



Red Hat Advanced Cluster Management for Kubernetes 2.1

Manage applications

Manage applications

Red Hat Advanced Cluster Management for Kubernetes 2.1 Manage applications

Manage applications

Legal Notice

Copyright © 2022 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

Manage applications in Red Hat Advanced Cluster Management for Kubernetes

Table of Contents

CHAPTER 1. MANAGING APPLICATIONS	4
1.1. APPLICATION MODEL AND DEFINITIONS	4
1.1.1. Applications	5
1.1.2. Channels	5
1.1.3. Subscriptions	6
1.1.4. Placement rules	6
1.2. APPLICATION CONSOLE	6
1.2.1. Applications overview	7
1.2.1.1. Single applications overview	7
1.2.2. Resource topology	8
1.2.3. Search	8
1.2.4. Advanced configuration	9
1.3. MANAGING APPLICATION RESOURCES	9
1.3.1. Managing apps with Git repositories	9
1.3.1.1. Sample YAML	10
1.3.1.2. Application Git Ops	11
1.3.1.2.1. Sample Repo for GitOps	11
1.3.1.2.2. GitOps root path	11
1.3.1.2.3. GitOps application to managed clusters	12
1.3.1.2.4. GitOps application to hub clusters	12
1.3.1.2.5. Apply GitOps	12
1.3.2. Managing apps with Helm repositories	13
1.3.2.1. Sample YAML	14
1.3.3. Managing apps with Object storage repositories	14
1.3.3.1. Sample YAML	15
1.4. APPLICATION ADVANCED CONFIGURATION	16
1.4.1. Subscribing Git resources	16
1.4.1.1. Granting users and groups subscription admin privilege	16
1.4.1.2. Application namespace example	17
1.4.1.3. Resource overwrite example	18
1.4.1.3.1. Reconcile option	20
1.4.2. Configuring package overrides	20
1.4.3. Setting up Ansible Tower tasks (Technology preview)	21
1.4.3.1. Prerequisites	21
1.4.3.2. Install Ansible Automation Platform Resource Operator:	21
1.4.3.3. Obtain the Ansible Tower URL and token	21
1.4.3.4. Obtaining a token	21
1.4.3.5. Ansible integration	22
1.4.3.6. Ansible operator components	22
1.4.3.6.1. Prehook	22
1.4.3.6.2. Posthook	22
1.4.3.6.3. Ansible placement rules	22
1.4.3.7. Ansible configuration	23
1.4.3.7.1. Ansible secrets	23
1.4.3.8. Set secret reconciliation	23
1.4.3.9. Ansible sample YAML	24
1.4.4. Channel samples	25
1.4.4.1. Channel YAML structure	25
1.4.4.2. Channel YAML table	25
1.4.4.3. Object storage bucket (ObjectBucket) channel	27
1.4.4.4. Helm repository (HelmRepo) channel	27

1.4.4.5. Git (Git) repository channel	28
1.4.5. Secret samples	28
1.4.5.1. Secret YAML structure	29
1.4.6. Subscription samples	29
1.4.6.1. Subscription YAML structure	29
1.4.6.2. Subscription YAML table	30
1.4.6.3. Subscription file samples	35
1.4.6.3.1. Subscription time window example	35
1.4.6.3.2. Subscription with overrides example	36
1.4.6.3.3. Helm repository subscription example	36
1.4.6.3.4. Git repository subscription example	37
1.4.6.3.4.1. Subscribing specific branch and directory of Git repository	37
1.4.6.3.4.2. Adding a .kubernetesignore file	38
1.4.6.3.4.3. Applying Kustomize	38
1.4.6.3.4.4. Enabling Git WebHook	38
1.4.6.3.4.4.1. Payload URL	39
1.4.6.3.4.4.2. Webhook secret	39
1.4.6.3.4.4.3. Configuring WebHook in Git repository	39
1.4.6.3.4.4.4. Enable WebHook event notification in channel	39
1.4.6.3.4.4.5. Subscriptions of webhook-enabled channel	39
1.4.7. Placement rule samples	39
1.4.7.1. Placement rule YAML structure	40
1.4.7.2. Placement rule YAML values table	40
1.4.7.3. Placement rule sample files	42
1.4.8. Application samples	43
1.4.8.1. Application YAML structure	43
1.4.8.2. Application YAML table	44
1.4.8.3. Application file samples	44

CHAPTER 1. MANAGING APPLICATIONS

Review the following topics to learn more about creating, deploying, and managing your applications. This guide assumes familiarity with Kubernetes concepts and terminology. Key Kubernetes terms and components are not defined. For more information about Kubernetes concepts, see [Kubernetes Documentation](#).

The application management functions provide you with unified and simplified options for constructing and deploying applications and application updates. With these functions, your developers and DevOps personnel can create and manage applications across environments through channel and subscription-based automation.

