy	The pull policy used for APB Pods.	IfNotPresent
sandboxRole	The role granted to the service account used to execute APBs.	edit
keepNamespace	Whether the transient namespace created to run the APB is deleted after the conclusion of the APB, regardless of the result.	false
keepNamespaceOnError	Whether the transient namespace created to run the APB is deleted after the conclusion of the APB, only in the event of an error result.	false
bootstrapOnStartup	Whether or not the broker should run its bootstrap routine on startup.	true
refreshInterval	The interval of time between broker bootstraps, refreshing its inventory of APBs.	600s
launchApbOnBind	<i>Experimental:</i> Toggles the broker executing APBs on bind operations.	false
autoEscalate	Whether the broker should escalate the permissions of a user while running the APB. This should typically remain false since the broker performs originating user authorization to ensure that the user has permissions granted to the APB sandbox.	false

YAML key	Description	Default value
outputRequest	Whether to output the low level HTTP requests that the broker receives.	false

Default array for registries

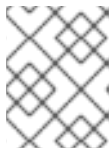
```
- type: rhcc
  name: rhcc
  url: https://registry.redhat.io
  white_list:
  - ".*-apb$"
  auth_type: secret
  auth_name: asb-registry-auth
```

4.6.2. Configuring monitoring for the OpenShift Ansible Broker

In order for Prometheus to monitor the OpenShift Ansible Broker, you must create the following resources to grant Prometheus permission to access the namespace where the OpenShift Ansible Broker was installed.

Prerequisites

- The OpenShift Ansible Broker is installed.



NOTE

This procedure assumes that you installed the OpenShift Ansible Broker into the **openshift-ansible-service-broker** namespace.

Procedure

- Create the role.
 - Navigate to **Administration** → **Roles** and click **Create Role**.
 - Replace the YAML in the editor with the following:

```
apiVersion: rbac.authorization.k8s.io/v1
kind: Role
metadata:
  name: prometheus-k8s
  namespace: openshift-ansible-service-broker
rules:
- apiGroups:
  - ""
  resources:
  - services
  - endpoints
  - pods
  verbs:
```

- get
- list
- watch

- c. Click **Create**.
2. Create the role binding.
 - a. Navigate to **Administration** → **Role Bindings** and click **Create Binding**.
 - b. For the **Binding Type**, select **Namespace Role Binding (RoleBinding)**.
 - c. For the **Role Binding**, enter **prometheus-k8s** in the **Name** field and **openshift-ansible-service-broker** in the **Namespace** field.
 - d. For the **Role**, select **prometheus-k8s**.
 - e. For the **Subject**, choose the **Service Account** option, select the **openshift-monitoring** namespace, and enter **prometheus-k8s** in the **Subject Name** field.
 - f. Click **Create**.

Prometheus will now have access to OpenShift Ansible Broker metrics.

4.7. PROVISIONING SERVICE BUNDLES

4.7.1. Provisioning service bundles

The following procedure provisions an example PostgreSQL service bundle (APB) that was made available by the OpenShift Ansible Broker.

Prerequisites

- The service catalog is installed.
- The OpenShift Ansible Broker is installed.

Procedure

1. Create a project.
 - a. Navigate in the web console to **Home** → **Projects** and click **Create Project**.
 - b. Enter **test-postgresql-apb** in the **Name** field and click **Create**.
2. Create a service instance.
 - a. Navigate to the **Catalog** → **Developer Catalog** page.
 - b. Select the **PostgreSQL (APB)** service bundle and click **Create Service Instance**.
 - c. Review the default selections and set any other required fields, and click **Create**.
 - d. Go to **Catalog** → **Provisioned Services** and verify that the **dh-postgresql-apb** service instance is created and has a status of **Ready**.

You can check the progress on the **Home** → **Events** page. After a few moments, you should see an event for **dh-postgresql-apb** with the message "The instance was provisioned successfully".

3. Create a service binding.
 - a. From the **Provisioned Services** page, click **dh-postgresql-apb** and click **Create Service Binding**.
 - b. Review the default service binding name and click **Create**.
This creates a new secret for binding using the name provided.
4. Review the secret that was created.
 - a. Navigate to **Workloads** → **Secrets** and verify that a secret named **dh-postgresql-apb** was created.
 - b. Click **dh-postgresql-apb** and review the key-value pairs in the **Data** section, which are used for binding to other apps.

4.8. UNINSTALLING THE OPENSIFT ANSIBLE BROKER

You can uninstall the OpenShift Ansible Broker if you no longer require access to the service bundles that it provides.



IMPORTANT

The OpenShift Ansible Broker is deprecated in OpenShift Container Platform 4. Equivalent and better functionality is present in the Operator Framework and Operator Lifecycle Manager (OLM).

4.8.1. Uninstalling the OpenShift Ansible Broker

The following procedure uninstalls the OpenShift Ansible Broker and its Operator using the web console.



WARNING

Do not uninstall the OpenShift Ansible Broker if there are any provisioned services from it in your cluster, otherwise you might encounter errors when trying to manage the services.

Prerequisites

- The OpenShift Ansible Broker is installed.

Procedure

This procedure assumes that you installed the OpenShift Ansible Broker into the **openshift-ansible-service-broker** project.

1. Uninstall the OpenShift Ansible Broker.
 - a. Navigate to **Catalog** → **Installed Operators** and select the **openshift-ansible-service-broker** project from the drop-down menu.
 - b. Click **OpenShift Ansible Service Broker Operator**.
 - c. Select the **Automation Broker** tab.
 - d. Click **ansible-service-broker**.
 - e. From the **Actions** drop-down menu, select **Delete Automation Broker**.
 - f. Click **Delete** from the confirmation pop-up window.

The OpenShift Ansible Broker is now uninstalled, and service bundles will soon be removed from the Developer Catalog.

2. Uninstall the OpenShift Ansible Service Broker Operator.
 - a. Navigate to **Catalog** → **Operator Management** and select the **openshift-ansible-service-broker** project from the drop-down menu.
 - b. Click **View subscription** for the **OpenShift Ansible Service Broker Operator**.
 - c. Select **automationbroker**.
 - d. From the **Actions** drop-down menu, select **Remove Subscription**.
 - e. Verify that the checkbox is checked next to **Also completely remove the automationbroker Operator from the selected namespace** and click **Remove**.

The OpenShift Ansible Service Broker Operator is no longer installed in your cluster.

After the OpenShift Ansible Broker is uninstalled, users will no longer have access to the service bundles provided by the OpenShift Ansible Broker.

CHAPTER 5. DEPLOYMENTS

5.1. UNDERSTANDING DEPLOYMENTS AND DEPLOYMENTCONFIGS

Deployments and *DeploymentConfigs* in OpenShift Container Platform are API objects that provide two similar but different methods for fine-grained management over common user applications. They are composed of the following separate API objects:

- A *DeploymentConfig* or a *Deployment*, either of which describes the desired state of a particular component of the application as a Pod template.
- *DeploymentConfigs* involve one or more *ReplicationControllers*, which contain a point-in-time record of the state of a *DeploymentConfig* as a Pod template. Similarly, *Deployments* involve one or more *ReplicaSets*, a successor of *ReplicationControllers*.
- One or more *Pods*, which represent an instance of a particular version of an application.

5.1.1. Building blocks of a deployment

Deployments and *DeploymentConfigs* are enabled by the use of native Kubernetes API objects *ReplicationControllers* and *ReplicaSets*, respectively, as their building blocks.

Users do not have to manipulate *ReplicationControllers*, *ReplicaSets*, or *Pods* owned by *DeploymentConfigs* or *Deployments*. The deployment systems ensures changes are propagated appropriately.

TIP

If the existing deployment strategies are not suited for your use case and you must run manual steps during the lifecycle of your deployment, then you should consider creating a Custom deployment strategy.

The following sections provide further details on these objects.

5.1.1.1. ReplicationControllers

A *ReplicationController* ensures that a specified number of replicas of a Pod are running at all times. If Pods exit or are deleted, the *ReplicationController* acts to instantiate more up to the defined number. Likewise, if there are more running than desired, it deletes as many as necessary to match the defined amount.

A *ReplicationController* configuration consists of:

- The number of replicas desired (which can be adjusted at runtime).
- A Pod definition to use when creating a replicated Pod.
- A selector for identifying managed Pods.

A selector is a set of labels assigned to the Pods that are managed by the *ReplicationController*. These labels are included in the Pod definition that the *ReplicationController* instantiates. The *ReplicationController* uses the selector to determine how many instances of the Pod are already running in order to adjust as needed.

The ReplicationController does not perform auto-scaling based on load or traffic, as it does not track either. Rather, this requires its replica count to be adjusted by an external auto-scaler.

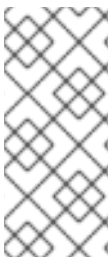
The following is an example definition of a ReplicationController:

```
apiVersion: v1
kind: ReplicationController
metadata:
  name: frontend-1
spec:
  replicas: 1 ①
  selector: ②
    name: frontend
  template: ③
    metadata:
      labels: ④
        name: frontend ⑤
    spec:
      containers:
      - image: openshift/hello-openshift
        name: helloworld
        ports:
        - containerPort: 8080
          protocol: TCP
      restartPolicy: Always
```

- ① The number of copies of the Pod to run.
- ② The label selector of the Pod to run.
- ③ A template for the Pod the controller creates.
- ④ Labels on the Pod should include those from the label selector.
- ⑤ The maximum name length after expanding any parameters is 63 characters.

5.1.1.2. ReplicaSets

Similar to a ReplicationController, a ReplicaSet is a native Kubernetes API object that ensures a specified number of pod replicas are running at any given time. The difference between a ReplicaSet and a ReplicationController is that a ReplicaSet supports set-based selector requirements whereas a replication controller only supports equality-based selector requirements.



NOTE

Only use ReplicaSets if you require custom update orchestration or do not require updates at all. Otherwise, use Deployments. ReplicaSets can be used independently, but are used by deployments to orchestrate pod creation, deletion, and updates. Deployments manage their ReplicaSets automatically, provide declarative updates to pods, and do not have to manually manage the ReplicaSets that they create.

The following is an example **ReplicaSet** definition:

```

apiVersion: apps/v1
kind: ReplicaSet
metadata:
  name: frontend-1
  labels:
    tier: frontend
spec:
  replicas: 3
  selector: ①
  matchLabels: ②
    tier: frontend
  matchExpressions: ③
    - {key: tier, operator: In, values: [frontend]}
  template:
    metadata:
      labels:
        tier: frontend
    spec:
      containers:
        - image: openshift/hello-openshift
          name: helloworld
          ports:
            - containerPort: 8080
              protocol: TCP
          restartPolicy: Always

```

- ① A label query over a set of resources. The result of **matchLabels** and **matchExpressions** are logically conjoined.
- ② Equality-based selector to specify resources with labels that match the selector.
- ③ Set-based selector to filter keys. This selects all resources with key equal to **tier** and value equal to **frontend**.

5.1.2. DeploymentConfigs

Building on ReplicationControllers, OpenShift Container Platform adds expanded support for the software development and deployment lifecycle with the concept of *DeploymentConfigs*. In the simplest case, a DeploymentConfig creates a new ReplicationController and lets it start up Pods.

However, OpenShift Container Platform deployments from DeploymentConfigs also provide the ability to transition from an existing deployment of an image to a new one and also define hooks to be run before or after creating the ReplicationController.

The DeploymentConfig deployment system provides the following capabilities:

- A DeploymentConfig, which is a template for running applications.
- Triggers that drive automated deployments in response to events.
- User-customizable deployment strategies to transition from the previous version to the new version. A strategy runs inside a Pod commonly referred as the deployment process.
- A set of hooks (lifecycle hooks) for executing custom behavior in different points during the lifecycle of a deployment.

- Versioning of your application in order to support rollbacks either manually or automatically in case of deployment failure.
- Manual replication scaling and autoscaling.

When you create a DeploymentConfig, a ReplicationController is created representing the DeploymentConfig's Pod template. If the DeploymentConfig changes, a new ReplicationController is created with the latest Pod template, and a deployment process runs to scale down the old ReplicationController and scale up the new one.

Instances of your application are automatically added and removed from both service load balancers and routers as they are created. As long as your application supports graceful shutdown when it receives the **TERM** signal, you can ensure that running user connections are given a chance to complete normally.

The OpenShift Container Platform **DeploymentConfig** object defines the following details:

1. The elements of a **ReplicationController** definition.
2. Triggers for creating a new deployment automatically.
3. The strategy for transitioning between deployments.
4. Lifecycle hooks.

Each time a deployment is triggered, whether manually or automatically, a deployer Pod manages the deployment (including scaling down the old ReplicationController, scaling up the new one, and running hooks). The deployment pod remains for an indefinite amount of time after it completes the Deployment in order to retain its logs of the Deployment. When a deployment is superseded by another, the previous ReplicationController is retained to enable easy rollback if needed.

Example DeploymentConfig definition

```
apiVersion: v1
kind: DeploymentConfig
metadata:
  name: frontend
spec:
  replicas: 5
  selector:
    name: frontend
  template: { ... }
  triggers:
  - type: ConfigChange 1
  - imageChangeParams:
      automatic: true
      containerNames:
      - helloworld
      from:
        kind: ImageStreamTag
        name: hello-openshift:latest
      type: ImageChange 2
  strategy:
    type: Rolling 3
```

- 1 A **ConfigChange** trigger causes a new Deployment to be created any time the ReplicationController template changes.
- 2 An **ImageChange** trigger causes a new Deployment to be created each time a new version of the backing image is available in the named imagestream.
- 3 The default **Rolling** strategy makes a downtime-free transition between Deployments.

5.1.3. Deployments

Kubernetes provides a first-class, native API object type in OpenShift Container Platform called *Deployments*. Deployments serve as a descendant of the OpenShift Container Platform-specific DeploymentConfig.

Like DeploymentConfigs, Deployments describe the desired state of a particular component of an application as a Pod template. Deployments create ReplicaSets, which orchestrate Pod lifecycles.

For example, the following Deployment definition creates a ReplicaSet to bring up one **hello-openshift** Pod:

Deployment definition

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: hello-openshift
spec:
  replicas: 1
  selector:
    matchLabels:
      app: hello-openshift
  template:
    metadata:
      labels:
        app: hello-openshift
    spec:
      containers:
        - name: hello-openshift
          image: openshift/hello-openshift:latest
          ports:
            - containerPort: 80
```

5.1.4. Comparing Deployments and DeploymentConfigs

Both Kubernetes Deployments and OpenShift Container Platform-provided DeploymentConfigs are supported in OpenShift Container Platform; however, it is recommended to use Deployments unless you need a specific feature or behavior provided by DeploymentConfigs.

The following sections go into more detail on the differences between the two object types to further help you decide which type to use.

5.1.4.1. Design

One important difference between Deployments and DeploymentConfigs is the properties of the [CAP theorem](#) that each design has chosen for the rollout process. DeploymentConfigs prefer consistency, whereas Deployments take availability over consistency.

For DeploymentConfigs, if a node running a deployer Pod goes down, it will not get replaced. The process waits until the node comes back online or is manually deleted. Manually deleting the node also deletes the corresponding Pod. This means that you can not delete the Pod to unstick the rollout, as the kubelet is responsible for deleting the associated Pod.

However, Deployments rollouts are driven from a controller manager. The controller manager runs in high availability mode on masters and uses leader election algorithms to value availability over consistency. During a failure it is possible for other masters to act on the same Deployment at the same time, but this issue will be reconciled shortly after the failure occurs.

5.1.4.2. DeploymentConfigs-specific features

Automatic rollbacks

Currently, Deployments do not support automatically rolling back to the last successfully deployed ReplicaSet in case of a failure.

Triggers

Deployments have an implicit **ConfigChange** trigger in that every change in the pod template of a deployment automatically triggers a new rollout. If you do not want new rollouts on pod template changes, pause the deployment:

```
$ oc rollout pause deployments/<name>
```

Lifecycle hooks

Deployments do not yet support any lifecycle hooks.

Custom strategies

Deployments do not support user-specified Custom deployment strategies yet.

5.1.4.3. Deployments-specific features

Rollover

The deployment process for Deployments is driven by a controller loop, in contrast to DeploymentConfigs which use deployer pods for every new rollout. This means that a Deployment can have as many active ReplicaSets as possible, and eventually the deployment controller will scale down all old ReplicaSets and scale up the newest one.

DeploymentConfigs can have at most one deployer pod running, otherwise multiple deployers end up conflicting while trying to scale up what they think should be the newest ReplicationController. Because of this, only two ReplicationControllers can be active at any point in time. Ultimately, this translates to faster rapid rollouts for Deployments.

Proportional scaling

Because the Deployment controller is the sole source of truth for the sizes of new and old ReplicaSets owned by a Deployment, it is able to scale ongoing rollouts. Additional replicas are distributed proportionally based on the size of each ReplicaSet.

DeploymentConfigs cannot be scaled when a rollout is ongoing because the DeploymentConfig controller will end up having issues with the deployer process about the size of the new ReplicationController.

Pausing mid-rollout

Deployments can be paused at any point in time, meaning you can also pause ongoing rollouts. On the other hand, you cannot pause deployer pods currently, so if you try to pause a DeploymentConfig in the middle of a rollout, the deployer process will not be affected and will continue until it finishes.

5.2. MANAGING DEPLOYMENT PROCESSES

5.2.1. Managing DeploymentConfigs

DeploymentConfigs can be managed from the OpenShift Container Platform web console's **Workloads** page or using the **oc** CLI. The following procedures show CLI usage unless otherwise stated.

5.2.1.1. Starting a deployment

You can start a *rollout* to begin the deployment process of your application.

Procedure

1. To start a new deployment process from an existing DeploymentConfig, run the following command:

```
$ oc rollout latest dc/<name>
```



NOTE

If a deployment process is already in progress, the command displays a message and a new ReplicationController will not be deployed.

5.2.1.2. Viewing a deployment

You can view a deployment to get basic information about all the available revisions of your application.

Procedure

1. To show details about all recently created ReplicationControllers for the provided DeploymentConfig, including any currently running deployment process, run the following command:

```
$ oc rollout history dc/<name>
```

2. To view details specific to a revision, add the **--revision** flag:

```
$ oc rollout history dc/<name> --revision=1
```

3. For more detailed information about a deployment configuration and its latest revision, use the **oc describe** command:

```
$ oc describe dc <name>
```

5.2.1.3. Retrying a deployment

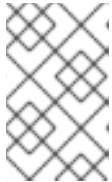
If the current revision of your DeploymentConfig failed to deploy, you can restart the deployment process.

Procedure

1. To restart a failed deployment process:

```
$ oc rollout retry dc/<name>
```

If the latest revision of it was deployed successfully, the command displays a message and the deployment process is not be retried.



NOTE

Retrying a deployment restarts the deployment process and does not create a new deployment revision. The restarted ReplicationController has the same configuration it had when it failed.

5.2.1.4. Rolling back a deployment

Rollbacks revert an application back to a previous revision and can be performed using the REST API, the CLI, or the web console.

Procedure

1. To rollback to the last successful deployed revision of your configuration:

```
$ oc rollout undo dc/<name>
```

The DeploymentConfig's template is reverted to match the deployment revision specified in the undo command, and a new ReplicationController is started. If no revision is specified with **--to-revision**, then the last successfully deployed revision is used.

2. Image change triggers on the DeploymentConfig are disabled as part of the rollback to prevent accidentally starting a new deployment process soon after the rollback is complete.
To re-enable the image change triggers:

```
$ oc set triggers dc/<name> --auto
```



NOTE

DeploymentConfigs also support automatically rolling back to the last successful revision of the configuration in case the latest deployment process fails. In that case, the latest template that failed to deploy stays intact by the system and it is up to users to fix their configurations.

5.2.1.5. Executing commands inside a container

You can add a command to a container, which modifies the container's startup behavior by overruling the image's **ENTRYPOINT**. This is different from a lifecycle hook, which instead can be run once per deployment at a specified time.

Procedure

1. Add the **command** parameters to the **spec** field of the DeploymentConfig. You can also add an **args** field, which modifies the **command** (or the **ENTRYPOINT** if **command** does not exist).