See the following topics:

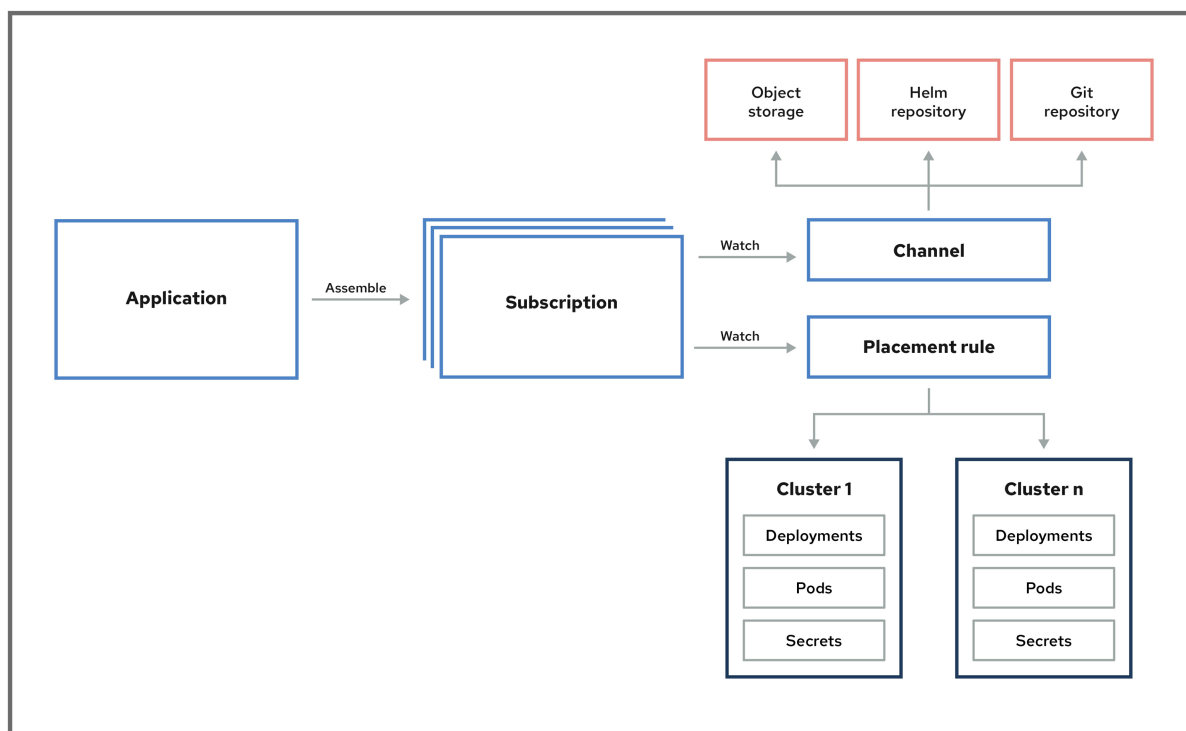
- [Application model and definitions](#)
- [Application console](#)
- [Managing application resources](#)
- [Managing apps with Git repositories](#)
- [Managing apps with Helm repositories](#)
- [Managing apps with Object storage repositories](#)
- [Application advanced configuration](#)
- [Subscribing Git resources](#)
- [Configuring package overrides](#)
- [Setting up Ansible Tower tasks](#)
- [Channel samples](#)
- [Subscription samples](#)
- [Placement rule samples](#)
- [Application samples](#)

1.1. APPLICATION MODEL AND DEFINITIONS

The application model is based on subscribing to one or more Kubernetes resource repositories (*channel* resources) that contains resources that are deployed on managed clusters. Both single and multicluster applications use the same Kubernetes specifications, but multicluster applications involve more automation of the deployment and application management lifecycle.

See the following image to understand more about the application model:

APPLICATION SUBSCRIPTION MODEL



View the following application resource sections:

1.1.1. Applications

Applications (**application.app.k8s.io**) in Red Hat Advanced Cluster Management for Kubernetes are used for grouping Kubernetes resources that make up an application.

All of the application component resources for Red Hat Advanced Cluster Management for Kubernetes applications are defined in YAML file spec sections. When you need to create or update an application component resource, you need to create or edit the appropriate spec section to include the labels for defining your resource.

1.1.2. Channels

Channels (**channel.apps.open-cluster-management.io**) define the source repositories that a cluster can subscribe to with a subscription, and can be the following types: Git, Helm release, and Object storage repositories, and resource templates on the hub cluster.

If you have applications that require Kubernetes resources or Helm charts from channels that require authorization, such as entitled Git repositories, you can use secrets to provide access to these channels. Your subscriptions can access Kubernetes resources and Helm charts for deployment from these channels, while maintaining data security.

Channels use a namespace within the hub cluster and point to a physical place where resources are stored for deployment. Clusters can subscribe to channels for identifying the resources to deploy to each cluster.

Notes: It is best practice to create each channel in a unique namespace. However, a Git channel can share a namespace with another type of channel, including Git, Helm, and Object storage.

Resources within a channel can be accessed by only the clusters that subscribe to that channel.

1.1.3. Subscriptions

Subscriptions (**subscription.apps.open-cluster-management.io**) allow clusters to subscribe to a source repository (channel) that can be the following types: Git repository, Helm release registry, or Object storage repository.

Subscriptions can deploy application resources locally to the hub cluster, if the hub cluster is self-managed. You can then view the **local-cluster** subscription in the topology.

Subscriptions can point to a channel or storage location for identifying new or updated resource templates. The subscription operator can then download directly from the storage location and deploy to targeted managed clusters without checking the hub cluster first. With a subscription, the subscription operator can monitor the channel for new or updated resources instead of the hub cluster.

1.1.4. Placement rules

Placement rules (**placementrule.apps.open-cluster-management.io**) define the target clusters where resource templates can be deployed. Use placement rules to help you facilitate the multicluster deployment of your deployables. Placement policies are also used for governance and risk policies.

Learn more from the following documentation:

- [Application console](#)
- [Managing application resources](#)
- [Managing apps with Git repositories](#)
- [Managing apps with Helm repositories](#)
- [Managing apps with Object storage repositories](#)
- [Application advanced configuration](#)
- [Subscribing Git resources](#)
- [Setting up Ansible Tower tasks](#)
- [Channel samples](#)
- [Subscription samples](#)
- [Placement rule samples](#)
- [Application samples](#)

1.2. APPLICATION CONSOLE

The console includes a dashboard for managing the application lifecycle. You can use the console dashboard to create and manage applications and view the status of applications. Enhanced capabilities help your developers and operations personnel create, deploy, update, manage, and visualize applications across your clusters.

See the following application console capabilities:

Important: Actions are based on your assigned role. Learn about access requirements from the [Role-based access control](#) documentation.

- Visualize deployed applications across your clusters, including any associated resource repositories, subscriptions, and placement configurations.
- Create and edit applications, and subscribe resources. By default, the hub cluster can manage itself, and is named the **local cluster**. You can choose to deploy application resources to this local cluster, though deploying applications on the local cluster is not best practice.
- Use **Advanced configuration** to view or edit channels, subscriptions, and placement rules.
- Access a topology view that encompasses application resources, including resource repositories, subscriptions, placement rules, and deployed resources, including any optional pre and post deployment hooks using Ansible Tower tasks (for Git repositories).
- View individual status in the context of an application, including deployments, updates, and subscriptions.

The console includes tools that each provide different application management capabilities. These capabilities allow you to easily create, find, update, and deploy application resources.

- [Applications overview](#)
- [Resource topology](#)
- [Search](#)
- [Advanced configuration](#)

1.2.1. Applications overview

From the main *Overview* tab, see the following:

- A table that lists all applications
- The *Find resources* box to filter the applications that are listed
- The application name and namespace
- The number of remote and local clusters where resources are deployed through a subscription
- The links to repositories where the definitions for the resources that are deployed by the application are located
- An indication of Time window constraints, if any were created
- The date when the application was created
- More actions, such as **Delete application**. Actions are available if the user has permission to take action.

1.2.1.1. Single applications overview

Click on an application name in the table to view details about a single application. See the following information:

- Cluster details, such as resource status.
- Subscription details
- Resource topology

Click the **Editor** tab to edit your application and related resources.

1.2.2. Resource topology

The topology provides a visual representation of the application that was selected including the resources deployed by this application on target clusters.

- You can select any component from the topology view to view more details.
- View the deployment details for any resource deployed by this application by clicking on the resource node to open the properties view.
- View cluster CPU and memory from the cluster node, on the properties dialog.
Notes: The cluster CPU and memory percentage that is displayed is the percentage that is currently utilized. This value is rounded down, so a very small value might display as **0**.

For Helm subscriptions, see [Configuring package overrides](#) to define the appropriate **packageName** and the **packageAlias** to get an accurate topology display.

- View a successful Ansible Tower deployment if you are using Ansible tasks as prehook or posthook for the deployed application.
Click **Actions** to see the details about the Ansible task deployment, including Ansible Tower Job URL and template name. Additionally, you can see errors if your Ansible Tower deployment is not successful.
- Click **Launch resource in Search** to search for related resources.

1.2.3. Search

The console *Search* page supports searching for application resources by the component **kind** for each resource. To search for resources, use the following values:

Application resource	Kind (search parameter)
Subscription	Subscription
Channel	Channel
Secret	Secret
Placement rule	PlacementRule
Application	Application

You can also search by other fields, including name, namespace, cluster, label, and more.

From the search results, you can view identifying details for each resource, including the name, namespace, cluster, labels, and creation date.

If you have access, you can also click **Actions** in the search results and select to delete that resource.

Click the resource name in the search results to open the YAML editor and make changes. Changes that you save are applied to the resource immediately.

For more information about using search, see [Search in the console](#).