```
spec:
  containers:
  -
    name: <container_name>
    image: 'image'
    command:
    - '<command>'
    args:
    - '<argument_1>'
    - '<argument_2>'
    - '<argument_3>'
```

For example, to execute the **java** command with the **-jar** and **/opt/app-root/springboots2idemo.jar** arguments:

```
spec:
  containers:
  -
    name: example-spring-boot
    image: 'image'
    command:
    - java
    args:
    - '-jar'
    - '/opt/app-root/springboots2idemo.jar'
```

5.2.1.6. Viewing deployment logs

Procedure

1. To stream the logs of the latest revision for a given DeploymentConfig:

```
$ oc logs -f dc/<name>
```

If the latest revision is running or failed, the command returns the logs of the process that is responsible for deploying your pods. If it is successful, it returns the logs from a Pod of your application.

2. You can also view logs from older failed deployment processes, if and only if these processes (old ReplicationControllers and their deployer Pods) exist and have not been pruned or deleted manually:

```
$ oc logs --version=1 dc/<name>
```

5.2.1.7. Deployment triggers

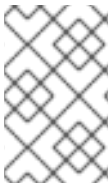
A DeploymentConfig can contain triggers, which drive the creation of new deployment processes in response to events inside the cluster.

**WARNING**

If no triggers are defined on a DeploymentConfig, a **ConfigChange** trigger is added by default. If triggers are defined as an empty field, deployments must be started manually.

ConfigChange deployment triggers

The **ConfigChange** trigger results in a new ReplicationController whenever configuration changes are detected in the Pod template of the DeploymentConfig.

**NOTE**

If a **ConfigChange** trigger is defined on a DeploymentConfig, the first ReplicationController is automatically created soon after the DeploymentConfig itself is created and it is not paused.

ConfigChange deployment trigger

```
triggers:
  - type: "ConfigChange"
```

ImageChange deployment triggers

The **ImageChange** trigger results in a new ReplicationController whenever the content of an imagestreamtag changes (when a new version of the image is pushed).

ImageChange deployment trigger

```
triggers:
  - type: "ImageChange"
    imageChangeParams:
      automatic: true 1
      from:
        kind: "ImageStreamTag"
        name: "origin-ruby-sample:latest"
        namespace: "myproject"
      containerNames:
        - "helloworld"
```

1 If the **imageChangeParams.automatic** field is set to **false**, the trigger is disabled.

With the above example, when the **latest** tag value of the **origin-ruby-sample** imagestream changes and the new image value differs from the current image specified in the DeploymentConfig's **helloworld** container, a new ReplicationController is created using the new image for the **helloworld** container.

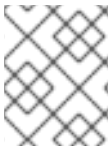
**NOTE**

If an **ImageChange** trigger is defined on a DeploymentConfig (with a **ConfigChange** trigger and **automatic=false**, or with **automatic=true**) and the **ImageStreamTag** pointed by the **ImageChange** trigger does not exist yet, then the initial deployment process will automatically start as soon as an image is imported or pushed by a build to the **ImageStreamTag**.

5.2.1.7.1. Setting deployment triggers**Procedure**

1. You can set deployment triggers for a DeploymentConfig using the **oc set triggers** command. For example, to set a **ImageChangeTrigger**, use the following command:

```
$ oc set triggers dc/<dc_name> \
  --from-image=<project>/<image>:<tag> -c <container_name>
```

5.2.1.8. Setting deployment resources**NOTE**

This resource is available only if a cluster administrator has enabled the ephemeral storage technology preview. This feature is disabled by default.

A deployment is completed by a Pod that consumes resources (memory, CPU, and ephemeral storage) on a node. By default, Pods consume unbounded node resources. However, if a project specifies default container limits, then Pods consume resources up to those limits.

You can also limit resource use by specifying resource limits as part of the deployment strategy. Deployment resources can be used with the Recreate, Rolling, or Custom deployment strategies.

Procedure

1. In the following example, each of **resources**, **cpu**, **memory**, and **ephemeral-storage** is optional:

```
type: "Recreate"
resources:
  limits:
    cpu: "100m" 1
    memory: "256Mi" 2
    ephemeral-storage: "1Gi" 3
```

- 1 **cpu** is in CPU units: **100m** represents 0.1 CPU units ($100 * 1e-3$).
- 2 **memory** is in bytes: **256Mi** represents 268435456 bytes ($256 * 2^{20}$).
- 3 **ephemeral-storage** is in bytes: **1Gi** represents 1073741824 bytes (2^{30}). This applies only if your cluster administrator enabled the ephemeral storage technology preview.

However, if a quota has been defined for your project, one of the following two items is required:

- A **resources** section set with an explicit **requests**:

```
type: "Recreate"
resources:
  requests: 1
    cpu: "100m"
    memory: "256Mi"
    ephemeral-storage: "1Gi"
```

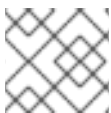
- 1 The **requests** object contains the list of resources that correspond to the list of resources in the quota.

- A limit range defined in your project, where the defaults from the **LimitRange** object apply to Pods created during the deployment process.

To set deployment resources, choose one of the above options. Otherwise, deploy Pod creation fails, citing a failure to satisfy quota.

5.2.1.9. Scaling manually

In addition to rollbacks, you can exercise fine-grained control over the number of replicas by manually scaling them.



NOTE

Pods can also be autoscaled using the **oc autoscale** command.

Procedure

1. To manually scale a DeploymentConfig, use the **oc scale** command. For example, the following command sets the replicas in the **frontend** DeploymentConfig to **3**.

```
$ oc scale dc frontend --replicas=3
```

The number of replicas eventually propagates to the desired and current state of the deployment configured by the DeploymentConfig **frontend**.

5.2.1.10. Accessing private repositories from DeploymentConfigs

You can add a Secret to your DeploymentConfig so that it can access images from a private repository. This procedure shows the OpenShift Container Platform web console method.

Procedure

1. Create a new project.
2. From the **Workloads** page, create a Secret that contains credentials for accessing a private image repository.
3. Create a DeploymentConfig.
4. On the DeploymentConfig editor page, set the **Pull Secret** and save your changes.

5.2.1.11. Assigning pods to specific nodes

You can use node selectors in conjunction with labeled nodes to control Pod placement.

Cluster administrators can set the default node selector for a project in order to restrict Pod placement to specific nodes. As a developer, you can set a node selector on a Pod configuration to restrict nodes even further.

Procedure

1. To add a node selector when creating a pod, edit the Pod configuration, and add the **nodeSelector** value. This can be added to a single Pod configuration, or in a Pod template:

```
apiVersion: v1
kind: Pod
spec:
  nodeSelector:
    disktype: ssd
  ...
```

Pods created when the node selector is in place are assigned to nodes with the specified labels. The labels specified here are used in conjunction with the labels added by a cluster administrator.

For example, if a project has the **type=user-node** and **region=east** labels added to a project by the cluster administrator, and you add the above **disktype: ssd** label to a Pod, the Pod is only ever scheduled on nodes that have all three labels.



NOTE

Labels can only be set to one value, so setting a node selector of **region=west** in a Pod configuration that has **region=east** as the administrator-set default, results in a Pod that will never be scheduled.

5.2.1.12. Running a Pod with a different service account

You can run a Pod with a service account other than the default.

Procedure

1. Edit the DeploymentConfig:

```
$ oc edit dc/<deployment_config>
```

2. Add the **serviceAccount** and **serviceAccountName** parameters to the **spec** field, and specify the service account you want to use:

```
spec:
  securityContext: {}
  serviceAccount: <service_account>
  serviceAccountName: <service_account>
```

5.3. USING DEPLOYMENTCONFIG STRATEGIES

A *deployment strategy* is a way to change or upgrade an application. The aim is to make the change without downtime in a way that the user barely notices the improvements.

Because the end user usually accesses the application through a route handled by a router, the deployment strategy can focus on DeploymentConfig features or routing features. Strategies that focus on the DeploymentConfig impact all routes that use the application. Strategies that use router features target individual routes.

Many deployment strategies are supported through the DeploymentConfig, and some additional strategies are supported through router features. DeploymentConfig strategies are discussed in this section.

Choosing a deployment strategy

Consider the following when choosing a deployment strategy:

- Long-running connections must be handled gracefully.
- Database conversions can be complex and must be done and rolled back along with the application.
- If the application is a hybrid of microservices and traditional components, downtime might be required to complete the transition.
- You must have the infrastructure to do this.
- If you have a non-isolated test environment, you can break both new and old versions.

A deployment strategy uses readiness checks to determine if a new Pod is ready for use. If a readiness check fails, the DeploymentConfig retries to run the Pod until it times out. The default timeout is **10m**, a value set in **TimeoutSeconds** in **dc.spec.strategy.*params**.

5.3.1. Rolling strategy

A rolling deployment slowly replaces instances of the previous version of an application with instances of the new version of the application. The Rolling strategy is the default deployment strategy used if no strategy is specified on a DeploymentConfig.

A rolling deployment typically waits for new pods to become **ready** via a **readiness check** before scaling down the old components. If a significant issue occurs, the rolling deployment can be aborted.

When to use a Rolling deployment:

- When you want to take no downtime during an application update.
- When your application supports having old code and new code running at the same time.

A Rolling deployment means you to have both old and new versions of your code running at the same time. This typically requires that your application handle N-1 compatibility.

Example Rolling strategy definition

```
strategy:
  type: Rolling
  rollingParams:
    updatePeriodSeconds: 1 1
```

```

intervalSeconds: 1 2
timeoutSeconds: 120 3
maxSurge: "20%" 4
maxUnavailable: "10%" 5
pre: {} 6
post: {}

```

- 1 The time to wait between individual Pod updates. If unspecified, this value defaults to **1**.
- 2 The time to wait between polling the deployment status after update. If unspecified, this value defaults to **1**.
- 3 The time to wait for a scaling event before giving up. Optional; the default is **600**. Here, *giving up* means automatically rolling back to the previous complete deployment.
- 4 **maxSurge** is optional and defaults to **25%** if not specified. See the information below the following procedure.
- 5 **maxUnavailable** is optional and defaults to **25%** if not specified. See the information below the following procedure.
- 6 **pre** and **post** are both lifecycle hooks.

The Rolling strategy:

1. Executes any **pre** lifecycle hook.
2. Scales up the new ReplicationController based on the surge count.
3. Scales down the old ReplicationController based on the max unavailable count.
4. Repeats this scaling until the new ReplicationController has reached the desired replica count and the old ReplicationController has been scaled to zero.
5. Executes any **post** lifecycle hook.



IMPORTANT

When scaling down, the Rolling strategy waits for Pods to become ready so it can decide whether further scaling would affect availability. If scaled up Pods never become ready, the deployment process will eventually time out and result in a deployment failure.

The **maxUnavailable** parameter is the maximum number of Pods that can be unavailable during the update. The **maxSurge** parameter is the maximum number of Pods that can be scheduled above the original number of Pods. Both parameters can be set to either a percentage (e.g., **10%**) or an absolute value (e.g., **2**). The default value for both is **25%**.

These parameters allow the deployment to be tuned for availability and speed. For example:

- **maxUnavailable*=0** and **maxSurge*=20%** ensures full capacity is maintained during the update and rapid scale up.
- **maxUnavailable*=10%** and **maxSurge*=0** performs an update using no extra capacity (an in-place update).

- **maxUnavailable*=10%** and **maxSurge*=10%** scales up and down quickly with some potential for capacity loss.

Generally, if you want fast rollouts, use **maxSurge**. If you have to take into account resource quota and can accept partial unavailability, use **maxUnavailable**.

5.3.1.1. Canary deployments

All Rolling deployments in OpenShift Container Platform are *canary deployments*; a new version (the canary) is tested before all of the old instances are replaced. If the readiness check never succeeds, the canary instance is removed and the DeploymentConfig will be automatically rolled back.

The readiness check is part of the application code and can be as sophisticated as necessary to ensure the new instance is ready to be used. If you must implement more complex checks of the application (such as sending real user workloads to the new instance), consider implementing a Custom deployment or using a blue-green deployment strategy.

5.3.1.2. Creating a Rolling deployment

Rolling deployments are the default type in OpenShift Container Platform. You can create a Rolling deployment using the CLI.

Procedure

1. Create an application based on the example deployment images found in [DockerHub](#):

```
$ oc new-app openshift/deployment-example
```

2. If you have the router installed, make the application available via a route (or use the service IP directly)

```
$ oc expose svc/deployment-example
```

3. Browse to the application at **deployment-example.<project>.<router_domain>** to verify you see the **v1** image.

4. Scale the DeploymentConfig up to three replicas:

```
$ oc scale dc/deployment-example --replicas=3
```

5. Trigger a new deployment automatically by tagging a new version of the example as the **latest** tag:

```
$ oc tag deployment-example:v2 deployment-example:latest
```

6. In your browser, refresh the page until you see the **v2** image.

7. When using the CLI, the following command shows how many Pods are on version 1 and how many are on version 2. In the web console, the Pods are progressively added to v2 and removed from v1:

```
$ oc describe dc deployment-example
```

During the deployment process, the new ReplicationController is incrementally scaled up. After the new Pods are marked as **ready** (by passing their readiness check), the deployment process continues.

If the Pods do not become ready, the process aborts, and the DeploymentConfig rolls back to its previous version.

5.3.2. Recreate strategy

The Recreate strategy has basic rollout behavior and supports lifecycle hooks for injecting code into the deployment process.

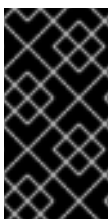
Example Recreate strategy definition

```
strategy:  
  type: Recreate  
  recreateParams: 1  
  pre: {} 2  
  mid: {}  
  post: {}
```

- 1** **recreateParams** are optional.
- 2** **pre**, **mid**, and **post** are lifecycle hooks.

The Recreate strategy:

1. Executes any **pre** lifecycle hook.
2. Scales down the previous deployment to zero.
3. Executes any **mid** lifecycle hook.
4. Scales up the new deployment.
5. Executes any **post** lifecycle hook.



IMPORTANT

During scale up, if the replica count of the deployment is greater than one, the first replica of the deployment will be validated for readiness before fully scaling up the deployment. If the validation of the first replica fails, the deployment will be considered a failure.

When to use a Recreate deployment:

- When you must run migrations or other data transformations before your new code starts.
- When you do not support having new and old versions of your application code running at the same time.
- When you want to use a RWO volume, which is not supported being shared between multiple replicas.

A Recreate deployment incurs downtime because, for a brief period, no instances of your application are running. However, your old code and new code do not run at the same time.

5.3.3. Custom strategy

The Custom strategy allows you to provide your own deployment behavior.

Example Custom strategy definition

```
strategy:
  type: Custom
  customParams:
    image: organization/strategy
    command: [ "command", "arg1" ]
  environment:
    - name: ENV_1
      value: VALUE_1
```

In the above example, the **organization/strategy** container image provides the deployment behavior. The optional **command** array overrides any **CMD** directive specified in the image's **Dockerfile**. The optional environment variables provided are added to the execution environment of the strategy process.

Additionally, OpenShift Container Platform provides the following environment variables to the deployment process:

Environment variable	Description
OPENSIFT_DEPLOYMENT_NAME	The name of the new deployment (a ReplicationController).
OPENSIFT_DEPLOYMENT_NAMESPACE	The name space of the new deployment.

The replica count of the new deployment will initially be zero. The responsibility of the strategy is to make the new deployment active using the logic that best serves the needs of the user.

Alternatively, use **customParams** to inject the custom deployment logic into the existing deployment strategies. Provide a custom shell script logic and call the **openshift-deploy** binary. Users do not have to supply their custom deployer container image; in this case, the default OpenShift Container Platform deployer image is used instead:

```
strategy:
  type: Rolling
  customParams:
    command:
      - /bin/sh
      - -c
      - |
        set -e
        openshift-deploy --until=50%
```

```
echo Halfway there
openshift-deploy
echo Complete
```

This results in following deployment:

```
Started deployment #2
--> Scaling up custom-deployment-2 from 0 to 2, scaling down custom-deployment-1 from 2 to 0
(keep 2 pods available, don't exceed 3 pods)
  Scaling custom-deployment-2 up to 1
--> Reached 50% (currently 50%)
Halfway there
--> Scaling up custom-deployment-2 from 1 to 2, scaling down custom-deployment-1 from 2 to 0
(keep 2 pods available, don't exceed 3 pods)
  Scaling custom-deployment-1 down to 1
  Scaling custom-deployment-2 up to 2
  Scaling custom-deployment-1 down to 0
--> Success
Complete
```

If the custom deployment strategy process requires access to the OpenShift Container Platform API or the Kubernetes API the container that executes the strategy can use the service account token available inside the container for authentication.

5.3.4. Lifecycle hooks

The Rolling and Recreate strategies support *lifecycle hooks*, or deployment hooks, which allow behavior to be injected into the deployment process at predefined points within the strategy:

Example pre lifecycle hook

```
pre:
  failurePolicy: Abort
  execNewPod: {} 1
```

1 **execNewPod** is a Pod-based lifecycle hook.

Every hook has a **failurePolicy**, which defines the action the strategy should take when a hook failure is encountered:

Abort	The deployment process will be considered a failure if the hook fails.
Retry	The hook execution should be retried until it succeeds.
Ignore	Any hook failure should be ignored and the deployment should proceed.

Hooks have a type-specific field that describes how to execute the hook. Currently, Pod-based hooks are the only supported hook type, specified by the **execNewPod** field.

Pod-based lifecycle hook

Pod-based lifecycle hooks execute hook code in a new Pod derived from the template in a DeploymentConfig.

The following simplified example DeploymentConfig uses the Rolling strategy. Triggers and some other minor details are omitted for brevity:

```
kind: DeploymentConfig
apiVersion: v1
metadata:
  name: frontend
spec:
  template:
    metadata:
      labels:
        name: frontend
    spec:
      containers:
        - name: helloworld
          image: openshift/origin-ruby-sample
      replicas: 5
    selector:
      name: frontend
    strategy:
      type: Rolling
      rollingParams:
        pre:
          failurePolicy: Abort
          execNewPod:
            containerName: helloworld ❶
            command: [ "/usr/bin/command", "arg1", "arg2" ] ❷
            env: ❸
              - name: CUSTOM_VAR1
                value: custom_value1
            volumes:
              - data ❹
```

- ❶ The **helloworld** name refers to **spec.template.spec.containers[0].name**.
- ❷ This **command** overrides any **ENTRYPOINT** defined by the **openshift/origin-ruby-sample** image.
- ❸ **env** is an optional set of environment variables for the hook container.
- ❹ **volumes** is an optional set of volume references for the hook container.

In this example, the **pre** hook will be executed in a new Pod using the **openshift/origin-ruby-sample** image from the **helloworld** container. The hook Pod has the following properties:

- The hook command is **/usr/bin/command arg1 arg2**.
- The hook container has the **CUSTOM_VAR1=custom_value1** environment variable.
- The hook failure policy is **Abort**, meaning the deployment process fails if the hook fails.
- The hook Pod inherits the **data** volume from the DeploymentConfig Pod.

5.3.4.1. Setting lifecycle hooks

You can set lifecycle hooks, or deployment hooks, for a DeploymentConfig using the CLI.

Procedure

1. Use the **oc set deployment-hook** command to set the type of hook you want: **--pre**, **--mid**, or **--post**. For example, to set a pre-deployment hook:

```
$ oc set deployment-hook dc/frontend \  
  --pre -c helloworld -e CUSTOM_VAR1=custom_value1 \  
  -v data --failure-policy=abort -- /usr/bin/command arg1 arg2
```

5.4. USING ROUTE-BASED DEPLOYMENT STRATEGIES

Deployment strategies provide a way for the application to evolve. Some strategies use DeploymentConfigs to make changes that are seen by users of all routes that resolve to the application. Other advanced strategies, such as the ones described in this section, use router features in conjunction with DeploymentConfigs to impact specific routes.

The most common route-based strategy is to use a *blue-green deployment*. The new version (the blue version) is brought up for testing and evaluation, while the users still use the stable version (the green version). When ready, the users are switched to the blue version. If a problem arises, you can switch back to the green version.

A common alternative strategy is to use *A/B versions* that are both active at the same time and some users use one version, and some users use the other version. This can be used for experimenting with user interface changes and other features to get user feedback. It can also be used to verify proper operation in a production context where problems impact a limited number of users.

A canary deployment tests the new version but when a problem is detected it quickly falls back to the previous version. This can be done with both of the above strategies.

The route-based deployment strategies do not scale the number of Pods in the services. To maintain desired performance characteristics the deployment configurations might have to be scaled.

5.4.1. Proxy shards and traffic splitting

In production environments, you can precisely control the distribution of traffic that lands on a particular shard. When dealing with large numbers of instances, you can use the relative scale of individual shards to implement percentage based traffic. That combines well with a *proxy shard*, which forwards or splits the traffic it receives to a separate service or application running elsewhere.

In the simplest configuration, the proxy forwards requests unchanged. In more complex setups, you can duplicate the incoming requests and send to both a separate cluster as well as to a local instance of the application, and compare the result. Other patterns include keeping the caches of a DR installation warm, or sampling incoming traffic for analysis purposes.

Any TCP (or UDP) proxy could be run under the desired shard. Use the **oc scale** command to alter the relative number of instances serving requests under the proxy shard. For more complex traffic management, consider customizing the OpenShift Container Platform router with proportional balancing capabilities.

5.4.2. N-1 compatibility

Applications that have new code and old code running at the same time must be careful to ensure that data written by the new code can be read and handled (or gracefully ignored) by the old version of the code. This is sometimes called *schema evolution* and is a complex problem.

This can take many forms: data stored on disk, in a database, in a temporary cache, or that is part of a user's browser session. While most web applications can support rolling deployments, it is important to test and design your application to handle it.

For some applications, the period of time that old code and new code is running side by side is short, so bugs or some failed user transactions are acceptable. For others, the failure pattern may result in the entire application becoming non-functional.

One way to validate N-1 compatibility is to use an A/B deployment: run the old code and new code at the same time in a controlled way in a test environment, and verify that traffic that flows to the new deployment does not cause failures in the old deployment.

5.4.3. Graceful termination

OpenShift Container Platform and Kubernetes give application instances time to shut down before removing them from load balancing rotations. However, applications must ensure they cleanly terminate user connections as well before they exit.

On shutdown, OpenShift Container Platform sends a **TERM** signal to the processes in the container. Application code, on receiving **SIGTERM**, stop accepting new connections. This ensures that load balancers route traffic to other active instances. The application code then waits until all open connections are closed (or gracefully terminate individual connections at the next opportunity) before exiting.

After the graceful termination period expires, a process that has not exited is sent the **KILL** signal, which immediately ends the process. The **terminationGracePeriodSeconds** attribute of a Pod or Pod template controls the graceful termination period (default 30 seconds) and may be customized per application as necessary.

5.4.4. Blue-green deployments

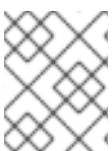
Blue-green deployments involve running two versions of an application at the same time and moving traffic from the in-production version (the green version) to the newer version (the blue version). You can use a Rolling strategy or switch services in a route.

Because many applications depend on persistent data, you must have an application that supports *N-1 compatibility*, which means it shares data and implements live migration between the database, store, or disk by creating two copies of the data layer.

Consider the data used in testing the new version. If it is the production data, a bug in the new version can break the production version.

5.4.4.1. Setting up a blue-green deployment

Blue-green deployments use two DeploymentConfigs. Both are running, and the one in production depends on the service the route specifies, with each DeploymentConfig exposed to a different service.



NOTE

Routes are intended for web (HTTP and HTTPS) traffic, so this technique is best suited for web applications.

You can create a new route to the new version and test it. When ready, change the service in the production route to point to the new service and the new (blue) version is live.

If necessary, you can roll back to the older (green) version by switching the service back to the previous version.

Procedure

1. Create two copies of the example application:

```
$ oc new-app openshift/deployment-example:v1 --name=example-green
$ oc new-app openshift/deployment-example:v2 --name=example-blue
```

This creates two independent application components: one running the **v1** image under the **example-green** service, and one using the **v2** image under the **example-blue** service.

2. Create a route that points to the old service:

```
$ oc expose svc/example-green --name=bluegreen-example
```

3. Browse to the application at **example-green.<project>.<router_domain>** to verify you see the **v1** image.

4. Edit the route and change the service name to **example-blue**:

```
$ oc patch route/bluegreen-example -p '{"spec":{"to":{"name":"example-blue"}}}'
```

5. To verify that the route has changed, refresh the browser until you see the **v2** image.

5.4.5. A/B deployments

The A/B deployment strategy lets you try a new version of the application in a limited way in the production environment. You can specify that the production version gets most of the user requests while a limited fraction of requests go to the new version.

Because you control the portion of requests to each version, as testing progresses you can increase the fraction of requests to the new version and ultimately stop using the previous version. As you adjust the request load on each version, the number of Pods in each service might have to be scaled as well to provide the expected performance.

In addition to upgrading software, you can use this feature to experiment with versions of the user interface. Since some users get the old version and some the new, you can evaluate the user's reaction to the different versions to inform design decisions.

For this to be effective, both the old and new versions must be similar enough that both can run at the same time. This is common with bug fix releases and when new features do not interfere with the old. The versions require N-1 compatibility to properly work together.

OpenShift Container Platform supports N-1 compatibility through the web console as well as the CLI.

5.4.5.1. Load balancing for A/B testing

The user sets up a route with multiple services. Each service handles a version of the application.

Each service is assigned a **weight** and the portion of requests to each service is the **service_weight** divided by the **sum_of_weights**. The **weight** for each service is distributed to the service's endpoints so that the sum of the endpoint **weights** is the service **weight**.

The route can have up to four services. The **weight** for the service can be between **0** and **256**. When the **weight** is **0**, the service does not participate in load-balancing but continues to serve existing persistent connections. When the service **weight** is not **0**, each endpoint has a minimum **weight** of **1**. Because of this, a service with a lot of endpoints can end up with higher **weight** than desired. In this case, reduce the number of Pods to get the desired load balance **weight**.

Procedure

To set up the A/B environment:

1. Create the two applications and give them different names. Each creates a DeploymentConfig. The applications are versions of the same program; one is usually the current production version and the other the proposed new version:

```
$ oc new-app openshift/deployment-example --name=ab-example-a
$ oc new-app openshift/deployment-example --name=ab-example-b
```

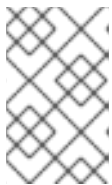
Both applications are deployed and services are created.

2. Make the application available externally via a route. At this point, you can expose either. It can be convenient to expose the current production version first and later modify the route to add the new version.

```
$ oc expose svc/ab-example-a
```

Browse to the application at **ab-example-`<project>`.`<router_domain>`** to verify that you see the desired version.

3. When you deploy the route, the router balances the traffic according to the **weights** specified for the services. At this point, there is a single service with default **weight=1** so all requests go to it. Adding the other service as an **alternateBackends** and adjusting the **weights** brings the A/B setup to life. This can be done by the **oc set route-backends** command or by editing the route. Setting the **oc set route-backend** to **0** means the service does not participate in load-balancing, but continues to serve existing persistent connections.



NOTE

Changes to the route just change the portion of traffic to the various services. You might have to scale the DeploymentConfigs to adjust the number of Pods to handle the anticipated loads.

To edit the route, run:

```
$ oc edit route <route_name>
...
metadata:
  name: route-alternate-service
  annotations:
    haproxy.router.openshift.io/balance: roundrobin
spec:
  host: ab-example.my-project.my-domain
```

```

to:
  kind: Service
  name: ab-example-a
  weight: 10
alternateBackends:
- kind: Service
  name: ab-example-b
  weight: 15
...

```

5.4.5.1.1. Managing weights using the web console

Procedure

1. Navigate to the Route details page (Applications/Routes).
2. Select **Edit** from the Actions menu.
3. Check **Split traffic across multiple services**
4. The **Service Weights** slider sets the percentage of traffic sent to each service. For traffic split between more than two services, the relative weights are specified by integers between 0 and 256 for each service.

Traffic weightings are shown on the **Overview** in the expanded rows of the applications between which traffic is split.

5.4.5.1.2. Managing weights using the CLI

Procedure

1. To manage the services and corresponding weights load balanced by the route, use the **oc set route-backends** command:

```

$ oc set route-backends ROUTENAME \
  [--zero|--equal] [--adjust] SERVICE=WEIGHT[%] [...] [options]

```

For example, the following sets **ab-example-a** as the primary service with **weight=198** and **ab-example-b** as the first alternate service with a **weight=2**:

```

$ oc set route-backends ab-example ab-example-a=198 ab-example-b=2

```

This means 99% of traffic is sent to service **ab-example-a** and 1% to service **ab-example-b**.

This command does not scale the DeploymentConfigs. You might be required to do so to have enough Pods to handle the request load.

2. Run the command with no flags to verify the current configuration:

```

$ oc set route-backends ab-example
NAME           KIND  TO           WEIGHT
routes/ab-example  Service  ab-example-a 198 (99%)
routes/ab-example  Service  ab-example-b  2  (1%)

```

- To alter the weight of an individual service relative to itself or to the primary service, use the **--adjust** flag. Specifying a percentage adjusts the service relative to either the primary or the first alternate (if you specify the primary). If there are other backends, their weights are kept proportional to the changed.

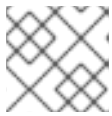
For example:

```
$ oc set route-backends ab-example --adjust ab-example-a=200 ab-example-b=10
$ oc set route-backends ab-example --adjust ab-example-b=5%
$ oc set route-backends ab-example --adjust ab-example-b=+15%
```

The **--equal** flag sets the **weight** of all services to **100**:

```
$ oc set route-backends ab-example --equal
```

The **--zero** flag sets the **weight** of all services to **0**. All requests then return with a 503 error.



NOTE

Not all routers may support multiple or weighted backends.

5.4.5.1.3. One service, multiple DeploymentConfigs

Procedure

- Create a new application, adding a label **ab-example=true** that will be common to all shards:

```
$ oc new-app openshift/deployment-example --name=ab-example-a
```

The application is deployed and a service is created. This is the first shard.

- Make the application available via a route (or use the service IP directly):

```
$ oc expose svc/ab-example-a --name=ab-example
```

- Browse to the application at **ab-example-<project>.<router_domain>** to verify you see the **v1** image.
- Create a second shard based on the same source image and label as the first shard, but with a different tagged version and unique environment variables:

```
$ oc new-app openshift/deployment-example:v2 \
  --name=ab-example-b --labels=ab-example=true \
  SUBTITLE="shard B" COLOR="red"
```

- At this point, both sets of Pods are being served under the route. However, because both browsers (by leaving a connection open) and the router (by default, through a cookie) attempt to preserve your connection to a back-end server, you might not see both shards being returned to you.

To force your browser to one or the other shard:

- Use the **oc scale** command to reduce replicas of **ab-example-a** to **0**.

```
$ oc scale dc/ab-example-a --replicas=0
```

Refresh your browser to show **v2** and **shard B** (in red).

- b. Scale **ab-example-a** to **1** replica and **ab-example-b** to **0**:

```
$ oc scale dc/ab-example-a --replicas=1; oc scale dc/ab-example-b --replicas=0
```

Refresh your browser to show **v1** and **shard A** (in blue).

6. If you trigger a deployment on either shard, only the Pods in that shard are affected. You can trigger a deployment by changing the **SUBTITLE** environment variable in either DeploymentConfig:

```
$ oc edit dc/ab-example-a
```

or

```
$ oc edit dc/ab-example-b
```

CHAPTER 6. CRDS

6.1. EXTENDING THE KUBERNETES API WITH CUSTOM RESOURCE DEFINITIONS

This guide describes how cluster administrators can extend their OpenShift Container Platform cluster by creating and managing Custom Resource Definitions (CRDs).

6.1.1. Custom Resource Definitions

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

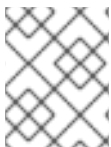
A *Custom Resource Definition* (CRD) object defines a new, unique object **Kind** in the cluster and lets the Kubernetes API server handle its entire lifecycle.

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

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

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

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



NOTE

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

6.1.2. Creating a Custom Resource Definition

To create Custom Resource (CR) objects, cluster administrators must first create a Custom Resource Definition (CRD).

Prerequisites

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

Procedure

To create a CRD:

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

Example YAML file for a CRD

```

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

```

- 1** Use the **apiextensions.k8s.io/v1beta1** API.
- 2** Specify a name for the definition. This must be in the `<plural-name>.<group>` format using the values from the **group** and **plural** fields.
- 3** Specify a group name for the API. An API group is a collection of objects that are logically related. For example, all batch objects like **Job** or **ScheduledJob** could be in the batch API Group (such as `batch.api.example.com`). A good practice is to use a fully-qualified-domain name of your organization.
- 4** Specify a version name to be used in the URL. Each API Group can exist in multiple versions. For example: **v1alpha**, **v1beta**, **v1**.
- 5** Specify whether the custom objects are available to a project (**Namespaced**) or all projects in the cluster (**Cluster**).
- 6** Specify the plural name to use in the URL. The **plural** field is the same as a resource in an API URL.
- 7** Specify a singular name to use as an alias on the CLI and for display.
- 8** Specify the kind of objects that can be created. The type can be in CamelCase.
- 9** Specify a shorter string to match your resource on the CLI.

**NOTE**

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

2. Create the CRD object:

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

A new RESTful API endpoint is created at:

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

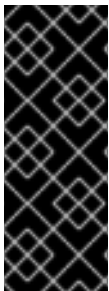
For example, using the example file, the following endpoint is created:

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

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

6.1.3. Creating cluster roles for Custom Resource Definitions

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



IMPORTANT

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

Prerequisites

- Create a CRD.

Procedure

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

Example YAML file for a cluster role definition

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

```

    rbac.authorization.k8s.io/aggregate-to-cluster-reader: "true" 10
  rules:
  - apiGroups: ["stable.example.com"] 11
    resources: ["crontabs"] 12
    verbs: ["get", "list", "watch"] 13

```

- 1 Use the **rbac.authorization.k8s.io/v1** API.
- 2 8 Specify a name for the definition.
- 3 Specify this label to grant permissions to the admin default role.
- 4 Specify this label to grant permissions to the edit default role.
- 5 11 Specify the group name of the CRD.
- 6 12 Specify the plural name of the CRD that these rules apply to.
- 7 13 Specify the verbs that represent the permissions that are granted to the role. For example, apply read and write permissions to the **admin** and **edit** roles and only read permission to the **view** role.
- 9 Specify this label to grant permissions to the **view** default role.
- 10 Specify this label to grant permissions to the **cluster-reader** default role.

2. Create the cluster role:

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

6.1.4. Creating Custom Resources from a file

After a Custom Resource Definition (CRD) has been added to the cluster, Custom Resources (CRs) can be created with the CLI from a file using the CR specification.

Prerequisites

- CRD added to the cluster by a cluster administrator.

Procedure

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

Example YAML file for a CR

```

apiVersion: "stable.example.com/v1" 1
kind: CronTab 2
metadata:
  name: my-new-cron-object 3
  finalizers: 4
  - finalizer.stable.example.com

```



```
spec: 5
  cronSpec: "* * * * /5"
  image: my-awesome-cron-image
```

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

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

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

6.1.5. Inspecting Custom Resources

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

Prerequisites

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

Procedure

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

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

For example:

```
$ oc get crontab

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

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

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

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

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

```

$ oc get ct -o yaml

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

```

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

6.2. MANAGING RESOURCES FROM CUSTOM RESOURCE DEFINITIONS

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

6.2.1. Custom Resource Definitions

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

A *Custom Resource Definition* (CRD) object defines a new, unique object **Kind** in the cluster and lets the Kubernetes API server handle its entire lifecycle.

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

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



NOTE

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

6.2.2. Creating Custom Resources from a file

After a Custom Resource Definition (CRD) has been added to the cluster, Custom Resources (CRs) can be created with the CLI from a file using the CR specification.

Prerequisites

- CRD added to the cluster by a cluster administrator.

Procedure

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

Example YAML file for a CR

```

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

```

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

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

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

6.2.3. Inspecting Custom Resources

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

Prerequisites

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

Procedure

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

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

For example:

```
$ oc get crontab
```

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

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

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

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

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

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

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

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

CHAPTER 7. QUOTAS

7.1. RESOURCE QUOTAS PER PROJECT

A *resource quota*, defined by a ResourceQuota object, provides constraints that limit aggregate resource consumption per project. It can limit the quantity of objects that can be created in a project by type, as well as the total amount of compute resources and storage that may be consumed by resources in that project.

This guide describes how resource quotas work, how cluster administrators can set and manage resource quotas on a per project basis, and how developers and cluster administrators can view them.

7.1.1. Resources managed by quotas

The following describes the set of compute resources and object types that can be managed by a quota.



NOTE

A pod is in a terminal state if **status.phase in (Failed, Succeeded)** is true.

Table 7.1. Compute resources managed by quota

Resource Name	Description
cpu	The sum of CPU requests across all pods in a non-terminal state cannot exceed this value. cpu and requests.cpu are the same value and can be used interchangeably.
memory	The sum of memory requests across all pods in a non-terminal state cannot exceed this value. memory and requests.memory are the same value and can be used interchangeably.
ephemeral-storage	The sum of local ephemeral storage requests across all pods in a non-terminal state cannot exceed this value. ephemeral-storage and requests.ephemeral-storage are the same value and can be used interchangeably. This resource is available only if you enabled the ephemeral storage technology preview. This feature is disabled by default.
requests.cpu	The sum of CPU requests across all pods in a non-terminal state cannot exceed this value. cpu and requests.cpu are the same value and can be used interchangeably.
requests.memory	The sum of memory requests across all pods in a non-terminal state cannot exceed this value. memory and requests.memory are the same value and can be used interchangeably.

Resource Name	Description
requests.ephemeral-storage	The sum of ephemeral storage requests across all pods in a non-terminal state cannot exceed this value. ephemeral-storage and requests.ephemeral-storage are the same value and can be used interchangeably. This resource is available only if you enabled the ephemeral storage technology preview. This feature is disabled by default.
limits.cpu	The sum of CPU limits across all pods in a non-terminal state cannot exceed this value.
limits.memory	The sum of memory limits across all pods in a non-terminal state cannot exceed this value.
limits.ephemeral-storage	The sum of ephemeral storage limits across all pods in a non-terminal state cannot exceed this value. This resource is available only if you enabled the ephemeral storage technology preview. This feature is disabled by default.

Table 7.2. Storage resources managed by quota

Resource Name	Description
requests.storage	The sum of storage requests across all persistent volume claims in any state cannot exceed this value.
persistentvolumeclaims	The total number of persistent volume claims that can exist in the project.
<storage-class-name>.storageclass.storage.k8s.io/requests.storage	The sum of storage requests across all persistent volume claims in any state that have a matching storage class, cannot exceed this value.
<storage-class-name>.storageclass.storage.k8s.io/persistentvolumeclaims	The total number of persistent volume claims with a matching storage class that can exist in the project.

Table 7.3. Object counts managed by quota

Resource Name	Description
pods	The total number of pods in a non-terminal state that can exist in the project.
replicationcontrollers	The total number of ReplicationControllers that can exist in the project.
resourcequotas	The total number of resource quotas that can exist in the project.

Resource Name	Description
services	The total number of services that can exist in the project.
services.loadbalancers	The total number of services of type LoadBalancer that can exist in the project.
services.nodeports	The total number of services of type NodePort that can exist in the project.
secrets	The total number of secrets that can exist in the project.
configmaps	The total number of ConfigMap objects that can exist in the project.
persistentvolumeclaims	The total number of persistent volume claims that can exist in the project.
openshift.io/imagestreams	The total number of imagestreams that can exist in the project.

7.1.2. Quota scopes

Each quota can have an associated set of *scopes*. A quota only measures usage for a resource if it matches the intersection of enumerated scopes.

Adding a scope to a quota restricts the set of resources to which that quota can apply. Specifying a resource outside of the allowed set results in a validation error.

Scope	Description
Terminating	Match pods where spec.activeDeadlineSeconds >= 0 .
NotTerminating	Match pods where spec.activeDeadlineSeconds is nil .
BestEffort	Match pods that have best effort quality of service for either cpu or memory .
NotBestEffort	Match pods that do not have best effort quality of service for cpu and memory .

A **BestEffort** scope restricts a quota to limiting the following resources:

- **pods**

A **Terminating**, **NotTerminating**, and **NotBestEffort** scope restricts a quota to tracking the following resources:

- **pods**

- **memory**
- **requests.memory**
- **limits.memory**
- **cpu**
- **requests.cpu**
- **limits.cpu**
- **ephemeral-storage**
- **requests.ephemeral-storage**
- **limits.ephemeral-storage**



NOTE

Ephemeral storage requests and limits apply only if you enabled the ephemeral storage technology preview. This feature is disabled by default.

7.1.3. Quota enforcement

After a resource quota for a project is first created, the project restricts the ability to create any new resources that may violate a quota constraint until it has calculated updated usage statistics.

After a quota is created and usage statistics are updated, the project accepts the creation of new content. When you create or modify resources, your quota usage is incremented immediately upon the request to create or modify the resource.

When you delete a resource, your quota use is decremented during the next full recalculation of quota statistics for the project. A configurable amount of time determines how long it takes to reduce quota usage statistics to their current observed system value.

If project modifications exceed a quota usage limit, the server denies the action, and an appropriate error message is returned to the user explaining the quota constraint violated, and what their currently observed usage statistics are in the system.

7.1.4. Requests versus limits

When allocating compute resources, each container might specify a request and a limit value each for CPU, memory, and ephemeral storage. Quotas can restrict any of these values.

If the quota has a value specified for **requests.cpu** or **requests.memory**, then it requires that every incoming container make an explicit request for those resources. If the quota has a value specified for **limits.cpu** or **limits.memory**, then it requires that every incoming container specify an explicit limit for those resources.

7.1.5. Sample resource quota definitions

core-object-counts.yaml

```
apiVersion: v1
```



```

kind: ResourceQuota
metadata:
  name: core-object-counts
spec:
  hard:
    configmaps: "10" ❶
    persistentvolumeclaims: "4" ❷
    replicationcontrollers: "20" ❸
    secrets: "10" ❹
    services: "10" ❺
    services.loadbalancers: "2" ❻

```

- ❶ The total number of **ConfigMap** objects that can exist in the project.
- ❷ The total number of persistent volume claims (PVCs) that can exist in the project.
- ❸ The total number of ReplicationControllers that can exist in the project.
- ❹ The total number of secrets that can exist in the project.
- ❺ The total number of services that can exist in the project.
- ❻ The total number of services of type **LoadBalancer** that can exist in the project.

openshift-object-counts.yaml

```

apiVersion: v1
kind: ResourceQuota
metadata:
  name: openshift-object-counts
spec:
  hard:
    openshift.io/imagestreams: "10" ❶

```

- ❶ The total number of imagestreams that can exist in the project.

compute-resources.yaml

```

apiVersion: v1
kind: ResourceQuota
metadata:
  name: compute-resources
spec:
  hard:
    pods: "4" ❶
    requests.cpu: "1" ❷
    requests.memory: 1Gi ❸
    requests.ephemeral-storage: 2Gi ❹
    limits.cpu: "2" ❺
    limits.memory: 2Gi ❻
    limits.ephemeral-storage: 4Gi ❼

```

- 1 The total number of pods in a non-terminal state that can exist in the project.
- 2 Across all pods in a non-terminal state, the sum of CPU requests cannot exceed 1 core.
- 3 Across all pods in a non-terminal state, the sum of memory requests cannot exceed 1Gi.
- 4 Across all pods in a non-terminal state, the sum of ephemeral storage requests cannot exceed 2Gi.
- 5 Across all pods in a non-terminal state, the sum of CPU limits cannot exceed 2 cores.
- 6 Across all pods in a non-terminal state, the sum of memory limits cannot exceed 2Gi.
- 7 Across all pods in a non-terminal state, the sum of ephemeral storage limits cannot exceed 4Gi.

besteffort.yaml

```
apiVersion: v1
kind: ResourceQuota
metadata:
  name: besteffort
spec:
  hard:
    pods: "1" 1
  scopes:
    - BestEffort 2
```

- 1 The total number of pods in a non-terminal state with **BestEffort** quality of service that can exist in the project.
- 2 Restricts the quota to only matching pods that have **BestEffort** quality of service for either memory or CPU.

compute-resources-long-running.yaml

```
apiVersion: v1
kind: ResourceQuota
metadata:
  name: compute-resources-long-running
spec:
  hard:
    pods: "4" 1
    limits.cpu: "4" 2
    limits.memory: "2Gi" 3
    limits.ephemeral-storage: "4Gi" 4
  scopes:
    - NotTerminating 5
```

- 1 The total number of pods in a non-terminal state.
- 2 Across all pods in a non-terminal state, the sum of CPU limits cannot exceed this value.
- 3 Across all pods in a non-terminal state, the sum of memory limits cannot exceed this value.

- 4 Across all pods in a non-terminal state, the sum of ephemeral storage limits cannot exceed this value.
- 5 Restricts the quota to only matching pods where **spec.activeDeadlineSeconds** is set to **nil**. Build pods will fall under **NotTerminating** unless the **RestartNever** policy is applied.

compute-resources-time-bound.yaml

```

apiVersion: v1
kind: ResourceQuota
metadata:
  name: compute-resources-time-bound
spec:
  hard:
    pods: "2" 1
    limits.cpu: "1" 2
    limits.memory: "1Gi" 3
    limits.ephemeral-storage: "1Gi" 4
  scopes:
    - Terminating 5

```

- 1 The total number of pods in a non-terminal state.
- 2 Across all pods in a non-terminal state, the sum of CPU limits cannot exceed this value.
- 3 Across all pods in a non-terminal state, the sum of memory limits cannot exceed this value.
- 4 Across all pods in a non-terminal state, the sum of ephemeral storage limits cannot exceed this value.
- 5 Restricts the quota to only matching pods where **spec.activeDeadlineSeconds** ≥ 0 . For example, this quota would charge for build or deployer pods, but not long running pods like a web server or database.

storage-consumption.yaml

```

apiVersion: v1
kind: ResourceQuota
metadata:
  name: storage-consumption
spec:
  hard:
    persistentvolumeclaims: "10" 1
    requests.storage: "50Gi" 2
    gold.storageclass.storage.k8s.io/requests.storage: "10Gi" 3
    silver.storageclass.storage.k8s.io/requests.storage: "20Gi" 4
    silver.storageclass.storage.k8s.io/persistentvolumeclaims: "5" 5
    bronze.storageclass.storage.k8s.io/requests.storage: "0" 6
    bronze.storageclass.storage.k8s.io/persistentvolumeclaims: "0" 7

```

- 1 The total number of persistent volume claims in a project

- 2 Across all persistent volume claims in a project, the sum of storage requested cannot exceed this value.
- 3 Across all persistent volume claims in a project, the sum of storage requested in the gold storage class cannot exceed this value.
- 4 Across all persistent volume claims in a project, the sum of storage requested in the silver storage class cannot exceed this value.
- 5 Across all persistent volume claims in a project, the total number of claims in the silver storage class cannot exceed this value.
- 6 Across all persistent volume claims in a project, the sum of storage requested in the bronze storage class cannot exceed this value. When this is set to **0**, it means bronze storage class cannot request storage.
- 7 Across all persistent volume claims in a project, the sum of storage requested in the bronze storage class cannot exceed this value. When this is set to **0**, it means bronze storage class cannot create claims.

7.1.6. Creating a quota

You can create a quota to constrain resource usage in a given project.

Procedure

1. Define the quota in a file.
2. Use the file to create the quota and apply it to a project:

```
$ oc create -f <file> [-n <project_name>]
```

For example:

```
$ oc create -f core-object-counts.yaml -n demoproject
```

7.1.6.1. Creating object count quotas

You can create an object count quota for all OpenShift Container Platform standard namespaced resource types, such as **BuildConfig**, and **DeploymentConfig**. An object quota count places a defined quota on all standard namespaced resource types.

When using a resource quota, an object is charged against the quota if it exists in server storage. These types of quotas are useful to protect against exhaustion of storage resources.

Procedure

To configure an object count quota for a resource:

1. Run the following command:

```
$ oc create quota <name> \
  --hard=count/<resource>.<group>=<quota>,count/<resource>.<group>=<quota> 1
```

- 1 **<resource>** is the name of the resource, and **<group>** is the API group, if applicable. Use the **oc api-resources** command for a list of resources and their associated API groups.

For example:

```
$ oc create quota test \
--
hard=count/deployments.extensions=2,count/replicasets.extensions=4,count/pods=3,count/secrets=4
resourcequota "test" created
```

This example limits the listed resources to the hard limit in each project in the cluster.

2. Verify that the quota was created:

```
$ oc describe quota test
Name:                test
Namespace:           quota
Resource              Used Hard
-----
count/deployments.extensions 0  2
count/pods                0  3
count/replicasets.extensions 0  4
count/secrets              0  4
```

7.1.6.2. Setting resource quota for extended resources

Overcommitment of resources is not allowed for extended resources, so you must specify **requests** and **limits** for the same extended resource in a quota. Currently, only quota items with the prefix **requests.** is allowed for extended resources. The following is an example scenario of how to set resource quota for the GPU resource **nvidia.com/gpu**.

Procedure

1. Determine how many GPUs are available on a node in your cluster. For example:

```
# oc describe node ip-172-31-27-209.us-west-2.compute.internal | egrep
'Capacity|Allocatable|gpu'
      openshift.com/gpu-accelerator=true
Capacity:
nvidia.com/gpu: 2
Allocatable:
nvidia.com/gpu: 2
nvidia.com/gpu 0      0
```

In this example, 2 GPUs are available.

2. Set a quota in the namespace **nvidia**. In this example, the quota is **1**:

```
# cat gpu-quota.yaml
apiVersion: v1
kind: ResourceQuota
metadata:
  name: gpu-quota
```

```

namespace: nvidia
spec:
  hard:
    requests.nvidia.com/gpu: 1

```

3. Create the quota:

```

# oc create -f gpu-quota.yaml
resourcequota/gpu-quota created

```

4. Verify that the namespace has the correct quota set:

```

# oc describe quota gpu-quota -n nvidia
Name:          gpu-quota
Namespace:     nvidia
Resource       Used Hard
-----
requests.nvidia.com/gpu 0   1

```

5. Run a pod that asks for a single GPU:

```

# oc create -f gpu-pod.yaml

apiVersion: v1
kind: Pod
metadata:
  generateName: gpu-pod-
  namespace: nvidia
spec:
  restartPolicy: OnFailure
  containers:
  - name: rhel7-gpu-pod
    image: rhel7
    env:
    - name: NVIDIA_VISIBLE_DEVICES
      value: all
    - name: NVIDIA_DRIVER_CAPABILITIES
      value: "compute,utility"
    - name: NVIDIA_REQUIRE_CUDA
      value: "cuda>=5.0"
    command: ["sleep"]
    args: ["infinity"]
  resources:
    limits:
      nvidia.com/gpu: 1

```

6. Verify that the pod is running:

```

# oc get pods
NAME          READY   STATUS    RESTARTS  AGE
gpu-pod-s46h7 1/1     Running   0          1m

```

7. Verify that the quota **Used** counter is correct:

```
# oc describe quota gpu-quota -n nvidia
Name:          gpu-quota
Namespace:     nvidia
Resource       Used Hard
-----
requests.nvidia.com/gpu 1 1
```

- Attempt to create a second GPU pod in the **nvidia** namespace. This is technically available on the node because it has 2 GPUs:

```
# oc create -f gpu-pod.yaml
Error from server (Forbidden): error when creating "gpu-pod.yaml": pods "gpu-pod-f7z2w" is forbidden: exceeded quota: gpu-quota, requested: requests.nvidia.com/gpu=1, used: requests.nvidia.com/gpu=1, limited: requests.nvidia.com/gpu=1
```

This **Forbidden** error message is expected because you have a quota of 1 GPU and this pod tried to allocate a second GPU, which exceeds its quota.

7.1.7. Viewing a quota

You can view usage statistics related to any hard limits defined in a project's quota by navigating in the web console to the project's **Quota** page.

You can also use the CLI to view quota details.

Procedure

- Get the list of quotas defined in the project. For example, for a project called **demoproject**:

```
$ oc get quota -n demoproject
NAME          AGE
besteffort    11m
compute-resources 2m
core-object-counts 29m
```

- Describe the quota you are interested in, for example the **core-object-counts** quota:

```
$ oc describe quota core-object-counts -n demoproject
Name: core-object-counts
Namespace: demoproject
Resource Used Hard
-----
configmaps 3 10
persistentvolumeclaims 0 4
replicationcontrollers 3 20
secrets 9 10
services 2 10
```

7.1.8. Configuring quota synchronization period

When a set of resources are deleted, but before quota usage is restored, a user might encounter problems when attempting to reuse the resources. The synchronization time frame of resources is determined by the **resource-quota-sync-period** setting, which can be configured by a cluster

administrator.

Adjusting the regeneration time can be helpful for creating resources and determining resource usage when automation is used.



NOTE

The **resource-quota-sync-period** setting is designed to balance system performance. Reducing the sync period can result in a heavy load on the master.

Procedure

To configure the quota synchronization period:

1. Edit the Kubernetes controller manager.

```
$ oc edit kubecontrollermanager cluster
```

2. Change the **unsupportedConfigOverrides** field to have the following settings, specifying the amount of time, in seconds, for the **resource-quota-sync-period** field:

```
unsupportedConfigOverrides:
  extendedArguments:
    resource-quota-sync-period:
      - 60s
```

7.2. RESOURCE QUOTAS ACROSS MULTIPLE PROJECTS

A multi-project quota, defined by a ClusterResourceQuota object, allows quotas to be shared across multiple projects. Resources used in each selected project are aggregated and that aggregate is used to limit resources across all the selected projects.

This guide describes how cluster administrators can set and manage resource quotas across multiple projects.

7.2.1. Selecting multiple projects during quota creation

When creating quotas, you can select multiple projects based on annotation selection, label selection, or both.

Procedure

1. To select projects based on annotations, run the following command:

```
$ oc create clusterquota for-user \
  --project-annotation-selector openshift.io/requester=<user_name> \
  --hard pods=10 \
  --hard secrets=20
```

This creates the following ClusterResourceQuota object:

```
apiVersion: v1
kind: ClusterResourceQuota
```



```

metadata:
  name: for-user
spec:
  quota: 1
  hard:
    pods: "10"
    secrets: "20"
  selector:
    annotations: 2
    openshift.io/requester: <user_name>
    labels: null 3
status:
  namespaces: 4
  - namespace: ns-one
    status:
      hard:
        pods: "10"
        secrets: "20"
      used:
        pods: "1"
        secrets: "9"
  total: 5
  hard:
    pods: "10"
    secrets: "20"
  used:
    pods: "1"
    secrets: "9"

```

- 1 The **ResourceQuotaSpec** object that will be enforced over the selected projects.
- 2 A simple key-value selector for annotations.
- 3 A label selector that can be used to select projects.
- 4 A per-namespace map that describes current quota usage in each selected project.
- 5 The aggregate usage across all selected projects.

This multi-project quota document controls all projects requested by **<user_name>** using the default project request endpoint. You are limited to 10 pods and 20 secrets.

2. Similarly, to select projects based on labels, run this command:

```

$ oc create clusterresourcequota for-name \ 1
  --project-label-selector=name=frontend \ 2
  --hard=pods=10 --hard=secrets=20

```

- 1 Both **clusterresourcequota** and **clusterquota** are aliases of the same command. **for-name** is the name of the ClusterResourceQuota object.
- 2 To select projects by label, provide a key-value pair by using the format **--project-label-selector=key=value**.

This creates the following ClusterResourceQuota object definition:

```
apiVersion: v1
kind: ClusterResourceQuota
metadata:
  creationTimestamp: null
  name: for-name
spec:
  quota:
    hard:
      pods: "10"
      secrets: "20"
  selector:
    annotations: null
    labels:
      matchLabels:
        name: frontend
```

7.2.2. Viewing applicable ClusterResourceQuotas

A project administrator is not allowed to create or modify the multi-project quota that limits his or her project, but the administrator is allowed to view the multi-project quota documents that are applied to his or her project. The project administrator can do this via the **AppliedClusterResourceQuota** resource.

Procedure

1. To view quotas applied to a project, run:

```
$ oc describe AppliedClusterResourceQuota
```

For example:

```
Name: for-user
Namespace: <none>
Created: 19 hours ago
Labels: <none>
Annotations: <none>
Label Selector: <null>
AnnotationSelector: map[openshift.io/requester:<user-name>]
Resource Used Hard
----- ---- ----
pods      1   10
secrets   9   20
```

7.2.3. Selection granularity

Because of the locking consideration when claiming quota allocations, the number of active projects selected by a multi-project quota is an important consideration. Selecting more than 100 projects under a single multi-project quota can have detrimental effects on API server responsiveness in those projects.

CHAPTER 8. MONITORING APPLICATION HEALTH

In software systems, components can become unhealthy due to transient issues such as temporary connectivity loss, configuration errors, or problems with external dependencies. OpenShift Container Platform applications have a number of options to detect and handle unhealthy containers.

8.1. UNDERSTANDING HEALTH CHECKS

A probe is a Kubernetes action that periodically performs diagnostics on a running container. Currently, two types of probes exist, each serving a different purpose.

Readiness Probe

A Readiness check determines if the container in which it is scheduled is ready to service requests. If the readiness probe fails a container, the endpoints controller ensures the container has its IP address removed from the endpoints of all services. A readiness probe can be used to signal to the endpoints controller that even though a container is running, it should not receive any traffic from a proxy.

For example, a Readiness check can control which Pods are used. When a Pod is not ready, it is removed.

Liveness Probe

A Liveness checks determines if the container in which it is scheduled is still running. If the liveness probe fails due to a condition such as a deadlock, the kubelet kills the container. The container then responds based on its restart policy.

For example, a liveness probe on a node with a **restartPolicy** of **Always** or **OnFailure** kills and restarts the Container on the node.

Sample Liveness Check

```
apiVersion: v1
kind: Pod
metadata:
  labels:
    test: liveness
  name: liveness-http
spec:
  containers:
  - name: liveness-http
    image: k8s.gcr.io/liveness 1
    args:
    - /server
    livenessProbe: 2
      httpGet: 3
        # host: my-host
        # scheme: HTTPS
        path: /healthz
        port: 8080
        httpHeaders:
        - name: X-Custom-Header
          value: Awesome
        initialDelaySeconds: 15 4
        timeoutSeconds: 1 5
      name: liveness 6
```

-
- 1 Specifies the image to use for the liveness probe.
- 2 Specifies the type of health check.
- 3 Specifies the type of Liveness check:
 - HTTP Checks. Specify **httpGet**.
 - Container Execution Checks. Specify **exec**.
 - TCP Socket Check. Specify **tcpSocket**.
- 4 Specifies the number of seconds before performing the first probe after the container starts.
- 5 Specifies the number of seconds between probes.

Sample Liveness check output wth unhealthy container

```
$ oc describe pod pod1
....

FirstSeen LastSeen  Count  From              SubobjectPath  Type    Reason  Message
-----
37s      37s      1  {default-scheduler}           Normal    Scheduled  Successfully assigned
liveness-exec to worker0
36s      36s      1  {kubelet worker0} spec.containers{liveness} Normal    Pulling    pulling image
"k8s.gcr.io/busybox"
36s      36s      1  {kubelet worker0} spec.containers{liveness} Normal    Pulled     Successfully
pulled image "k8s.gcr.io/busybox"
36s      36s      1  {kubelet worker0} spec.containers{liveness} Normal    Created    Created
container with docker id 86849c15382e; Security:[seccomp=unconfined]
36s      36s      1  {kubelet worker0} spec.containers{liveness} Normal    Started    Started
container with docker id 86849c15382e
2s       2s       1  {kubelet worker0} spec.containers{liveness} Warning   Unhealthy  Liveness
probe failed: cat: can't open '/tmp/healthy': No such file or directory
```

8.1.1. Understanding the types of health checks

Liveness checks and Readiness checks can be configured in three ways:

HTTP Checks

The kubelet uses a web hook to determine the healthiness of the container. The check is deemed successful if the HTTP response code is between 200 and 399.

A HTTP check is ideal for applications that return HTTP status codes when completely initialized.

Container Execution Checks

The kubelet executes a command inside the container. Exiting the check with status 0 is considered a success.

TCP Socket Checks

The kubelet attempts to open a socket to the container. The container is only considered healthy if the check can establish a connection. A TCP socket check is ideal for applications that do not start listening until initialization is complete.

8.2. CONFIGURING HEALTH CHECKS

To configure health checks, create a pod for each type of check you want.

Procedure

To create health checks:

1. Create a Liveness Container Execution Check:
 - a. Create a YAML file similar to the following:

```
apiVersion: v1
kind: Pod
metadata:
  labels:
    test: liveness
    name: liveness-exec
spec:
  containers:
  - args:
    image: k8s.gcr.io/liveness
    livenessProbe:
      exec: ❶
        command: ❷
        - cat
        - /tmp/health
      initialDelaySeconds: 15 ❸
  ...
```

- ❶ Specify a Liveness check and the type of Liveness check.
- ❷ Specify the commands to use in the container.
- ❸ Specify the number of seconds before performing the first probe after the container starts.

- b. Verify the state of the health check pod:

```
$ oc describe pod liveness-exec

Events:
  Type    Reason      Age   From              Message
  ----    -
  Normal  Scheduled   9s   default-scheduler   Successfully assigned
  openshift-logging/liveness-exec to ip-10-0-143-40.ec2.internal
  Normal  Pulling    2s   kubelet, ip-10-0-143-40.ec2.internal   pulling image
  "k8s.gcr.io/liveness"
  Normal  Pulled     1s   kubelet, ip-10-0-143-40.ec2.internal   Successfully pulled image
```

```
"k8s.gcr.io/liveness"
```

```
Normal Created 1s kubelet, ip-10-0-143-40.ec2.internal Created container
Normal Started 1s kubelet, ip-10-0-143-40.ec2.internal Started container
```

NOTE

The **timeoutSeconds** parameter has no effect on the Readiness and Liveness probes for Container Execution Checks. You can implement a timeout inside the probe itself, as OpenShift Container Platform cannot time out on an exec call into the container. One way to implement a timeout in a probe is by using the **timeout** parameter to run your liveness or readiness probe:

```
spec:
  containers:
    livenessProbe:
      exec:
        command:
          - /bin/bash
          - '-c'
          - timeout 60 /opt/eap/bin/livenessProbe.sh 1
      timeoutSeconds: 1
      periodSeconds: 10
      successThreshold: 1
      failureThreshold: 3
```

1

Timeout value and path to the probe script.

c. Create the check:

```
$ oc create -f <file-name>.yaml
```

2. Create a Liveness TCP Socket Check:

a. Create a YAML file similar to the following:

```
apiVersion: v1
kind: Pod
metadata:
  labels:
    test: liveness
  name: liveness-tcp
spec:
  containers:
    - name: container1 1
      image: k8s.gcr.io/liveness
      ports:
        - containerPort: 8080 2
      livenessProbe: 3
        tcpSocket:
          port: 8080
        initialDelaySeconds: 15 4
        timeoutSeconds: 1 5
```

- 1 2 Specify the container name and port for the check to connect to.
- 3 Specify the Liveness health check and the type of Liveness check.
- 4 Specify the number of seconds before performing the first probe after the container starts.
- 5 Specify the number of seconds between probes.

b. Create the check:

```
$ oc create -f <file-name>.yaml
```

3. Create an Readiness HTTP Check:

a. Create a YAML file similar to the following:

```
apiVersion: v1
kind: Pod
metadata:
  labels:
    test: readiness
    name: readiness-http
spec:
  containers:
  - args:
    image: k8s.gcr.io/readiness 1
    readinessProbe: 2
    httpGet:
      # host: my-host 3
      # scheme: HTTPS 4
      path: /healthz
      port: 8080
    initialDelaySeconds: 15 5
    timeoutSeconds: 1 6
```

- 1 Specify the image to use for the liveness probe.
- 2 Specify the Readiness health check and the type of Readiness check.
- 3 Specify a host IP address. When **host** is not defined, the **PodIP** is used.
- 4 Specify **HTTP** or **HTTPS**. When **scheme** is not defined, the **HTTP** scheme is used.
- 5 Specify the number of seconds before performing the first probe after the container starts.
- 6 Specify the number of seconds between probes.

b. Create the check:

```
$ oc create -f <file-name>.yaml
```

CHAPTER 9. IDLING APPLICATIONS

Cluster administrators can idle applications to reduce resource consumption. This is useful when the cluster is deployed on a public cloud where cost is related to resource consumption.