1.2.4. Advanced configuration

Click the **Advanced configuration** tab to view terminology and tables of resources for all applications. You can find resources and you can filter subscriptions, placement rules, and channels. If you have access, you can also click multiple **Actions**, such as Edit, Search, and Delete.

Select a resource to view or edit the YAML.

1.3. MANAGING APPLICATION RESOURCES

From the console, you can create applications by using Git repositories, Helm repositories, and Object storage repositories.

Important: Git Channels can share a namespace with all other channel types: Helm, Object storage, and other Git namespaces.

See the following topics to start managing apps:

- [Managing apps with Git repositories](#)
- [Managing apps with Helm repositories](#)
- [Managing apps with Object storage repositories](#)

1.3.1. Managing apps with Git repositories

When you deploy Kubernetes resources using an application, the resources are located in specific repositories. Learn how to deploy resources from Git repositories in the following procedure. Learn more about the application model at [Application model and definitions](#).

User required access: A user role that can create applications. You can only perform actions that your role is assigned. Learn about access requirements from the [Role-based access control](#) documentation.

1. From the console navigation menu, click **Manage applications**.
2. Click **Create application**.
For the following steps, select **YAML: On** to view the YAML in the console as you create your application. See YAML samples later in the topic.
3. Enter the following values in the correct fields:
 - Name: Enter a valid Kubernetes name for the application.
 - Namespace: Select a namespace from the list. You can also create a namespace by using a valid Kubernetes name if you are assigned the correct access role.

4. Choose **Git** from the list of repositories that you can use.
5. Enter the required URL path or select an existing path.
If you select an existing Git repository path, you do not need to specify connection information if this is a private repository. The connection information is pre-set and you do not need to view these values.

If you enter a new Git repository path, you can optionally enter Git connection information if this is a private Git repository.
6. Enter values for the optional fields, such as branch and folder.
7. Set any optional pre and post-deployment tasks.
Technology preview: Set the Ansible Tower secret if you have Ansible Tower jobs that you want to run before or after the subscription deploys the application resources. The Ansible Tower tasks that define Ansible jobs must be placed within prehook and posthook folders in this repository.

You can select an Ansible Tower secret from drop-down menu if a secret was created in the application namespace. In this instance, the connection information is pre-set, and you do not need to view these values.

If you enter a new Ansible Tower secret name to create a new secret, you need to enter the Ansible Tower host and token.
8. From *Select clusters to deploy*, you can update the placement rule information for your application. Choose from the following:
 - Deploy on local cluster
 - Deploy to all online clusters and local cluster
 - Deploy application resources only on clusters matching specified labels
 - You have the option to *Select existing placement configuration* if you create an application in an existing namespace with placement rules already defined.
9. From *Settings*, you can specify application behavior. To block or activate changes to your deployment during a specific time window, choose an option for *Deployment window* and your *Time window configuration*.
10. You can either choose another repository or Click **Save**.
11. You are redirected to the *Overview* page where you can view the details and topology.

1.3.1.1. Sample YAML

The following example channel definition shows an example of a channel for the Git Repository. In the following example, **secretRef** refers to the user identity used to access the Git repo that is specified in the **pathname**. If you have a public repo, you do not need the **secretRef**:

```
apiVersion: apps.open-cluster-management.io/v1
kind: Channel
metadata:
  name: hive-cluster-gitrepo
  namespace: gitops-cluster-lifecycle
spec:
```

```

type: Git
pathname: https://github.com/open-cluster-management/gitops-clusters.git
secretRef:
  name: github-gitops-clusters
---
apiVersion: v1
kind: Secret
metadata:
  name: github-gitops-clusters
  namespace: gitops-cluster-lifecycle
data:
  user: dXNlcgo=          # Value of user and accessToken is Base 64 coded.
  accessToken: cGFzc3dvcmQ

```

Note: To see REST API, use the [APIs](#).

1.3.1.2. Application Git Ops

Using a **local-cluster** placement rule, you can use a subscription to deliver configuration-related resources to the hub cluster.

These resources can be the subscriptions that configure applications and policies, run Ansible jobs, and configure your clusters after you provision, or after you import.

1.3.1.2.1. Sample Repo for GitOps

In the following example repository, you see a folder for each hub cluster. You can also create a repository for each hub cluster, or a branch for each hub cluster.

A single subscription that is defined on the hub cluster pulls in all the other configuration resources, such as hub cluster config, policies, and common applications that will configure and deploy to the hub and managed clusters.