If any scalable resources are not in use, OpenShift Container Platform discovers and idles them by scaling their replicas to **0**. The next time network traffic is directed to the resources, the resources are unidled by scaling up the replicas, and normal operation continues.

Applications are made of services, as well as other scalable resources, such as DeploymentConfigs. The action of idling an application involves idling all associated resources.

9.1. IDLING APPLICATIONS

Idling an application involves finding the scalable resources (deployment configurations, replication controllers, and others) associated with a service. Idling an application finds the service and marks it as idled, scaling down the resources to zero replicas.

You can use the **oc idle** command to idle a single service, or use the **--resource-names-file** option to idle multiple services.

9.1.1. Idling a single service

Procedure

1. To idle a single service, run:

```
$ oc idle <service>
```

9.1.2. Idling multiple services

Idling multiple services is helpful if an application spans across a set of services within a project, or when idling multiple services in conjunction with a script in order to idle multiple applications in bulk within the same project.

Procedure

1. Create a file containing a list of the services, each on their own line.
2. Idle the services using the **--resource-names-file** option:

```
$ oc idle --resource-names-file <filename>
```



NOTE

The **idle** command is limited to a single project. For idling applications across a cluster, run the **idle** command for each project individually.

9.2. UNIDLING APPLICATIONS

Application services become active again when they receive network traffic and are scaled back up their previous state. This includes both traffic to the services and traffic passing through routes.

Applications can also be manually unidled by scaling up the resources.

Procedure

1. To scale up a DeploymentConfig, run:

```
$ oc scale --replicas=1 dc <dc_name>
```



NOTE

Automatic unidling by a router is currently only supported by the default HAProxy router.

CHAPTER 10. PRUNING OBJECTS TO RECLAIM RESOURCES

Over time, API objects created in OpenShift Container Platform can accumulate in the cluster's etcd data store through normal user operations, such as when building and deploying applications.

Cluster administrators can periodically prune older versions of objects from the cluster that are no longer required. For example, by pruning images you can delete older images and layers that are no longer in use, but are still taking up disk space.

10.1. BASIC PRUNING OPERATIONS

The CLI groups prune operations under a common parent command:

```
$ oc adm prune <object_type> <options>
```

This specifies:

- The **<object_type>** to perform the action on, such as **groups, builds, deployments, or images**.
- The **<options>** supported to prune that object type.

10.2. PRUNING GROUPS

To prune groups records from an external provider, administrators can run the following command:

```
$ oc adm prune groups \
  --sync-config=path/to/sync/config [<options>]
```

Table 10.1. Prune groups CLI configuration options

Options	Description
--confirm	Indicate that pruning should occur, instead of performing a dry-run.
--blacklist	Path to the group blacklist file.
--whitelist	Path to the group whitelist file.
--sync-config	Path to the synchronization configuration file.

To see the groups that the prune command deletes:

```
$ oc adm prune groups --sync-file=ldap-sync-config.yaml
```

To perform the prune operation:

```
$ oc adm prune groups --sync-file=ldap-sync-config.yaml --confirm
```

10.3. PRUNING DEPLOYMENTS

In order to prune deployments that are no longer required by the system due to age and status, administrators can run the following command:

```
$ oc adm prune deployments [<options>]
```

Table 10.2. Prune deployments CLI configuration options

Option	Description
--confirm	Indicate that pruning should occur, instead of performing a dry-run.
--orphans	Prune all deployments that no longer have a DeploymentConfig, has status is Complete or Failed , and has a replica count of zero.
--keep-complete=<N>	Per DeploymentConfig, keep the last N deployments that have a status of Complete and replica count of zero. (default 5)
--keep-failed=<N>	Per DeploymentConfig, keep the last N deployments that have a status of Failed and replica count of zero. (default 1)
--keep-younger-than=<duration>	Do not prune any object that is younger than <duration> relative to the current time. (default 60m) Valid units of measurement include nanoseconds (ns), microseconds (us), milliseconds (ms), seconds (s), minutes (m), and hours (h).

To see what a pruning operation would delete:

```
$ oc adm prune deployments --orphans --keep-complete=5 --keep-failed=1 \
  --keep-younger-than=60m
```

To actually perform the prune operation:

```
$ oc adm prune deployments --orphans --keep-complete=5 --keep-failed=1 \
  --keep-younger-than=60m --confirm
```

10.4. PRUNING BUILDS

In order to prune builds that are no longer required by the system due to age and status, administrators can run the following command:

```
$ oc adm prune builds [<options>]
```

Table 10.3. Prune builds CLI configuration options

Option	Description
--confirm	Indicate that pruning should occur, instead of performing a dry-run.

Option	Description
--orphans	Prune all builds whose Build Configuration no longer exists, status is complete, failed, error, or canceled.
--keep-complete=<N>	Per Build Configuration, keep the last N builds whose status is complete (default 5).
--keep-failed=<N>	Per Build Configuration, keep the last N builds whose status is failed, error, or canceled (default 1).
--keep-younger-than=<duration>	Do not prune any object that is younger than <duration> relative to the current time (default 60m).

To see what a pruning operation would delete:

```
$ oc adm prune builds --orphans --keep-complete=5 --keep-failed=1 \
  --keep-younger-than=60m
```

To actually perform the prune operation:

```
$ oc adm prune builds --orphans --keep-complete=5 --keep-failed=1 \
  --keep-younger-than=60m --confirm
```



NOTE

Developers can enable automatic build pruning by modifying their Build Configuration.

Additional resources

- [Performing advanced builds → Pruning builds](#)

10.5. PRUNING IMAGES

In order to prune images that are no longer required by the system due to age, status, or exceed limits, administrators can run the following command:

```
$ oc adm prune images [<options>]
```

Currently, to prune images you must first log in to the CLI as a user with an access token. The user must also have the cluster role **system:image-pruner** or greater (for example, **cluster-admin**).

Pruning images removes data from the integrated registry unless **--prune-registry=false** is used. For this operation to work properly, the registry must be configured with **storage:delete:enabled** set to **true**.

Pruning images with the **--namespace** flag does not remove images, only image streams. Images are non-namespaced resources. Therefore, limiting pruning to a particular namespace makes it impossible to calculate their current usage.

By default, the integrated registry caches blobs metadata to reduce the number of requests to storage,

and increase the speed of processing the request. Pruning does not update the integrated registry cache. Images pushed after pruning that contain pruned layers will be broken, because the pruned layers that have metadata in the cache will not be pushed. Therefore it is necessary to clear the cache after pruning. This can be accomplished by redeploying the registry:

```
# oc patch deployment image-registry -n openshift-image-registry --type=merge --patch="{\"spec\":
{\"template\":{\"metadata\":{\"annotations\":{\"kubectl.kubernetes.io/restartedAt\": \"$(date '+%Y-%m-
%dT%H:%M:%SZ' -u)\"}}}}}"
```

If the integrated registry uses a Redis cache, you must clean the database manually.

oc adm prune images operations require a route for your registry. Registry routes are not created by default. See [Image Registry Operator in OpenShift Container Platform](#) for information on how to create a registry route and see [Exposing the registry](#) for details on how to expose the registry service.

Table 10.4. Prune images CLI configuration options

Option	Description
--all	Include images that were not pushed to the registry, but have been mirrored by pullthrough. This is on by default. To limit the pruning to images that were pushed to the integrated registry, pass --all=false .
--certificate-authority	The path to a certificate authority file to use when communicating with the OpenShift Container Platform-managed registries. Defaults to the certificate authority data from the current user's configuration file. If provided, a secure connection is initiated.
--confirm	Indicate that pruning should occur, instead of performing a dry-run. This requires a valid route to the integrated container image registry. If this command is run outside of the cluster network, the route must be provided using --registry-url .
--force-insecure	Use caution with this option. Allow an insecure connection to the container registry that is hosted via HTTP or has an invalid HTTPS certificate.
--keep-tag-revisions=<N>	For each imagestream, keep up to at most N image revisions per tag (default 3).
--keep-younger-than=<duration>	Do not prune any image that is younger than <duration> relative to the current time. Do not prune any image that is referenced by any other object that is younger than <duration> relative to the current time (default 60m).
--prune-over-size-limit	Prune each image that exceeds the smallest limit defined in the same project. This flag cannot be combined with --keep-tag-revisions nor --keep-younger-than .

Option	Description
--registry-url	The address to use when contacting the registry. The command attempts to use a cluster-internal URL determined from managed images and imagestreams. In case it fails (the registry cannot be resolved or reached), an alternative route that works needs to be provided using this flag. The registry host name can be prefixed by https:// or http:// which enforces particular connection protocol.
--prune-registry	In conjunction with the conditions stipulated by the other options, this option controls whether the data in the registry corresponding to the OpenShift Container Platform image API object is pruned. By default, image pruning processes both the image API objects and corresponding data in the registry. This options is useful when you are only concerned with removing etcd content, possibly to reduce the number of image objects (but are not concerned with cleaning up registry storage) or intend to do that separately by hard pruning the registry, possibly during an appropriate maintenance window for the registry.

10.5.1. Image prune conditions

- Remove any image "managed by OpenShift Container Platform" (images with the annotation **openshift.io/image.managed**) that was created at least **--keep-younger-than** minutes ago and is not currently referenced by:
 - any Pod created less than **--keep-younger-than** minutes ago.
 - any imagestream created less than **--keep-younger-than** minutes ago.
 - any running Pods.
 - any pending Pods.
 - any ReplicationControllers.
 - any DeploymentConfigs.
 - any Build Configurations.
 - any Builds.
 - the **--keep-tag-revisions** most recent items in **stream.status.tags[].items**.
- Remove any image "managed by OpenShift Container Platform" (images with the annotation **openshift.io/image.managed**) that is exceeding the smallest limit defined in the same project and is not currently referenced by:
 - any running Pods.
 - any pending Pods.
 - any ReplicationControllers.
 - any DeploymentConfigs.

- any Build Configurations.
- any Builds.
- There is no support for pruning from external registries.
- When an image is pruned, all references to the image are removed from all imagestreams that have a reference to the image in **status.tags**.
- Image layers that are no longer referenced by any images are removed.



NOTE

The **--prune-over-size-limit** flag cannot be combined with **--keep-tag-revisions** nor **--keep-younger-than** flags. Doing so returns information that this operation is not allowed.

Separating the removal of OpenShift Container Platform image API objects and image data from the Registry by using **--prune-registry=false** followed by hard pruning the registry narrows some timing windows and is safer when compared to trying to prune both through one command. However, timing windows are not completely removed.

For example, you can still create a Pod referencing an image as pruning identifies that image for pruning. You should still keep track of an API Object created during the pruning operations that might reference images, so you can mitigate any references to deleted content.

Also, keep in mind that re-doing the pruning without the **--prune-registry** option or with **--prune-registry=true** does not lead to pruning the associated storage in the image registry for images previously pruned by **--prune-registry=false**. Any images that were pruned with **--prune-registry=false** can only be deleted from registry storage by hard pruning the registry.

10.5.2. Running the image prune operation

Procedure

1. To see what a pruning operation would delete:
 - a. Keeping up to three tag revisions, and keeping resources (images, image streams and Pods) younger than sixty minutes:

```
$ oc adm prune images --keep-tag-revisions=3 --keep-younger-than=60m
```

- b. Pruning every image that exceeds defined limits:

```
$ oc adm prune images --prune-over-size-limit
```

2. To actually perform the prune operation with the options from the previous step:

```
$ oc adm prune images --keep-tag-revisions=3 --keep-younger-than=60m --confirm
```

```
$ oc adm prune images --prune-over-size-limit --confirm
```

10.5.3. Using secure or insecure connections

The secure connection is the preferred and recommended approach. It is done over HTTPS protocol with a mandatory certificate verification. The **prune** command always attempts to use it if possible. If it is not possible, in some cases it can fall-back to insecure connection, which is dangerous. In this case, either certificate verification is skipped or plain HTTP protocol is used.

The fall-back to insecure connection is allowed in the following cases unless **--certificate-authority** is specified:

1. The **prune** command is run with the **--force-insecure** option.
2. The provided **registry-url** is prefixed with the **http://** scheme.
3. The provided **registry-url** is a local-link address or **localhost**.
4. The configuration of the current user allows for an insecure connection. This can be caused by the user either logging in using **--insecure-skip-tls-verify** or choosing the insecure connection when prompted.



IMPORTANT

If the registry is secured by a certificate authority different from the one used by OpenShift Container Platform, it must be specified using the **--certificate-authority** flag. Otherwise, the **prune** command fails with an error.

10.5.4. Image pruning problems

Images not being pruned

If your images keep accumulating and the **prune** command removes just a small portion of what you expect, ensure that you understand the image prune conditions that must apply for an image to be considered a candidate for pruning.

Ensure that images you want removed occur at higher positions in each tag history than your chosen tag revisions threshold. For example, consider an old and obsolete image named **sha:abz**. By running the following command in namespace **N**, where the image is tagged, the image is tagged three times in a single imagestream named **myapp**:

```
$ image_name="sha:abz"
$ oc get is -n N -o go-template='{{range $isi, $is := .items}}{{range $ti, $tag := $is.status.tags}\
  '{{range $ii, $item := $tag.items}}{{if eq $item.image ""${image_name}"\
  $"}}{{$.metadata.name}}:{{$.tag.tag}} at position {{$.ii}} out of {{len $.tag.items}}\n\
  '{{end}}'{{end}}'
myapp:v2 at position 4 out of 5
myapp:v2.1 at position 2 out of 2
myapp:v2.1-may-2016 at position 0 out of 1
```

When default options are used, the image is never pruned because it occurs at position **0** in a history of **myapp:v2.1-may-2016** tag. For an image to be considered for pruning, the administrator must either:

- Specify **--keep-tag-revisions=0** with the **oc adm prune images** command.

CAUTION

This action effectively removes all the tags from all the namespaces with underlying images, unless they are younger or they are referenced by objects younger than the specified threshold.

- Delete all the **istags** where the position is below the revision threshold, which means **myapp:v2.1** and **myapp:v2.1-may-2016**.
- Move the image further in the history, either by running new Builds pushing to the same **istag**, or by tagging other image. Unfortunately, this is not always desirable for old release tags.

Tags having a date or time of a particular image's Build in their names should be avoided, unless the image must be preserved for an undefined amount of time. Such tags tend to have just one image in its history, which effectively prevents them from ever being pruned.

Using a secure connection against insecure registry

If you see a message similar to the following in the output of the **oadm prune images** command, then your registry is not secured and the **oadm prune images** client attempts to use a secure connection:

```
error: error communicating with registry: Get https://172.30.30.30:5000/healthz: http: server gave HTTP response to HTTPS client
```

1. The recommend solution is to secure the registry. Otherwise, you can force the client to use an insecure connection by appending **--force-insecure** to the command, however this is not recommended.

Using an insecure connection against a secured registry

If you see one of the following errors in the output of the **oadm prune images** command, it means that your registry is secured using a certificate signed by a certificate authority other than the one used by **oadm prune images** client for connection verification:

```
error: error communicating with registry: Get http://172.30.30.30:5000/healthz: malformed HTTP response "\x15\x03\x01\x00\x02\x02"
error: error communicating with registry: [Get https://172.30.30.30:5000/healthz: x509: certificate signed by unknown authority, Get http://172.30.30.30:5000/healthz: malformed HTTP response "\x15\x03\x01\x00\x02\x02"]
```

By default, the certificate authority data stored in the user's configuration file are used; the same is true for communication with the master API.

Use the **--certificate-authority** option to provide the right certificate authority for the container image registry server.

Using the wrong certificate authority

The following error means that the certificate authority used to sign the certificate of the secured container image registry is different than the authority used by the client:

```
error: error communicating with registry: Get https://172.30.30.30:5000/: x509: certificate signed by unknown authority
```

Make sure to provide the right one with the flag **--certificate-authority**.

As a workaround, the **--force-insecure** flag can be added instead. However, this is not recommended.

Additional resources

- [Accessing the registry](#)
- [Exposing the registry](#)

10.6. HARD PRUNING THE REGISTRY

The OpenShift Container Registry can accumulate blobs that are not referenced by the OpenShift Container Platform cluster's etcd. The basic pruning images procedure, therefore, is unable to operate on them. These are called *orphaned blobs*.

Orphaned blobs can occur from the following scenarios:

- Manually deleting an image with **oc delete image <sha256:image-id>** command, which only removes the image from etcd, but not from the registry's storage.
- Pushing to the registry initiated by **docker** daemon failures, which causes some blobs to get uploaded, but the image manifest (which is uploaded as the very last component) does not. All unique image blobs become orphans.
- OpenShift Container Platform refusing an image because of quota restrictions.
- The standard image pruner deleting an image manifest, but is interrupted before it deletes the related blobs.
- A bug in the registry pruner, which fails to remove the intended blobs, causing the image objects referencing them to be removed and the blobs becoming orphans.

Hard pruning the registry, a separate procedure from basic image pruning, allows cluster administrators to remove orphaned blobs. You should hard prune if you are running out of storage space in your OpenShift Container Registry and believe you have orphaned blobs.

This should be an infrequent operation and is necessary only when you have evidence that significant numbers of new orphans have been created. Otherwise, you can perform standard image pruning at regular intervals, for example, once a day (depending on the number of images being created).

Procedure

To hard prune orphaned blobs from the registry:

1. **Log in.**

Log in to the cluster with the CLI as a user with an access token.

2. **Run a basic image prune**

Basic image pruning removes additional images that are no longer needed. The hard prune does not remove images on its own. It only removes blobs stored in the registry storage. Therefore, you should run this just before the hard prune.

3. **Switch the registry to read-only mode.**

If the registry is not running in read-only mode, any pushes happening at the same time as the prune will either:

- fail and cause new orphans, or
- succeed although the images cannot be pulled (because some of the referenced blobs were deleted).

Pushes will not succeed until the registry is switched back to read-write mode. Therefore, the hard prune must be carefully scheduled.

To switch the registry to read-only mode:

- a. Set the following environment variable:

```
$ oc set env -n default \
  dc/docker-registry \
  'REGISTRY_STORAGE_MAINTENANCE_READONLY={"enabled":true}'
```

- b. By default, the registry automatically redeploys when the previous step completes; wait for the redeployment to complete before continuing. However, if you have disabled these triggers, you must manually redeploy the registry so that the new environment variables are picked up:

```
$ oc rollout -n default \
  latest dc/docker-registry
```

4. Add the **system:image-pruner** role.

The service account used to run the registry instances requires additional permissions in order to list some resources.

- a. Get the service account name:

```
$ service_account=$(oc get -n default \
  -o jsonpath='{$system:serviceaccount:{.metadata.namespace}:
  {spec.template.spec.serviceAccountName}}\n' \
  dc/docker-registry)
```

- b. Add the **system:image-pruner** cluster role to the service account:

```
$ oc adm policy add-cluster-role-to-user \
  system:image-pruner \
  ${service_account}
```

5. (Optional) Run the pruner in dry-run mode.

To see how many blobs would be removed, run the hard pruner in dry-run mode. No changes are actually made:

```
$ oc -n default \
  exec -i -t "$(oc -n default get pods -l deploymentconfig=docker-registry \
  -o jsonpath='{$.items[0].metadata.name}\n')" \
  -- /usr/bin/dockerregistry -prune=check
```

Alternatively, to get the exact paths for the prune candidates, increase the logging level:

```
$ oc -n default \
  exec "$(oc -n default get pods -l deploymentconfig=docker-registry \
  -o jsonpath='{$.items[0].metadata.name}\n')" \
  -- /bin/sh \
  -c 'REGISTRY_LOG_LEVEL=info /usr/bin/dockerregistry -prune=check'
```

Truncated sample output

```
$ oc exec docker-registry-3-vhndw \
  -- /bin/sh -c 'REGISTRY_LOG_LEVEL=info /usr/bin/dockerregistry -prune=check'
```

```

time="2017-06-22T11:50:25.066156047Z" level=info msg="start prune (dry-run mode)"
distribution_version="v2.4.1+unknown" kubernetes_version=v1.6.1+${Format:%h$}
openshift_version=unknown
time="2017-06-22T11:50:25.092257421Z" level=info msg="Would delete blob:
sha256:00043a2a5e384f6b59ab17e2c3d3a3d0a7de01b2cabeb606243e468acc663fa5"
go.version=go1.7.5 instance.id=b097121c-a864-4e0c-ad6c-cc25f8fdf5a6
time="2017-06-22T11:50:25.092395621Z" level=info msg="Would delete blob:
sha256:0022d49612807cb348cab562c072ef34d756adfe0100a61952cbcb87ee6578a"
go.version=go1.7.5 instance.id=b097121c-a864-4e0c-ad6c-cc25f8fdf5a6
time="2017-06-22T11:50:25.092492183Z" level=info msg="Would delete blob:
sha256:0029dd4228961086707e53b881e25eba0564fa80033fbbb2e27847a28d16a37c"
go.version=go1.7.5 instance.id=b097121c-a864-4e0c-ad6c-cc25f8fdf5a6
time="2017-06-22T11:50:26.673946639Z" level=info msg="Would delete blob:
sha256:ff7664dfc213d6cc60fd5c5f5bb00a7bf4a687e18e1df12d349a1d07b2cf7663"
go.version=go1.7.5 instance.id=b097121c-a864-4e0c-ad6c-cc25f8fdf5a6
time="2017-06-22T11:50:26.674024531Z" level=info msg="Would delete blob:
sha256:ff7a933178ccd931f4b5f40f9f19a65be5eeec207e4fad2a5bafd28afbef57e"
go.version=go1.7.5 instance.id=b097121c-a864-4e0c-ad6c-cc25f8fdf5a6
time="2017-06-22T11:50:26.674675469Z" level=info msg="Would delete blob:
sha256:ff9b8956794b426cc80bb49a604a0b24a1553aae96b930c6919a6675db3d5e06"
go.version=go1.7.5 instance.id=b097121c-a864-4e0c-ad6c-cc25f8fdf5a6
...
Would delete 13374 blobs
Would free up 2.835 GiB of disk space
Use -prune=delete to actually delete the data

```

6. Run the hard prune.

Execute the following command inside one running instance of a **docker-registry** pod to run the hard prune:

```

$ oc -n default \
  exec -i -t "$(oc -n default get pods -l deploymentconfig=docker-registry -o
  jsonpath=${.items[0].metadata.name})" \
  -- /usr/bin/dockerregistry -prune=delete

```

Sample output

```

$ oc exec docker-registry-3-vhndw \
  -- /usr/bin/dockerregistry -prune=delete

Deleted 13374 blobs
Freed up 2.835 GiB of disk space

```

7. Switch the registry back to read-write mode.

After the prune is finished, the registry can be switched back to read-write mode by executing:

```

$ oc set env -n default dc/docker-registry
REGISTRY_STORAGE_MAINTENANCE_READONLY-

```

10.7. PRUNING CRON JOBS

Cron jobs can perform pruning of successful jobs, but might not properly handle failed jobs. Therefore, the cluster administrator should perform regular cleanup of jobs manually. They should also restrict the

access to cron jobs to a small group of trusted users and set appropriate quota to prevent the cron job from creating too many jobs and pods.

Additional resources

- [Running tasks in pods using jobs](#)
- [Resource quotas across multiple projects](#)
- [Using RBAC to define and apply permissions](#)

CHAPTER 11. OPERATOR SDK

11.1. GETTING STARTED WITH THE OPERATOR SDK

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

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



NOTE

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

11.1.1. Architecture of the Operator SDK

The [Operator Framework](#) is an open source toolkit to manage Kubernetes native applications, called *Operators*, in an effective, automated, and scalable way. Operators take advantage of Kubernetes' extensibility to deliver the automation advantages of cloud services like provisioning, scaling, and backup and restore, while being able to run anywhere that Kubernetes can run.

Operators make it easy to manage complex, stateful applications on top of Kubernetes. However, writing an Operator today can be difficult because of challenges such as using low-level APIs, writing boilerplate, and a lack of modularity, which leads to duplication.

The Operator SDK is a framework designed to make writing Operators easier by providing:

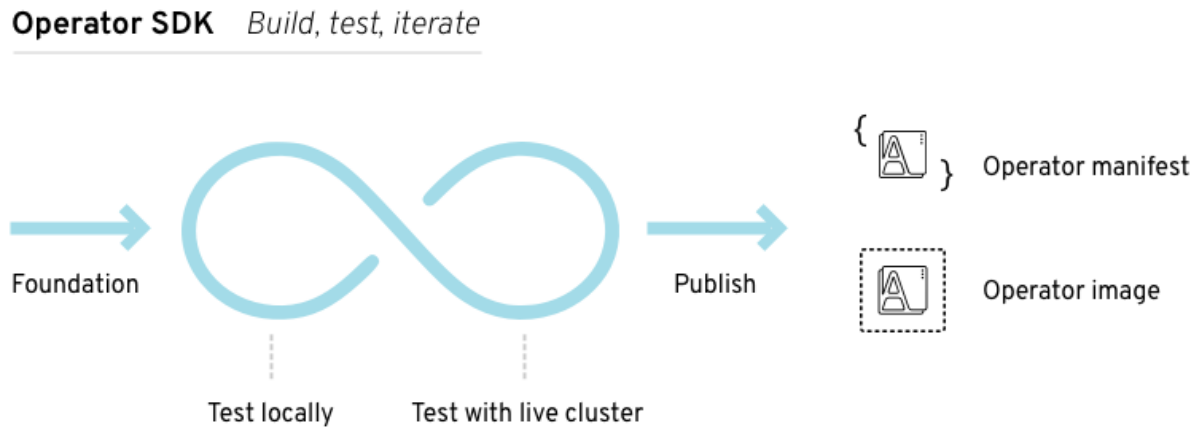
- High-level APIs and abstractions to write the operational logic more intuitively
- Tools for scaffolding and code generation to quickly bootstrap a new project
- Extensions to cover common Operator use cases

11.1.1.1. Workflow

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

1. Create a new Operator project using the Operator SDK command line interface (CLI).
2. Define new resource APIs by adding Custom Resource Definitions (CRDs).
3. Specify resources to watch using the Operator SDK API.
4. Define the Operator reconciling logic in a designated handler and use the Operator SDK API to interact with resources.
5. Use the Operator SDK CLI to build and generate the Operator deployment manifests.

Figure 11.1. Operator SDK workflow



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

11.1.1.2. Manager file

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

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

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

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

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

11.1.1.3. Prometheus Operator support

[Prometheus](#) is an open-source systems monitoring and alerting toolkit. The Prometheus Operator creates, configures, and manages Prometheus clusters running on Kubernetes-based clusters, such as OpenShift Container Platform.

Helper functions exist in the Operator SDK by default to automatically set up metrics in any generated Go-based Operator for use on clusters where the Prometheus Operator is deployed.

11.1.2. Installing the Operator SDK CLI

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

**NOTE**

This guide uses [minikube](#) v0.25.0+ as the local Kubernetes cluster and [Quay.io](#) for the public registry.

11.1.2.1. Installing from GitHub release

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

Prerequisites

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

Procedure

1. Set the release version variable:

```
RELEASE_VERSION=v0.8.0
```

2. Download the release binary.

- For Linux:

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

- For macOS:

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

3. Verify the downloaded release binary.

- a. Download the provided ASC file.

- For Linux:

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

- For macOS:

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


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

- For Linux:

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

- For macOS:

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

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

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

- 1 RSA key string.

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

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

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

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

- For Linux:

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

- For macOS:

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

5. Verify that the CLI tool was installed correctly:

```
$ operator-sdk version
```

11.1.2.2. Installing from Homebrew

You can install the SDK CLI using Homebrew.

Prerequisites

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

Procedure

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

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

2. Verify that the CLI tool was installed correctly:

```
$ operator-sdk version
```

11.1.2.3. Compiling and installing from source

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

Prerequisites

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

Procedure

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

```
$ mkdir -p $GOPATH/src/github.com/operator-framework
$ cd $GOPATH/src/github.com/operator-framework
$ git clone https://github.com/operator-framework/operator-sdk
$ cd operator-sdk
```

2. Check out the desired release branch:

-

```
$ git checkout master
```

3. Compile and install the SDK CLI:

```
$ make dep
$ make install
```

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

4. Verify that the CLI tool was installed correctly:

```
$ operator-sdk version
```

11.1.3. Building a Go-based Memcached Operator using the Operator SDK

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

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

Prerequisites

- Operator SDK CLI installed on the development workstation
- Operator Lifecycle Manager (OLM) installed on a Kubernetes-based cluster (v1.8 or above to support the **apps/v1beta2** API group), for example OpenShift Container Platform 4.1
- Access to the cluster using an account with **cluster-admin** permissions
- OpenShift CLI (**oc**) v4.1+ installed

Procedure

1. **Create a new project.**

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

```
$ mkdir -p $GOPATH/src/github.com/example-inc/
$ cd $GOPATH/src/github.com/example-inc/
$ operator-sdk new memcached-operator --dep-manager dep
$ cd memcached-operator
```

2. **Add a new Custom Resource Definition (CRD).**

- a. Use the CLI to add a new CRD API called **Memcached**, with **APIVersion** set to **cache.example.com/v1alpha1** and **Kind** set to **Memcached**:

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

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

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

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

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

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

3. Add a new Controller.

- a. Add a new Controller to the project to watch and reconcile the Memcached resource:

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

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

- b. For this example, replace the generated controller file **pkg/controller/memcached/memcached_controller.go** with the [example implementation](#). The example controller executes the following reconciliation logic for each **Memcached** CR:

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

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

- c. Inspect the Controller implementation at the **pkg/controller/memcached/memcached_controller.go** file to see how the Controller watches resources.

The first watch is for the Memcached type as the primary resource. For each Add, Update, or Delete event, the reconcile loop is sent a reconcile **Request** (a **<namespace>:<name>** key) for that Memcached object:

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

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

secondary resource:

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

- d. Every Controller has a Reconciler object with a **Reconcile()** method that implements the reconcile loop. The reconcile loop is passed the **Request** argument which is a **<namespace>:<name>** key used to lookup the primary resource object, Memcached, from the cache:

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

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

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

4. Build and run the Operator.

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

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

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

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

Choose one of the following methods.

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

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

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

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

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

- C. Ensure you have an account on [Quay.io](https://quay.io) for the next step, or substitute your preferred container registry. On the registry, [create a new public image](#) repository named **memcached-operator**.

- D. Push the image to the registry:

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

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

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

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

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

- ii. *Option B*: Running locally outside the cluster.

This method is preferred during development cycle to deploy and test faster.

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

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

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

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

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

```
$ cat deploy/crds/cache_v1alpha1_memcached_cr.yaml
apiVersion: "cache.example.com/v1alpha1"
kind: "Memcached"
metadata:
  name: "example-memcached"
spec:
  size: 3

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

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

```
$ oc get deployment
NAME                DESIRED  CURRENT  UP-TO-DATE  AVAILABLE  AGE
memcached-operator  1        1        1           1          2m
```

```
example-memcached 3 3 3 3 1m
```

- c. Check the Pods and CR status to confirm the status is updated with the **memcached** Pod names:

```
$ oc get pods
NAME                                READY  STATUS   RESTARTS  AGE
example-memcached-6fd7c98d8-7dqdr  1/1    Running  0         1m
example-memcached-6fd7c98d8-g5k7v  1/1    Running  0         1m
example-memcached-6fd7c98d8-m7vn7  1/1    Running  0         1m
memcached-operator-7cc7cfd86-vvjpk  1/1    Running  0         2m

$ oc get memcached/example-memcached -o yaml
apiVersion: cache.example.com/v1alpha1
kind: Memcached
metadata:
  clusterName: ""
  creationTimestamp: 2018-03-31T22:51:08Z
  generation: 0
  name: example-memcached
  namespace: default
  resourceVersion: "245453"
  selfLink:
/apis/cache.example.com/v1alpha1/namespaces/default/memcacheds/example-memcached
  uid: 0026cc97-3536-11e8-bd83-0800274106a1
spec:
  size: 3
status:
  nodes:
  - example-memcached-6fd7c98d8-7dqdr
  - example-memcached-6fd7c98d8-g5k7v
  - example-memcached-6fd7c98d8-m7vn7
```

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

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

```
$ cat deploy/crds/cache_v1alpha1_memcached_cr.yaml
apiVersion: "cache.example.com/v1alpha1"
kind: "Memcached"
metadata:
  name: "example-memcached"
spec:
  size: 4
```

- b. Apply the change:

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

- c. Confirm that the Operator changes the Deployment size:

```
$ oc get deployment
NAME          DESIRED  CURRENT  UP-TO-DATE  AVAILABLE  AGE
example-memcached  4        4        4           4          5m
```

7. Clean up the resources:

```
$ oc delete -f deploy/crds/cache_v1alpha1_memcached_cr.yaml
$ oc delete -f deploy/crds/cache_v1alpha1_memcached_crd.yaml
$ oc delete -f deploy/operator.yaml
$ oc delete -f deploy/role.yaml
$ oc delete -f deploy/role_binding.yaml
$ oc delete -f deploy/service_account.yaml
```

11.1.4. Managing a Memcached Operator using the Operator Lifecycle Manager

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

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

Prerequisites

- OLM installed on a Kubernetes-based cluster (v1.8 or above to support the **apps/v1beta2** API group), for example OpenShift Container Platform 4.1 Preview OLM enabled
- Memcached Operator built

Procedure

1. Generate an Operator manifest.

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

You can use the following command to generate CSV manifests:

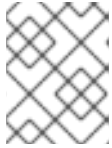
```
$ operator-sdk olm-catalog gen-csv --csv-version 0.0.1
```



NOTE

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

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

**NOTE**

See [Building a CSV for the Operator Framework](#) for more information on manually defining a manifest file.

2. Deploy the Operator.

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

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

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

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

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

- c. Create the role, role binding, and service account to grant resource permissions to the Operator, and the Custom Resource Definition (CRD) to create the Memcached type that the Operator manages:

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

**NOTE**

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

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

3. Create an application instance.

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

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

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

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

$ oc get Memcached
NAME                AGE
memcached-for-drupal 22s
memcached-for-wordpress 27s

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

4. Update an application.

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

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

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

11.1.5. Additional resources

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

11.2. CREATING ANSIBLE-BASED OPERATORS

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

11.2.1. Ansible support in the Operator SDK

The [Operator Framework](#) is an open source toolkit to manage Kubernetes native applications, called *Operators*, in an effective, automated, and scalable way. This framework includes the Operator SDK, which assists developers in bootstrapping and building an Operator based on their expertise without requiring knowledge of Kubernetes API complexities.

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

11.2.1.1. Custom Resource files

Operators use the Kubernetes' extension mechanism, Custom Resource Definitions (CRDs), so your Custom Resource (CR) looks and acts just like the built-in, native Kubernetes objects.

The CR file format is a Kubernetes resource file. The object has mandatory and optional fields:

Table 11.1. Custom Resource fields

Field	Description
apiVersion	Version of the CR to be created.
kind	Kind of the CR to be created.
metadata	Kubernetes-specific metadata to be created.
spec (optional)	Key-value list of variables which are passed to Ansible. This field is empty by default.
status	Summarizes the current state of the object. For Ansible-based Operators, the status subresource is enabled for CRDs and managed by the k8s_status Ansible module by default, which includes condition information to the CR's status .
annotations	Kubernetes-specific annotations to be appended to the CR.

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

Table 11.2. Ansible-based Operator annotations

Annotation	Description
ansible.operator-sdk/reconcile-period	Specifies the reconciliation interval for the CR. This value is parsed using the standard Golang package time . Specifically, ParseDuration is used which applies the default suffix of s , giving the value in seconds.

Example Ansible-based Operator annotation

```

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

```

11.2.1.2. Watches file

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

Table 11.3. Watches file mappings

Field	Description
group	Group of CR to watch.
version	Version of CR to watch.
kind	Kind of CR to watch
role (default)	Path to the Ansible role added to the container. For example, if your roles directory is at /opt/ansible/roles/ and your role is named busybox , this value would be /opt/ansible/roles/busybox . This field is mutually exclusive with the playbook field.
playbook	Path to the Ansible playbook added to the container. This playbook is expected to be simply a way to call roles. This field is mutually exclusive with the role field.
reconcilePeriod (optional)	The reconciliation interval, how often the role or playbook is run, for a given CR.
manageStatus (optional)	When set to true (default), the Operator manages the status of the CR generically. When set to false , the status of the CR is managed elsewhere, by the specified role or playbook or in a separate controller.

Example Watches file

```

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

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

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

```

- 1** Simple example mapping **Foo** to the **Foo** role.
- 2** Simple example mapping **Bar** to a playbook.
- 3** More complex example for the **Baz** kind. Disables re-queuing and managing the CR status in the playbook.

11.2.1.2.1. Advanced options

Advanced features can be enabled by adding them to your Watches file per GVK (group, version, and kind). They can go below the **group**, **version**, **kind** and **playbook** or **role** fields.

Some features can be overridden per resource using an annotation on that Custom Resource (CR). The options that can be overridden have the annotation specified below.

Table 11.4. Advanced Watches file options

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

Feature	YAML key	Description	Annotation for override	Default value
Watch cluster-scoped resources	watchClusterScopedResources	Allows the Operator to watch cluster-scoped resources that are created by Ansible.		false
Max runner artifacts	maxRunnerArtifacts	Manages the number of artifact directories that Ansible Runner keeps in the Operator container for each individual resource.	ansible.operator-sdk/max-runner-artifacts	20

Example Watches file with advanced options

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

11.2.1.3. Extra variables sent to Ansible

Extra variables can be sent to Ansible, which are then managed by the Operator. The **spec** section of the Custom Resource (CR) passes along the key-value pairs as extra variables. This is equivalent to extra variables passed in to the **ansible-playbook** command.

The Operator also passes along additional variables under the **meta** field for the name of the CR and the namespace of the CR.

For the following CR example:

```
apiVersion: "app.example.com/v1alpha1"
kind: "Database"
metadata:
  name: "example"
spec:
  message: "Hello world 2"
  newParameter: "newParam"
```

The structure passed to Ansible as extra variables is:

```
{ "meta": {
  "name": "<cr_name>",
  "namespace": "<cr_namespace>",
```

```

    },
    "message": "Hello world 2",
    "new_parameter": "newParam",
    "_app_example_com_database": {
      <full_crd>
    },
  }
}

```

The **message** and **newParameter** fields are set in the top level as extra variables, and **meta** provides the relevant metadata for the CR as defined in the Operator. The **meta** fields can be accessed using dot notation in Ansible, for example:

```

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

```

11.2.1.4. Ansible Runner directory

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

Additional resources

- To learn more about the **runner** directory, see the [Ansible Runner documentation](#).

11.2.2. Installing the Operator SDK CLI

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



NOTE

This guide uses [minikube](#) v0.25.0+ as the local Kubernetes cluster and [Quay.io](#) for the public registry.

11.2.2.1. Installing from GitHub release

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

Prerequisites

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

Procedure

1. Set the release version variable:

```
RELEASE_VERSION=v0.8.0
```

2. Download the release binary.

- For Linux:

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

- For macOS:

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

3. Verify the downloaded release binary.

a. Download the provided ASC file.

- For Linux:

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

- For macOS:

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

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

- For Linux:

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

- For macOS:

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

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

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

- 1** RSA key string.

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

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

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

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

- For Linux:

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

- For macOS:

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

5. Verify that the CLI tool was installed correctly:

```
$ operator-sdk version
```

11.2.2.2. Installing from Homebrew

You can install the SDK CLI using Homebrew.

Prerequisites

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

Procedure

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

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

2. Verify that the CLI tool was installed correctly:

```
$ operator-sdk version
```

11.2.2.3. Compiling and installing from source

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

Prerequisites

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

Procedure

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

```
$ mkdir -p $GOPATH/src/github.com/operator-framework
$ cd $GOPATH/src/github.com/operator-framework
$ git clone https://github.com/operator-framework/operator-sdk
$ cd operator-sdk
```

2. Check out the desired release branch:

```
$ git checkout master
```

3. Compile and install the SDK CLI:

```
$ make dep
$ make install
```

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

4. Verify that the CLI tool was installed correctly:

```
$ operator-sdk version
```

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

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

Prerequisites

- Operator SDK CLI installed on the development workstation

- Access to a Kubernetes-based cluster v1.11.3+ (for example OpenShift Container Platform 4.1) using an account with **cluster-admin** permissions
- OpenShift CLI (**oc**) v4.1+ installed
- **ansible** v2.6.0+
- **ansible-runner** v1.1.0+
- **ansible-runner-http** v1.0.0+

Procedure

1. **Create a new Operator project**, either namespace-scoped or cluster-scoped, using the **operator-sdk new** command. Choose one of the following:
 - a. A *namespace-scoped Operator* (the default) watches and manages resources in a single namespace. Namespace-scoped operators are preferred because of their flexibility. They enable decoupled upgrades, namespace isolation for failures and monitoring, and differing API definitions.

To create a new Ansible-based, namespace-scoped **memcached-operator** project and change to its directory, use the following commands:

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

This creates the **memcached-operator** project specifically for watching the Memcached resource with APIVersion **example.com/v1alpha1** and Kind **Memcached**.

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

To create your **memcached-operator** project to be cluster-scoped and change to its directory, use the following commands:

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

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

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

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

2. Customize the Operator logic.

For this example, the **memcached-operator** executes the following reconciliation logic for each **Memcached** Custom Resource (CR):

- Create a **memcached** Deployment if it does not exist.
- Ensure that the Deployment size is the same as specified by the **Memcached** CR.

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

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

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

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

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

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

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

3. Build the Memcached Ansible role.

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

- Define the Memcached spec.**

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

TIP

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

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

```
size: 1
```

b. **Define the Memcached Deployment.**

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

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

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