A Git repository to store this type of information can resemble the following sample file and directory structure. See the sections for root path, managed cluster, and hub cluster:

1.3.1.2.2. GitOps root path

These files in the root path of the repository create a subscription that references this Git repository, and applies all the YAML, except what is specified in the **.kubernetesignore**. It includes these four subscription files and the **./managed-cluster-common** directory.

```

/ # Repository root directory

# The subscription that delivers all the previous content to the hub cluster:

hub-application.yaml # This represents the hub cluster configuration in the console
hub-channels.yaml   # This points to `rhacm-hub-cluster` Git repository
hub-subscriptions.yaml # This defines the time window, branch to be used, and defines which
directories containing appropriate configs, such as `hub-policies`, should be used (can be all)
hub-placement.yaml  # Points back to the `local-cluster` (hub cluster that is managed)
.kubernetesignore   # Tells the subscription to ignore hub-application.yaml, hub-channels.yaml,
hub-subscription.yaml & hub-placement.yaml

```

1.3.1.2.3. GitOps application to managed clusters

The following directories contain subscriptions that will apply applications to the *managed* clusters. These subscriptions are applied to the hub cluster through the subscription in the root directory.

In the following sample, you see one subscription that is subscribing another:

```

common-managed/
  apps/
    app-name-0/
      application.yaml
      subscription.yaml
      channel.yaml    # This points to a repository named `app-name-0`, of type Git, Helm, or Object
  Storage
    placementrule.yaml
    app-name-1/
      application.yaml
      subscription.yaml
      channel.yaml    # This points to a repository named `app-name-0`, of type Git, Helm, or Object
  Storage
    placementrule.yaml
  config/
    application.yaml  # named: `day2-config`
    subscription.yaml # Points to the `managed-cluster-common/config` parent directory
    channel.yaml      # Can point to this Git repository or a different repository with the day-two
configuration
  placementrule.yaml # Defines the clusters to target
managed-cluster-common/
  configs/           # These configurations are referenced through the `config` subscription
  certmanagement.yaml
  auth-oidc.yaml
  autoscaler.yaml
  descheduler.yaml

```

1.3.1.2.4. GitOps application to hub clusters

The following policies are applied to the hub cluster and offer both configuration for the hub cluster, as well as policies for remote clusters.

These are delivered through the root subscription, as seen in the following sample:

```

managed-cluster-common/
  policies/
    policy-0.yaml
    policy-1.yaml
  hub-policies/
    policy-0.yaml
    vault.yaml
    operators.yaml

```

1.3.1.2.5. Apply GitOps

With the previous sample combination, you can specify the following:

1. A root subscription that can be applied with a CLI command. The root subscription will subscribe back to this repository to apply all the YAML to the hub cluster.
2. The subscription from step 1, which applies application and configuration subscriptions from **common-managed/**.
3. The configuration subscription in step 2, which applies the resources defined in **managed-cluster-common/**.
4. The policies that are defined in **managed-cluster-common/** are also applied to the hub cluster by the subscription in step 1. These policies include those targeted toward the hub cluster, as well as those that target managed clusters.

1.3.2. Managing apps with Helm repositories

When you deploy Kubernetes resources using an application, the resources are located in specific repositories. Learn how to deploy resources from Helm repositories in the following procedure. Learn more about the application model at [Application model and definitions](#).

User required access: A user role that can create applications. You can only perform actions that your role is assigned. Learn about access requirements from the [Role-based access control](#) documentation.

1. From the console navigation menu, click **Manage applications**.
2. Click **Create application**.
For the following steps, select **YAML: On** to view the YAML in the console as you create your application. See YAML samples later in the topic.
3. Enter the following values in the correct fields:
 - Name: Enter a valid Kubernetes name for the application.
 - Namespace: Select a namespace from the list. You can also create a namespace by using a valid Kubernetes name if you are assigned the correct access role.
4. Choose **Helm** from the list of repositories that you can use.
5. Enter the required URL path or select an existing path, then enter the package version.
If you select an existing Helm repository path, you do not need to specify connection information if this is a private repository. The connection information is pre-set and you do not need to view these values.

If you enter a new Helm repository path, you can optionally enter Helm connection information if this is a private Helm repository.
6. From *Select clusters to deploy*, you can update the placement rule information for your application. Choose from the following:
 - Deploy on local cluster
 - Deploy to all online clusters and local cluster
 - Deploy application resources only on clusters matching specified labels
 - You have the option to *Select existing placement configuration* if you create an application in an existing namespace with placement rules already defined.

7. From *Settings*, you can specify application behavior. To block or activate changes to your deployment during a specific time window, choose an option for *Deployment window* and your *Time window configuration*.
8. You can either choose another repository or Click **Save**.
9. You are redirected to the *Overview* page where you can view the details and topology.

1.3.2.1. Sample YAML

The following example channel definition abstracts a Helm repository as a channel:

Note: For Helm, all Kubernetes resources contained within the Helm chart must have the label `release: {{ .Release.Name }}` for the application topology to be displayed properly.

```
apiVersion: v1
kind: Namespace
metadata:
  name: hub-repo
---
apiVersion: apps.open-cluster-management.io/v1
kind: Channel
metadata:
  name: helm
  namespace: hub-repo
spec:
  pathname: [https://kubernetes-charts.storage.googleapis.com/] # URL points to a valid chart URL.
  type: HelmRepo
```

The following channel definition shows another example of a Helm repository channel:

```
apiVersion: apps.open-cluster-management.io/v1
kind: Channel
metadata:
  name: predev-ch
  namespace: ns-ch
  labels:
    app: nginx-app-details
spec:
  type: HelmRepo
  pathname: https://kubernetes-charts.storage.googleapis.com/
```

Note: To see REST API, use the [APIs](#).