```
- name: start memcached
  k8s:
    definition:
      kind: Deployment
      apiVersion: apps/v1
      metadata:
        name: '{{ meta.name }}-memcached'
        namespace: '{{ meta.namespace }}'
      spec:
        replicas: '{{size}}'
        selector:
          matchLabels:
            app: memcached
        template:
          metadata:
            labels:
              app: memcached
          spec:
            containers:
              - name: memcached
                command:
                  - memcached
                  - -m=64
                  - -o
                  - modern
                  - -v
                image: "docker.io/memcached:1.4.36-alpine"
            ports:
              - containerPort: 11211
```



NOTE

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

4. Deploy the CRD.

Before running the Operator, Kubernetes needs to know about the new Custom Resource Definition (CRD) the Operator will be watching. Deploy the **Memcached** CRD:

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

5. Build and run the Operator.

There are two ways to build and run the Operator:

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

Choose one of the following methods:

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

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

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

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

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

- iii. If you created your Operator using the **--cluster-scoped=true** flag, update the service account namespace in the generated **ClusterRoleBinding** to match where you are deploying your Operator:

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

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

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

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

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

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

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

- b. **Run outside the cluster.** This method is preferred during the development cycle to speed up deployment and testing.

Ensure that Ansible Runner and Ansible Runner HTTP Plug-in are installed or else you will see unexpected errors from Ansible Runner when a CR is created.

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

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

```
$ operator-sdk up local
```

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

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

6. Create a Memcached CR.

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

```
$ cat deploy/crds/cache_v1alpha1_memcached_cr.yaml
apiVersion: "cache.example.com/v1alpha1"
kind: "Memcached"
metadata:
  name: "example-memcached"
spec:
  size: 3

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

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

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

- c. Check the Pods to confirm three replicas were created:

```
$ oc get pods
NAME                READY  STATUS  RESTARTS  AGE
example-memcached-6fd7c98d8-7dqdr  1/1    Running  0         1m
example-memcached-6fd7c98d8-g5k7v  1/1    Running  0         1m
example-memcached-6fd7c98d8-m7vn7  1/1    Running  0         1m
memcached-operator-7cc7cfd86-vvjgk  1/1    Running  0         2m
```

7. Update the size.

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

```
$ cat deploy/crds/cache_v1alpha1_memcached_cr.yaml
apiVersion: "cache.example.com/v1alpha1"
kind: "Memcached"
metadata:
  name: "example-memcached"
spec:
  size: 4

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

- b. Confirm that the Operator changes the Deployment size:

```
$ oc get deployment
NAME             DESIRED  CURRENT  UP-TO-DATE  AVAILABLE  AGE
example-memcached  4        4        4           4          5m
```

8. Clean up the resources:

```
$ oc delete -f deploy/crds/cache_v1alpha1_memcached_cr.yaml
$ oc delete -f deploy/operator.yaml
$ oc delete -f deploy/role_binding.yaml
$ oc delete -f deploy/role.yaml
$ oc delete -f deploy/service_account.yaml
$ oc delete -f deploy/crds/cache_v1alpha1_memcached_crd.yaml
```

11.2.4. Managing application lifecycle using the k8s Ansible module

To manage the lifecycle of your application on Kubernetes using Ansible, you can use the [k8s Ansible module](#). This Ansible module allows a developer to either leverage their existing Kubernetes resource files (written in YAML) or express the lifecycle management in native Ansible.

One of the biggest benefits of using Ansible in conjunction with existing Kubernetes resource files is the ability to use Jinja templating so that you can customize resources with the simplicity of a few variables in Ansible.

This section goes into detail on usage of the **k8s** Ansible module. To get started, install the module on your local workstation and test it using a playbook before moving on to using it within an Operator.

11.2.4.1. Installing the k8s Ansible module

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

Procedure

1. Install Ansible 2.6+:

```
$ sudo yum install ansible
```

2. Install the [OpenShift python client](#) package using **pip**:

```
$ pip install openshift
```


11.2.4.2. Testing the k8s Ansible module locally

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

Procedure

1. Initialize a new Ansible-based Operator project:

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

```
$ cd foo-operator
```

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

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

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

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

```
state: present
```

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

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

5. Run the playbook:

```

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

PLAY [localhost] *****

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

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

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

```

6. Check that the namespace was created:

```

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

```

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

```

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

PLAY [localhost] *****

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

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

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

```

8. Check that the namespace was deleted:

```

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

```

11.2.4.3. Testing the k8s Ansible module inside an Operator

After you are familiar using the **k8s** Ansible module locally, you can trigger the same Ansible logic inside of an Operator when a Custom Resource (CR) changes. This example maps an Ansible role to a specific Kubernetes resource that the Operator watches. This mapping is done in the Watches file.

11.2.4.3.1. Testing an Ansible-based Operator locally

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

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

Procedure

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

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

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

```
$ oc create -f deploy/crds/foo_v1alpha1_foo_crd.yaml
$ oc create -f deploy/service_account.yaml
$ oc create -f deploy/role.yaml
$ oc create -f deploy/role_binding.yaml
```

3. Run the **up local** command:

```
$ operator-sdk up local
[...]
```

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

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

```
apiVersion: "foo.example.com/v1alpha1"
kind: "Foo"
metadata:
  name: "example"
```

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

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

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

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

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

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

```
apiVersion: "foo.example.com/v1alpha1"
kind: "Foo"
metadata:
  name: "example"
spec:
  state: "absent"
```

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

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

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

11.2.4.3.2. Testing an Ansible-based Operator on a cluster

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

Procedure

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

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

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

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

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

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

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

```
$ oc create -f deploy/crds/foo_v1alpha1_foo_crd.yaml # if CRD doesn't exist already
$ oc create -f deploy/service_account.yaml
$ oc create -f deploy/role.yaml
$ oc create -f deploy/role_binding.yaml
$ oc create -f deploy/operator.yaml
```

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

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

11.2.5. Managing Custom Resource status using the `k8s_status` Ansible module

Ansible-based Operators automatically update Custom Resource (CR) **status** subresources with generic information about the previous Ansible run. This includes the number of successful and failed tasks and relevant error messages as shown:

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

Ansible-based Operators also allow Operator authors to supply custom status values with the `k8s_status` Ansible module. This allows the author to update the **status** from within Ansible with any key-value pair as desired.

By default, Ansible-based Operators always include the generic Ansible run output as shown above. If you would prefer your application did *not* update the status with Ansible output, you can track the status manually from your application.

Procedure

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

```
- version: v1
  group: api.example.com
  kind: Foo
```

```
role: /opt/ansible/roles/Foo
manageStatus: false
```

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

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

Additional resources

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

11.2.5.1. Using the k8s_status Ansible module when testing locally

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

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

Procedure

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

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

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

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

11.2.6. Additional resources

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

11.3. CREATING HELM-BASED OPERATORS

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

11.3.1. Helm chart support in the Operator SDK

The [Operator Framework](#) is an open source toolkit to manage Kubernetes native applications, called *Operators*, in an effective, automated, and scalable way. This framework includes the Operator SDK, which assists developers in bootstrapping and building an Operator based on their expertise without requiring knowledge of Kubernetes API complexities.

One of the Operator SDK's options for generating an Operator project includes leveraging an existing Helm chart to deploy Kubernetes resources as a unified application, without having to write any Go code. Such Helm-based Operators are designed to excel at stateless applications that require very little logic when rolled out, because changes should be applied to the Kubernetes objects that are generated as part of the chart. This may sound limiting, but can be sufficient for a surprising amount of use-cases as shown by the proliferation of Helm charts built by the Kubernetes community.

The main function of an Operator is to read from a custom object that represents your application instance and have its desired state match what is running. In the case of a Helm-based Operator, the object's spec field is a list of configuration options that are typically described in Helm's **values.yaml** file. Instead of setting these values with flags using the Helm CLI (for example, **helm install -f values.yaml**), you can express them within a Custom Resource (CR), which, as a native Kubernetes object, enables the benefits of RBAC applied to it and an audit trail.

For an example of a simple CR called **Tomcat**:

```
apiVersion: apache.org/v1alpha1
kind: Tomcat
metadata:
  name: example-app
spec:
  replicaCount: 2
```

The **replicaCount** value, **2** in this case, is propagated into the chart's templates where following is used:

```
{{ .Values.replicaCount }}
```

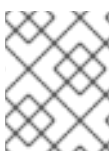
After an Operator is built and deployed, you can deploy a new instance of an app by creating a new instance of a CR, or list the different instances running in all environments using the **oc** command:

```
$ oc get Tomcats --all-namespaces
```

There is no requirement use the Helm CLI or install Tiller; Helm-based Operators import code from the Helm project. All you have to do is have an instance of the Operator running and register the CR with a Custom Resource Definition (CRD). And because it obeys RBAC, you can more easily prevent production changes.

11.3.2. Installing the Operator SDK CLI

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



NOTE

This guide uses [minikube](#) v0.25.0+ as the local Kubernetes cluster and [Quay.io](#) for the public registry.

11.3.2.1. Installing from GitHub release

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

Prerequisites

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

Procedure

1. Set the release version variable:

```
RELEASE_VERSION=v0.8.0
```

2. Download the release binary.

- For Linux:

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

- For macOS:

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

3. Verify the downloaded release binary.

- a. Download the provided ASC file.

- For Linux:

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

- For macOS:

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

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

- For Linux:


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

- For macOS:

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

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

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

- 1** RSA key string.

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

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

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

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

- For Linux:

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

- For macOS:

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

5. Verify that the CLI tool was installed correctly:

```
$ operator-sdk version
```

11.3.2.2. Installing from Homebrew

You can install the SDK CLI using Homebrew.

Prerequisites

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

Procedure

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

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

2. Verify that the CLI tool was installed correctly:

```
$ operator-sdk version
```

11.3.2.3. Compiling and installing from source

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

Prerequisites

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

Procedure

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

```
$ mkdir -p $GOPATH/src/github.com/operator-framework
$ cd $GOPATH/src/github.com/operator-framework
$ git clone https://github.com/operator-framework/operator-sdk
$ cd operator-sdk
```

2. Check out the desired release branch:

```
$ git checkout master
```

3. Compile and install the SDK CLI:

```
$ make dep
$ make install
```

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

4. Verify that the CLI tool was installed correctly:

```
$ operator-sdk version
```

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

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

TIP

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

Prerequisites

- Operator SDK CLI installed on the development workstation
- Access to a Kubernetes-based cluster v1.11.3+ (for example OpenShift Container Platform 4.1) using an account with **cluster-admin** permissions
- OpenShift CLI (**oc**) v4.1+ installed

Procedure

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

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

This creates the **nginx-operator** project specifically for watching the Nginx resource with APIVersion **example.com/v1alpha1** and Kind **Nginx**.

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

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

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

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

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

2. Customize the Operator logic.

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

- Create a Nginx Deployment if it does not exist.
- Create a Nginx Service if it does not exist.
- Create a Nginx Ingress if it is enabled and does not exist.
- Ensure that the Deployment, Service, and optional Ingress match the desired configuration (for example, replica count, image, service type) as specified by the Nginx CR.

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

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

a. Review the Nginx Helm chart.

When a Helm Operator project is created, the Operator SDK creates an example Helm chart that contains a set of templates for a simple Nginx release.

For this example, templates are available for Deployment, Service, and Ingress resources, along with a **NOTES.txt** template, which Helm chart developers use to convey helpful information about a release.

If you are not already familiar with Helm Charts, take a moment to review the [Helm Chart developer documentation](#).

b. Understand the Nginx CR spec.

Helm uses a concept called [values](#) to provide customizations to a Helm chart's defaults, which are defined in the Helm chart's **values.yaml** file.

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

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

Update the **deploy/crds/example_v1alpha1/nginx_cr.yaml** file to look like the following:

```
apiVersion: example.com/v1alpha1
kind: Nginx
metadata:
  name: example-nginx
spec:
  replicaCount: 2
```

- ii. Similarly, the default service port is set to **80**. To instead use **8080**, update the **deploy/crds/example_v1alpha1/nginx_cr.yaml** file again by adding the service port override:

```
apiVersion: example.com/v1alpha1
kind: Nginx
metadata:
  name: example-nginx
spec:
  replicaCount: 2
  service:
    port: 8080
```

The Helm Operator applies the entire spec as if it was the contents of a values file, just like the **helm install -f ./overrides.yaml** command works.

3. Deploy the CRD.

Before running the Operator, Kubernetes needs to know about the new custom resource definition (CRD) the operator will be watching. Deploy the following CRD:

```
$ oc create -f deploy/crds/example_v1alpha1/nginx_crd.yaml
```

4. Build and run the Operator.

There are two ways to build and run the Operator:

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

Choose one of the following methods:

- a. **Run as a Pod** inside a Kubernetes cluster. This is the preferred method for production use.
 - i. Build the **nginx-operator** image and push it to a registry:

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

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

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

- iii. If you created your Operator using the **--cluster-scoped=true** flag, update the service account namespace in the generated **ClusterRoleBinding** to match where you are deploying your Operator:

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

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

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

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

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

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

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

- b. **Run outside the cluster.** This method is preferred during the development cycle to speed up deployment and testing.

It is important that the chart path referenced in the **watches.yaml** file exists on your machine. By default, the **watches.yaml** file is scaffolded to work with an Operator image built with the **operator-sdk build** command. When developing and testing your operator with the **operator-sdk up local** command, the SDK looks in your local file system for this path.

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

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

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

```
$ operator-sdk up local
```

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

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

5. Deploy the Nginx CR.

Apply the **Nginx** CR that you modified earlier:

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

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

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

Check the Pods to confirm two replicas were created:

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

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

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

6. Update the replicaCount and remove the port.

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

```
$ cat deploy/crds/example_v1alpha1_nginx_cr.yaml
apiVersion: "example.com/v1alpha1"
kind: "Nginx"
metadata:
  name: "example-nginx"
spec:
  replicaCount: 3

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

Confirm that the Operator changes the Deployment size:

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

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

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

7. Clean up the resources:

```
$ oc delete -f deploy/crds/example_v1alpha1_nginx_cr.yaml
$ oc delete -f deploy/operator.yaml
$ oc delete -f deploy/role_binding.yaml
$ oc delete -f deploy/role.yaml
$ oc delete -f deploy/service_account.yaml
$ oc delete -f deploy/crds/example_v1alpha1_nginx_crd.yaml
```

11.3.4. Additional resources

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

11.4. GENERATING A CLUSTERSERVICEVERSION (CSV)

A *ClusterServiceVersion* (CSV) is a YAML manifest created from Operator metadata that assists the Operator Lifecycle Manager (OLM) in running the Operator in a cluster. It is the metadata that accompanies an Operator container image, used to populate user interfaces with information like its logo, description, and version. It is also a source of technical information that is required to run the Operator, like the RBAC rules it requires and which Custom Resources (CRs) it manages or depends on.

The Operator SDK includes the **olm-catalog gen-csv** subcommand to generate a *ClusterServiceVersion* (CSV) for the current Operator project customized using information contained in manually-defined YAML manifests and Operator source files.

A CSV-generating command removes the responsibility of Operator authors having in-depth Operator Lifecycle Manager (OLM) knowledge in order for their Operator to interact with OLM or publish metadata to the Catalog Registry. Further, because the CSV spec will likely change over time as new Kubernetes and OLM features are implemented, the Operator SDK is equipped to easily extend its update system to handle new CSV features going forward.

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

```
$ operator-sdk olm-catalog gen-csv --csv-version <version>
```

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

11.4.1. How CSV generation works

An Operator project's **deploy/** directory is the standard location for all manifests required to deploy an Operator. The Operator SDK can use data from manifests in **deploy/** to write a CSV. The following command:

```
$ operator-sdk olm-catalog gen-csv --csv-version <version>
```

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

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

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

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

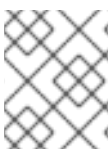
Workflow

Depending on whether an existing CSV is detected, and assuming all configuration defaults are used, the **olm-catalog gen-csv** subcommand either:

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

or:

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



NOTE

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

11.4.2. CSV composition configuration

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

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

11.4.3. Manually-defined CSV fields

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

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

Table 11.5. Required

Field	Description
metadata.name	A unique name for this CSV. Operator version should be included in the name to ensure uniqueness, for example app-operator.v0.1.1 .
spec.displayName	A public name to identify the Operator.
spec.description	A short description of the Operator's functionality.
spec.keywords	Keywords describing the operator.
spec.maintainers	Human or organizational entities maintaining the Operator, with a name and email .
spec.provider	The Operators' provider (usually an organization), with a name .
spec.labels	Key-value pairs to be used by Operator internals.
spec.version	Semantic version of the Operator, for example 0.1.1 .

Field	Description
spec.customresourcedefinitions	<p>Any CRDs the Operator uses. This field is populated automatically by the Operator SDK if any CRD YAML files are present in deploy/. However, several fields not in the CRD manifest spec require user input:</p> <ul style="list-style-type: none"> ● description: description of the CRD. ● resources: any Kubernetes resources leveraged by the CRD, for example Pods and StatefulSets. ● specDescriptors: UI hints for inputs and outputs of the Operator.

Table 11.6. Optional

Field	Description
spec.replaces	The name of the CSV being replaced by this CSV.
spec.links	URLs (for example, websites and documentation) pertaining to the Operator or application being managed, each with a name and url .
spec.selector	Selectors by which the Operator can pair resources in a cluster.
spec.icon	A base64-encoded icon unique to the Operator, set in a base64data field with a mediatype .
spec.maturity	The Operator's capability level according to the Operator maturity model, for example Seamless Upgrades .

Further details on what data each field above should hold are found in the [CSV spec](#).

**NOTE**

Several YAML fields currently requiring user intervention can potentially be parsed from Operator code; such Operator SDK functionality will be addressed in a future design document.

Additional resources

- [Operator maturity model](#)

11.4.4. Generating a CSV**Prerequisites**

- An Operator project generated using the Operator SDK

Procedure

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

```
$ operator-sdk olm-catalog gen-csv --csv-version <version>
```

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

11.4.5. Understanding your Custom Resource Definitions (CRDs)

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

11.4.5.1. Owned CRDs

The CRDs owned by your Operator are the most important part of your CSV. This establishes the link between your Operator and the required RBAC rules, dependency management, and other Kubernetes concepts.

It is common for your Operator to use multiple CRDs to link together concepts, such as top-level database configuration in one object and a representation of ReplicaSets in another. Each one should be listed out in the CSV file.

Table 11.7. Owned CRD fields

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

Field	Description	Required/Optional
Resources	<p>Your CRDs own one or more types of Kubernetes objects. These are listed in the resources section to inform your users of the objects they might need to troubleshoot or how to connect to the application, such as the Service or Ingress rule that exposes a database.</p> <p>It is recommended to only list out the objects that are important to a human, not an exhaustive list of everything you orchestrate. For example, ConfigMaps that store internal state that should not be modified by a user should not appear here.</p>	Optional
SpecDescriptors, StatusDescriptors, and ActionDescriptors	<p>These Descriptors are a way to hint UIs with certain inputs or outputs of your Operator that are most important to an end user. If your CRD contains the name of a Secret or ConfigMap that the user must provide, you can specify that here. These items are linked and highlighted in compatible UIs.</p> <p>There are three types of descriptors:</p> <ul style="list-style-type: none"> ● SpecDescriptors: A reference to fields in the spec block of an object. ● StatusDescriptors: A reference to fields in the status block of an object. ● ActionDescriptors: A reference to actions that can be performed on an object. <p>All Descriptors accept the following fields:</p> <ul style="list-style-type: none"> ● DisplayName: A human readable name for the Spec, Status, or Action. ● Description: A short description of the Spec, Status, or Action and how it is used by the Operator. ● Path: A dot-delimited path of the field on the object that this descriptor describes. ● X-Descriptors: Used to determine which "capabilities" this descriptor has and which UI component to use. See the openshift/console project for a canonical list of React UI X-Descriptors for OpenShift Container Platform. <p>Also see the openshift/console project for more information on Descriptors in general.</p>	Optional

The following example depicts a **MongoDB Standalone** CRD that requires some user input in the form of a Secret and ConfigMap, and orchestrates Services, StatefulSets, Pods and ConfigMaps:

Example owned CRD

```

- displayName: MongoDB Standalone
  group: mongodb.com
  kind: MongoDbStandalone
  name: mongodbstandalones.mongodb.com
  resources:
    - kind: Service
      name: ""
      version: v1
    - kind: StatefulSet
      name: ""
      version: v1beta2
    - kind: Pod
      name: ""
      version: v1
    - kind: ConfigMap
      name: ""
      version: v1
  specDescriptors:
    - description: Credentials for Ops Manager or Cloud Manager.
      displayName: Credentials
      path: credentials
      x-descriptors:
        - 'urn:alm:descriptor:com.tectonic.ui.selector:core:v1:Secret'
    - description: Project this deployment belongs to.
      displayName: Project
      path: project
      x-descriptors:
        - 'urn:alm:descriptor:com.tectonic.ui.selector:core:v1:ConfigMap'
    - description: MongoDB version to be installed.
      displayName: Version
      path: version
      x-descriptors:
        - 'urn:alm:descriptor:com.tectonic.ui:label'
  statusDescriptors:
    - description: The status of each of the Pods for the MongoDB cluster.
      displayName: Pod Status
      path: pods
      x-descriptors:
        - 'urn:alm:descriptor:com.tectonic.ui:podStatuses'
  version: v1
  description: >-
    MongoDB Deployment consisting of only one host. No replication of
    data.

```

11.4.5.2. Required CRDs

Relying on other required CRDs is completely optional and only exists to reduce the scope of individual Operators and provide a way to compose multiple Operators together to solve an end-to-end use case.

An example of this is an Operator that might set up an application and install an etcd cluster (from an etcd Operator) to use for distributed locking and a Postgres database (from a Postgres Operator) for data storage.

The Operator Lifecycle Manager (OLM) checks against the available CRDs and Operators in the cluster to fulfill these requirements. If suitable versions are found, the Operators are started within the desired namespace and a Service Account created for each Operator to create, watch, and modify the Kubernetes resources required.

Table 11.8. Required CRD fields

Field	Description	Required/Optional
Name	The full name of the CRD you require.	Required
Version	The version of that object API.	Required
Kind	The Kubernetes object kind.	Required
DisplayName	A human readable version of the CRD.	Required
Description	A summary of how the component fits in your larger architecture.	Required

Example required CRD

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

11.4.5.3. CRD templates

Users of your Operator will need to be aware of which options are required versus optional. You can provide templates for each of your CRDs with a minimum set of configuration as an annotation named **alm-examples**. Compatible UIs will pre-fill this template for users to further customize.

The annotation consists of a list of the **kind**, for example, the CRD name and the corresponding **metadata** and **spec** of the Kubernetes object.

The following full example provides templates for **EtcdCluster**, **EtcdBackup** and **EtcdRestore**:

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

11.4.6. Understanding your API services

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

11.4.6.1. Owned APIServices

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

An APIService is uniquely identified by the **group-version** it provides and can be listed multiple times to denote the different kinds it is expected to provide.

Table 11.9. Owned APIService fields

Field	Description	Required/Optional
Group	Group that the APIService provides, for example database.example.com .	Required
Version	Version of the APIService, for example v1alpha1 .	Required
Kind	A kind that the APIService is expected to provide.	Required
Name	The plural name for the APIService provided	Required
DeploymentName	Name of the deployment defined by your CSV that corresponds to your APIService (required for owned APIServices). During the CSV pending phase, the OLM Operator searches your CSV's InstallStrategy for a deployment spec with a matching name, and if not found, does not transition the CSV to the install ready phase.	Required
DisplayName	A human readable version of your APIService name, for example MongoDB Standalone .	Required
Description	A short description of how this APIService is used by the Operator or a description of the functionality provided by the APIService.	Required
Resources	Your APIServices own one or more types of Kubernetes objects. These are listed in the resources section to inform your users of the objects they might need to troubleshoot or how to connect to the application, such as the Service or Ingress rule that exposes a database. It is recommended to only list out the objects that are important to a human, not an exhaustive list of everything you orchestrate. For example, ConfigMaps that store internal state that should not be modified by a user should not appear here.	Optional

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

11.4.6.1.1. APIService Resource Creation

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

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

11.4.6.1.2. APIService Serving Certs

The OLM handles generating a serving key/cert pair whenever an owned APIService is being installed. The serving certificate has a CN containing the host name of the generated Service resource and is signed by the private key of the CA bundle embedded in the corresponding APIService resource.

The cert is stored as a type **kubernetes.io/tls** Secret in the deployment namespace, and a Volume named **apiservice-cert** is automatically appended to the Volumes section of the deployment in the CSV matching the APIServiceDescription's **DeploymentName** field.

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

11.4.6.2. Required APIServices

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

Table 11.10. Required APIService fields

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

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

11.5. CONFIGURING BUILT-IN MONITORING WITH PROMETHEUS

This guide describes the built-in monitoring support provided by the Operator SDK using the Prometheus Operator and details usage for Operator authors.

11.5.1. Prometheus Operator support

[Prometheus](#) is an open-source systems monitoring and alerting toolkit. The Prometheus Operator creates, configures, and manages Prometheus clusters running on Kubernetes-based clusters, such as OpenShift Container Platform.

Helper functions exist in the Operator SDK by default to automatically set up metrics in any generated Go-based Operator for use on clusters where the Prometheus Operator is deployed.

11.5.2. Metrics helper

In Go-based Operators generated using the Operator SDK, the following function exposes general metrics about the running program:

```
func ExposeMetricsPort(ctx context.Context, port int32) (*v1.Service, error)
```

These metrics are inherited from the **controller-runtime** library API. By default, the metrics are served on **0.0.0.0:8383/metrics**.

A Service object is created with the metrics port exposed, which can be then accessed by Prometheus. The Service object is garbage collected when the leader Pod's root owner is deleted.

The following example is present in the **cmd/manager/main.go** file in all Operators generated using the Operator SDK:

```
import(
    "github.com/operator-framework/operator-sdk/pkg/metrics"
    "machine.openshift.io/controller-runtime/pkg/manager"
)

var (
    // Change the below variables to serve metrics on a different host or port.
    metricsHost    = "0.0.0.0" 1
    metricsPort int32 = 8383 2
)
...
func main() {
```

```

...
// Pass metrics address to controller-runtime manager
mgr, err := manager.New(cfg, manager.Options{
    Namespace:      namespace,
    MetricsBindAddress: fmt.Sprintf("%s:%d", metricsHost, metricsPort),
})

...
// Create Service object to expose the metrics port.
_, err = metrics.ExposeMetricsPort(ctx, metricsPort)
if err != nil {
    // handle error
    log.Info(err.Error())
}
...
}

```

- 1 The host that the metrics are exposed on.
- 2 The port that the metrics are exposed on.

11.5.2.1. Modifying the metrics port

Operator authors can modify the port that metrics are exposed on.

Prerequisites

- Go-based Operator generated using the Operator SDK
- Kubernetes-based cluster with the Prometheus Operator deployed

Procedure

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

11.5.3. ServiceMonitor resources

A ServiceMonitor is a Custom Resource Definition (CRD) provided by the Prometheus Operator that discovers the **Endpoints** in Service objects and configures Prometheus to monitor those Pods.

In Go-based Operators generated using the Operator SDK, the `GenerateServiceMonitor()` helper function can take a Service object and generate a ServiceMonitor Custom Resource (CR) based on it.

Additional resources

- See the [Prometheus Operator documentation](#) for more information about the ServiceMonitor CRD.

11.5.3.1. Creating ServiceMonitor resources

Operator authors can add Service target discovery of created monitoring Services using the `metrics.CreateServiceMonitor()` helper function, which accepts the newly created Service.

Prerequisites

- Go-based Operator generated using the Operator SDK
- Kubernetes-based cluster with the Prometheus Operator deployed

Procedure

- Add the **metrics.CreateServiceMonitor()** helper function to your Operator code:

```
import(
    "k8s.io/api/core/v1"
    "github.com/operator-framework/operator-sdk/pkg/metrics"
    "machine.openshift.io/controller-runtime/pkg/client/config"
)
func main() {
    ...
    // Populate below with the Service(s) for which you want to create ServiceMonitors.
    services := []*v1.Service{}
    // Create one ServiceMonitor per application per namespace.
    // Change the below value to name of the Namespace you want the ServiceMonitor to be
    // created in.
    ns := "default"
    // restConfig is used for talking to the Kubernetes apiserver
    restConfig := config.GetConfig()

    // Pass the Service(s) to the helper function, which in turn returns the array of
    // ServiceMonitor objects.
    serviceMonitors, err := metrics.CreateServiceMonitors(restConfig, ns, services)
    if err != nil {
        // Handle errors here.
    }
    ...
}
```

11.6. CONFIGURING LEADER ELECTION

During the lifecycle of an Operator, it is possible that there may be more than one instance running at any given time, for example when rolling out an upgrade for the Operator. In such a scenario, it is necessary to avoid contention between multiple Operator instances using leader election. This ensures only one leader instance handles the reconciliation while the other instances are inactive but ready to take over when the leader steps down.

There are two different leader election implementations to choose from, each with its own trade-off:

- *Leader-for-life*: The leader Pod only gives up leadership (using garbage collection) when it is deleted. This implementation precludes the possibility of two instances mistakenly running as leaders (split brain). However, this method can be subject to a delay in electing a new leader. For example, when the leader Pod is on an unresponsive or partitioned node, the **pod-eviction-timeout** dictates how it takes for the leader Pod to be deleted from the node and step down (default **5m**). See the [Leader-for-life](#) Go documentation for more.
- *Leader-with-lease*: The leader Pod periodically renews the leader lease and gives up leadership when it cannot renew the lease. This implementation allows for a faster transition to a new

leader when the existing leader is isolated, but there is a possibility of split brain in [certain situations](#). See the [Leader-with-lease](#) Go documentation for more.

By default, the Operator SDK enables the Leader-for-life implementation. Consult the related Go documentation for both approaches to consider the trade-offs that make sense for your use case,

The following examples illustrate how to use the two options.

11.6.1. Using Leader-for-life election

With the Leader-for-life election implementation, a call to `leader.Become()` blocks the Operator as it retries until it can become the leader by creating the ConfigMap named **memcached-operator-lock**:

```
import (
    ...
    "github.com/operator-framework/operator-sdk/pkg/leader"
)

func main() {
    ...
    err = leader.Become(context.TODO(), "memcached-operator-lock")
    if err != nil {
        log.Error(err, "Failed to retry for leader lock")
        os.Exit(1)
    }
    ...
}
```

If the Operator is not running inside a cluster, `leader.Become()` simply returns without error to skip the leader election since it cannot detect the Operator's namespace.

11.6.2. Using Leader-with-lease election

The Leader-with-lease implementation can be enabled using the [Manager Options](#) for leader election:

```
import (
    ...
    "sigs.k8s.io/controller-runtime/pkg/manager"
)

func main() {
    ...
    opts := manager.Options{
        ...
        LeaderElection: true,
        LeaderElectionID: "memcached-operator-lock"
    }
    mgr, err := manager.New(cfg, opts)
    ...
}
```

When the Operator is not running in a cluster, the Manager returns an error when starting since it cannot detect the Operator's namespace in order to create the ConfigMap for leader election. You can override this namespace by setting the Manager's **LeaderElectionNamespace** option.

11.7. OPERATOR SDK CLI REFERENCE

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

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

11.7.1. build

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

Table 11.11. build arguments

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

Table 11.12. build flags

Flag	Description
--enable-tests (bool)	Enable in-cluster testing by adding test binary to the image.
--namespaced-manifest (string)	Path of namespaced resources manifest for tests. Default: deploy/operator.yaml .
--test-location (string)	Location of tests. Default: ./test/e2e
-h, --help	Usage help output.

If **--enable-tests** is set, the **build** command also builds the testing binary, adds it to the container image, and generates a **deploy/test-pod.yaml** file that allows a user to run the tests as a Pod on a cluster.

Example output

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

building example-operator...

building container quay.io/example/operator:v0.0.1...
Sending build context to Docker daemon 163.9MB
Step 1/4 : FROM alpine:3.6
---> 77144d8c6bdc
Step 2/4 : ADD tmp/_output/bin/example-operator /usr/local/bin/example-operator
---> 2ada0d6ca93c
Step 3/4 : RUN adduser -D example-operator
---> Running in 34b4bb507c14
Removing intermediate container 34b4bb507c14
---> c671ec1cff03
Step 4/4 : USER example-operator
---> Running in bd336926317c
```

```
Removing intermediate container bd336926317c
---> d6b58a0fcb8c
Successfully built d6b58a0fcb8c
Successfully tagged quay.io/example/operator:v0.0.1
```

11.7.2. completion

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

Table 11.13. completion subcommands

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

Table 11.14. completion flags

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

Example output

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

11.7.3. print-deps

The **operator-sdk print-deps** command prints the most recent Golang packages and versions required by Operators. It prints in columnar format by default.

Table 11.15. print-deps flags

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

Example output

```
$ operator-sdk print-deps --as-file
required = [
  "k8s.io/code-generator/cmd/defaulter-gen",
  "k8s.io/code-generator/cmd/deepcopy-gen",
```

```

"k8s.io/code-generator/cmd/conversion-gen",
"k8s.io/code-generator/cmd/client-gen",
"k8s.io/code-generator/cmd/lister-gen",
"k8s.io/code-generator/cmd/informer-gen",
"k8s.io/code-generator/cmd/openapi-gen",
"k8s.io/gengo/args",
]

[[override]]
name = "k8s.io/code-generator"
revision = "6702109cc68eb6fe6350b83e14407c8d7309fd1a"
...

```

11.7.4. generate

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

Table 11.16. **generate** subcommands

Subcommand	Description
k8s	Runs the Kubernetes code-generators for all CRD APIs under pkg/apis/ . Currently, k8s only runs deepcopy-gen to generate the required DeepCopy() functions for all Custom Resource (CR) types.



NOTE

This command must be run every time the API (**spec** and **status**) for a custom resource type is updated.

Example output

```

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

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

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

```

11.7.5. olm-catalog

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

11.7.5.1. gen-csv

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

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

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

Example output

```
$ operator-sdk olm-catalog gen-csv --csv-version 0.1.0 --update-crds
INFO[0000] Generating CSV manifest version 0.1.0
INFO[0000] Fill in the following required fields in file deploy/olm-catalog/operator-
name/0.1.0/operator-name.v0.1.0.clusterserviceversion.yaml:
spec.keywords
spec.maintainers
spec.provider
spec.labels
INFO[0000] Created deploy/olm-catalog/operator-name/0.1.0/operator-
name.v0.1.0.clusterserviceversion.yaml
```

11.7.6. new

The **operator-sdk new** command creates a new Operator application and generates (or *scaffolds*) a default project directory layout based on the input **<project_name>**.

Table 11.18. **new** arguments

Argument	Description
<project_name>	Name of the new project.

Table 11.19. **new** flags

Flag	Description
--api-version	CRD APIVersion in the format \$GROUP_NAME/\$VERSION , for example app.example.com/v1alpha1 . Used with ansible or helm types.
--dep-manager [dep modules]	Dependency manager the new project will use. Used with go type. (Default: modules)
--generate-playbook	Generate an Ansible playbook skeleton. Used with ansible type.
--header-file <string>	Path to file containing headers for generated Go files. Copied to hack/boilerplate.go.txt .
--helm-chart <string>	Initialize Helm operator with existing Helm chart: <url> , <repo>/<name> , or local path.
--helm-chart-repo <string>	Chart repository URL for the requested Helm chart.
--helm-chart-version <string>	Specific version of the Helm chart. (Default: latest version)
--help, -h	Usage and help output.
--kind <string>	CRD Kind , for example AppService . Used with ansible or helm types.
--skip-git-init	Do not initialize the directory as a Git repository.
--type	Type of Operator to initialize: go , ansible or helm . (Default: go)

Example usage for Go project

```
$ mkdir $GOPATH/src/github.com/example.com/
$ cd $GOPATH/src/github.com/example.com/
$ operator-sdk new app-operator
```

Example usage for Ansible project

```
$ operator-sdk new app-operator \
  --type=ansible \
  --api-version=app.example.com/v1alpha1 \
  --kind=AppService
```

11.7.7. add

The **operator-sdk add** command adds a controller or resource to the project. The command must be run from the Operator project root directory.

Table 11.20. `add` subcommands

Subcommand	Description
<code>api</code>	Adds a new API definition for a new Custom Resource (CR) under <code>pkg/apis</code> and generates the Customer Resource Definition (CRD) and Custom Resource (CR) files under <code>deploy/crds/</code> . If the API already exists at <code>pkg/apis/<group>/<version></code> , then the command does not overwrite and returns an error.
<code>controller</code>	Adds a new controller under <code>pkg/controller/<kind>/</code> . The controller expects to use the CR type that should already be defined under <code>pkg/apis/<group>/<version></code> via the <code>operator-sdk add api --kind=<kind> --api-version=<group>/<version></code> command. If the controller package for that <code>Kind</code> already exists at <code>pkg/controller/<kind></code> , then the command does not overwrite and returns an error.
<code>crd</code>	Adds a CRD and the CR files. The <code><project-name>/deploy</code> path must already exist. The <code>--api-version</code> and <code>--kind</code> flags are required to generate the new Operator application. <ul style="list-style-type: none"> Generated CRD filename: <code><project-name>/deploy/crds/<group>_<version>_<kind>_crd.yaml</code> Generated CR filename: <code><project-name>/deploy/crds/<group>_<version>_<kind>_cr.yaml</code>

Table 11.21. `add api` flags

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

Example `add api` output

```
$ operator-sdk add api --api-version app.example.com/v1alpha1 --kind AppService
Create pkg/apis/app/v1alpha1/appservice_types.go
Create pkg/apis/addtoscheme_app_v1alpha1.go
Create pkg/apis/app/v1alpha1/register.go
Create pkg/apis/app/v1alpha1/doc.go
Create deploy/crds/app_v1alpha1_appservice_cr.yaml
Create deploy/crds/app_v1alpha1_appservice_crd.yaml
Running code-generation for Custom Resource (CR) group versions: [app:v1alpha1]
Generating deepcopy funcs
```

```
$ tree pkg/apis
pkg/apis/
├── addtoscheme_app_appservice.go
└── apis.go
```

```

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

```

Example add controller output

```

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

```

```

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

```

Example add crd output

```

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

```

11.7.8. test

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

11.7.8.1. local

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

Table 11.22. **test local** arguments

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

Table 11.23. **test local** flags

Flags	Description
--kubeconfig (string)	Location of kubeconfig for a cluster. Default: ~/kube/config .
--global-manifest (string)	Path to manifest for global resources. Default: deploy/crd.yaml .

Flags	Description
--namespaced-manifest (string)	Path to manifest for per-test, namespaced resources. Default: combines deploy/service_account.yaml , deploy/rbac.yaml , and deploy/operator.yaml .
--namespace (string)	If non-empty, a single namespace to run tests in (e.g., operator-test). Default: ""
--go-test-flags (string)	Extra arguments to pass to go test (e.g., -f "-v -parallel=2").
--up-local	Enable running the Operator locally with go run instead of as an image in the cluster.
--no-setup	Disable test resource creation.
--image (string)	Use a different Operator image from the one specified in the namespaced manifest.
-h, --help	Usage help output.

Example output

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

```
# Output:
```

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

11.7.9. up

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

11.7.9.1. local

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

Table 11.24. **up local** arguments

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

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

Example output

```
$ operator-sdk up local \
  --kubeconfig "mycluster.kubecfg" \
  --namespace "default" \
  --operator-flags "--flag1 value1 --flag2=value2"
```

The following example uses the default **kubeconfig**, the default namespace environment variable, and passes in flags for the Operator. To use the Operator flags, your Operator must know how to handle the option. For example, for an Operator that understands the **resync-interval** flag:

```
$ operator-sdk up local --operator-flags "--resync-interval 10"
```

If you are planning on using a different namespace than the default, use the **--namespace** flag to change where the Operator is watching for Custom Resources (CRs) to be created:

```
$ operator-sdk up local --namespace "testing"
```

For this to work, your Operator must handle the **WATCH_NAMESPACE** environment variable. This can be accomplished using the [utility function](#) `k8sutil.GetWatchNamespace` in your Operator.

11.8. APPENDICES

11.8.1. Operator project scaffolding layout

The **operator-sdk** CLI generates a number of packages for each Operator project. The following sections describes a basic rundown of each generated file and directory.

11.8.1.1. Go-based projects

Go-based Operator projects (the default type) generated using the **operator-sdk new** command contain the following directories and files:

File/folders	Purpose
cmd/	Contains manager/main.go file, which is the main program of the Operator. This instantiates a new manager which registers all Custom Resource Definitions under pkg/apis/ and starts all controllers under pkg/controllers/ .
pkg/apis/	Contains the directory tree that defines the APIs of the Custom Resource Definitions (CRDs). Users are expected to edit the pkg/apis/<group>/<version>/<kind>_types.go files to define the API for each resource type and import these packages in their controllers to watch for these resource types.

File/folders	Purpose
pkg/controller	This pkg contains the controller implementations. Users are expected to edit the pkg/controller/<kind>/<kind>_controller.go files to define the controller's reconcile logic for handling a resource type of the specified kind .
build/	Contains the Dockerfile and build scripts used to build the Operator.
deploy/	Contains various YAML manifests for registering CRDs, setting up RBAC, and deploying the Operator as a Deployment.
Gopkg.toml Gopkg.lock	The Go Dep manifests that describe the external dependencies of this Operator.
vendor/	The golang vendor folder that contains the local copies of the external dependencies that satisfy the imports of this project. Go Dep manages the vendor directly.

11.8.1.2. Helm-based projects

Helm-based Operator projects generated using the **operator-sdk new --type helm** command contain the following directories and files:

File/folders	Purpose
deploy/	Contains various YAML manifests for registering CRDs, setting up RBAC, and deploying the Operator as a Deployment.
helm-charts/<kind>	Contains a Helm chart initialized using the equivalent of the helm create command.
build/	Contains the Dockerfile and build scripts used to build the Operator.
watches.yaml	Contains Group, Version, Kind , and Helm chart location.