1.3.3. Managing apps with Object storage repositories

When you deploy Kubernetes resources using an application, the resources are located in specific repositories. Learn how to deploy resources from Object storage repositories in the following procedure. Learn more about the application model at [Application model and definitions](#).

User required access: A user role that can create applications. You can only perform actions that your role is assigned. Learn about access requirements from the [Role-based access control](#) documentation.

When you deploy Kubernetes resources using an application, the resources are located in specific repositories. Learn how to deploy resources from Git repositories in the following procedure.

1. From the console navigation menu, click **Manage applications**.
2. Click **Create application**.
For the following steps, select **YAML: On** to view the YAML in the console as you create your application. See YAML samples later in the topic.
3. Enter the following values in the correct fields:
 - Name: Enter a valid Kubernetes name for the application.
 - Namespace: Select a namespace from the list. You can also create a namespace by using a valid Kubernetes name if you are assigned the correct access role.
4. Choose **Object storage** from the list of repositories that you can use.
5. Enter the required URL path or select an existing path.
If you select an existing Object storage repository path, you do not need to specify connection information if this is a private repository. The connection information is pre-set and you do not need to view these values.

If you enter a new Object storage repository path, you can optionally enter Object storage connection information if this is a private Object storage repository.
6. Enter values for the optional fields.
7. Set any optional pre and post-deployment tasks.
8. From *Select clusters to deploy*, you can update the placement rule information for your application. Choose from the following:
 - Deploy on local cluster
 - Deploy to all online clusters and local cluster
 - Deploy application resources only on clusters matching specified labels
 - You have the option to *Select existing placement configuration* if you create an application in an existing namespace with placement rules already defined.
9. From *Settings*, you can specify application behavior. To block or activate changes to your deployment during a specific time window, choose an option for *Deployment window* and your *Time window configuration*.
10. You can either choose another repository or Click **Save**.
11. You are redirected to the *Overview* page where you can view the details and topology.

1.3.3.1. Sample YAML

The following example channel definition abstracts an object storage as a channel:

```
apiVersion: apps.open-cluster-management.io/v1
kind: Channel
metadata:
  name: dev
  namespace: ch-obj
spec:
```

```
type: Object storage
pathname: [http://9.28.236.243:31311/dev] # URL is appended with the valid bucket name, which
matches the channel name.
secretRef:
  name: miniosecret
gates:
  annotations:
    dev-ready: true
```

Note: To see REST API, use the [APIs](#).

1.4. APPLICATION ADVANCED CONFIGURATION

Within Red Hat Advanced Cluster Management for Kubernetes, applications are composed of multiple application resources. You can use channel, subscription, and placement rule resources to help you deploy, update, and manage your overall applications.

Both single and multicluster applications use the same Kubernetes specifications, but multicluster applications involve more automation of the deployment and application management lifecycle.

All of the application component resources for Red Hat Advanced Cluster Management for Kubernetes applications are defined in YAML file spec sections. When you need to create or update an application component resource, you need to create or edit the appropriate spec section to include the labels for defining your resource.

View the following application advanced configuration topics:

- [Subscribing Git resources](#)
- [Setting up Ansible Tower tasks](#)
- [Channel samples](#)
- [Subscription samples](#)
- [Placement rule samples](#)
- [Application samples](#)

1.4.1. Subscribing Git resources

A subscription administrator can change default behavior. By default, when a subscription deploys subscribed applications to target clusters, the applications are deployed to that subscription namespace, even if the application resources are associated with other namespaces.

Additionally, if an application resource exists in the cluster and was not created by the subscription, the subscription cannot apply a new resource on that existing resource. See the following processes to change default settings as the subscription administrator.

Required access: Cluster administrator

1.4.1.1. Granting users and groups subscription admin privilege

Learn how to grant subscription administrator access.

1. From the console, log in to your Red Hat OpenShift Container Platform cluster.

2. Create one or more users. See [Preparing for users](#) for information about creating users. Users that you create are administrators for the **app.open-cluster-management.io/subscription** application. With OpenShift Container Platform, a subscription administrator can change default behavior. You can group these users to represent a subscription administrative group, which is demonstrated in later examples.
3. From the terminal, log in to your Red Hat Advanced Cluster Management cluster.
4. Add the following subjects into **open-cluster-management:subscription-admin** ClusterRoleBinding with the following command:

```
oc edit clusterrolebinding open-cluster-management:subscription-admin
```

Note: Initially, **open-cluster-management:subscription-admin** ClusterRoleBinding has no subject.

Your subjects might display as the following example:

```
subjects:
- apiGroup: rbac.authorization.k8s.io
  kind: User
  name: example-name
- apiGroup: rbac.authorization.k8s.io
  kind: Group
  name: example-group-name
```

Next, see more useful examples of how a subscription administrator can change default behavior.

1.4.1.2. Application namespace example

In this example, you are logged in as a subscription administrator. Create a subscription to subscribe the sample resource YAML file from a Git repository. The example file contains subscriptions that are located within the following different namespaces:

Applicable channel types: Git

- ConfigMap **test-configmap-1** gets created in **multins** namespace.
- ConfigMap **test-configmap-2** gets created in **default** namespace.
- ConfigMap **test-configmap-3** gets created in the **subscription** namespace.

```
---
apiVersion: v1
kind: Namespace
metadata:
  name: multins
---
apiVersion: v1
kind: ConfigMap
metadata:
  name: test-configmap-1
  namespace: multins
data:
  path: resource1
```

```

---
apiVersion: v1
kind: ConfigMap
metadata:
  name: test-configmap-2
  namespace: default
data:
  path: resource2
---
apiVersion: v1
kind: ConfigMap
metadata:
  name: test-configmap-3
data:
  path: resource3

```

If the subscription was created by other users, all the ConfigMaps get created in the same namespace as the subscription.

1.4.1.3. Resource overwrite example

Applicable channel types: Git, ObjectBucket (Object storage in the console)

In this example, the following ConfigMap already exists in the target cluster.

```

apiVersion: v1
kind: ConfigMap
metadata:
  name: test-configmap-1
  namespace: sub-ns
data:
  name: user1
  age: 19

```

Subscribe the following sample resource YAML file from a Git repository and replace the existing ConfigMap:

```

apiVersion: v1
kind: ConfigMap
metadata:
  name: test-configmap-1
  namespace: sub-ns
data:
  age: 20

```

You log in as a subscription administrator and create a subscription with **apps.open-cluster-management.io/reconcile-option: replace** annotation. See the following example:

```

apiVersion: apps.open-cluster-management.io/v1
kind: Subscription
metadata:
  name: subscription-example
  namespace: sub-ns
annotations:
  apps.open-cluster-management.io/git-path: sample-resources

```

```

apps.open-cluster-management.io/reconcile-option: replace
spec:
  channel: channel-ns/somechannel
  placement:
    placementRef:
      name: dev-clusters

```

When this subscription is created by a subscription administrator and subscribes the ConfigMap resource, the existing ConfigMap is replaced by the following:

```

apiVersion: v1
kind: ConfigMap
metadata:
  name: test-configmap-1
  namespace: sub-ns
data:
  age: 20

```

If you want to subscribe the following sample resource YAML file from a Git repository and **merge** with the existing ConfigMap, use the **apps.open-cluster-management.io/reconcile-option: merge** annotation. See the following example:

```

apiVersion: apps.open-cluster-management.io/v1
kind: Subscription
metadata:
  name: subscription-example
  namespace: sub-ns
  annotations:
    apps.open-cluster-management.io/git-path: sample-resources
    apps.open-cluster-management.io/reconcile-option: merge
spec:
  channel: channel-ns/somechannel
  placement:
    placementRef:
      name: dev-clusters

```

When this subscription is created by a subscription administrator and subscribes the ConfigMap resource, the existing ConfigMap is merged, as you can see in the following example:

```

apiVersion: v1
kind: ConfigMap
metadata:
  name: test-configmap-1
  namespace: sub-ns
data:
  name: user1
  age: 20

```

When the **merge** option is used, entries from subscribed resource are either created or updated in the existing resource. No entry is removed from the existing resource.

Important: If the existing resource you want to overwrite with a subscription is automatically reconciled by another operator or controller, the resource configuration is updated by both subscription and the controller or operator. Do not use this method in this case.

1.4.1.3.1. Reconcile option

You can also use **apps.open-cluster-management.io/reconcile-option** annotation in individual resources to override the subscription-level reconcile option.

For example, if you add **apps.open-cluster-management.io/reconcile-option: replace** annotation in the subscription and add **apps.open-cluster-management.io/reconcile-option: merge** annotation in a resource YAML in the subscribed Git repository, the resource will be merged on the target cluster, while other resources are replaced.

1.4.2. Configuring package overrides

Configure package overrides for a subscription override value for the Helm chart or Kubernetes resource that is subscribed to by the subscription.

To configure a package override, specify the field within the Kubernetes resource **spec** to override as the value for the **path** field. Specify the replacement value as the value for the **value** field.

For example, if you need to override the values field within the **spec** for a Helm release for a subscribed Helm chart, you need to set the value for the **path** field in your subscription definition to **spec**.

```
packageOverrides:
- packageName: nginx-ingress
  packageOverrides:
  - path: spec
    value: my-override-values
```

The contents for the **value** field are used to override the values within the **spec** field of the **Helm** spec.

- For a Helm release, override values for the **spec** field are merged into the Helm release **values.yaml** file to override the existing values. This file is used to retrieve the configurable variables for the Helm release.
- If you need to override the release name for a Helm release, include the **packageOverride** section within your definition. Define the **packageAlias** for the Helm release by including the following fields:
 - **packageName** to identify the Helm chart.
 - **packageAlias** to indicate that you are overriding the release name.

By default, if no Helm release name is specified, the Helm chart name is used to identify the release. In some cases, such as when there are multiple releases subscribed to the same chart, conflicts can occur. The release name must be unique among the subscriptions within a namespace. If the release name for a subscription that you are creating is not unique, an error occurs. You must set a different release name for your subscription by defining a **packageOverride**. If you want to change the name within an existing subscription, you must first delete that subscription and then recreate the subscription with the preferred release name.

+

```
packageOverrides:
- packageName: nginx-ingress
  packageAlias: my-helm-release-name
```


1.4.3. Setting up Ansible Tower tasks (Technology preview)

Red Hat Advanced Cluster Management is integrated with Ansible Tower automation so that you can create prehook and posthook AnsibleJob instances for Git subscription application management. With Ansible Tower jobs, you can automate tasks and integrate with external services, such as Slack and PagerDuty services. Your Git repository resource root path will contain **prehook** and **posthook** directories for Ansible Tower jobs that run as part of deploying the app, updating the app, or removing the app from a cluster.

Required access: Cluster administrator

1.4.3.1. Prerequisites

- OpenShift Container Platform 4.5 or later
- You must have Ansible Tower version 3.7.3 or a later version installed. It is best practice to install the latest supported version of Ansible Tower. See [Red Hat AnsibleTower documentation](#) for more details.
- Install the Ansible Automation Platform Resource Operator to connect Ansible jobs to the lifecycle of Git subscriptions. For best results when using the AnsibleJob to launch Ansible Tower jobs, the Ansible Tower job template should be idempotent when it is run.

Check **PROMPT ON LAUNCH** on the template for both INVENTORY and EXTRA VARIABLES. See [Job templates](#) for more information.

1.4.3.2. Install Ansible Automation Platform Resource Operator:

1. Log in to your OpenShift Container Platform cluster console.
2. Click **OperatorHub** in the console navigation.
3. Search for and install the *Ansible Automation Platform Resource Operator*.

1.4.3.3. Obtain the Ansible Tower URL and token

The Ansible Tower URL is the same URL that is used to log in to Tower. This is required by the *Application console* or the Tower access secret when configuring an application with Ansible prehooks and posthooks.

See the following example URL: <https://ansible-tower-web-svc-tower.apps.my-openshift-cluster.com>.

1.4.3.4. Obtaining a token

1. Log in to your Ansible Tower console.
2. Click **Users** in the console navigation.
3. Search for the correct user.
4. Click the **Edit user** icon.
5. Click **TOKENS** in the user section.
6. Click the + button to add a token.

7. Leave the **APPLICATION** field blank.
8. In the **DESCRIPTION** field, provide your intended use for this token.
9. Select **Write** in the **SCOPE** drop-down menu.
10. Click **SAVE** and record the **TOKEN** that is provided.

1.4.3.5. Ansible integration

You can integrate Ansible Tower jobs into Git subscriptions. For instance, for a database front-end and back-end application, the database is required to be instantiated using Ansible Tower with an Ansible Job, and the application is installed by a Git subscription. The database is instantiated *before* you deploy the front-end and back-end application with the subscription.

The application subscription operator is enhanced to define two subfolders: **prehook** and **posthook**. Both folders are in the Git repository resource root path and contain all prehook and posthook Ansible jobs, respectively.

When the Git subscription is created, all of the pre and post AnsibleJob resources are parsed and stored in memory as an object. The application subscription controller decides when to create the pre and post AnsibleJob instances.

1.4.3.6. Ansible operator components

When you create a subscription CR, the `Git-branch` and `Git-path` points to a Git repository root location. In the Git root location, the two subfolders **prehook** and **posthook** should contain at least one **Kind:AnsibleJob** resource.

1.4.3.6.1. Prehook

The application subscription controller searches all the **Kind:AnsibleJob** CRs in the prehook folder as the prehook AnsibleJob objects, then generates a new prehook AnsibleJob instance. The new instance name is the prehook AnsibleJob object name and a random suffix string.

See an example instance name: **database-sync-1-2913063**.

The application subscription controller queues the reconcile request again in a 1 minute loop, where it checks the prehook AnsibleJob **status.ansibleJobResult**. When the prehook **status.ansibleJobResult.status** is **successful**, the application subscription continues to deploy the main subscription.

1.4.3.6.2. Posthook

When the app subscription status is updated, if the subscription status is subscribed or propagated to all target clusters in subscribed status, the app subscription controller searches all of the **AnsibleJob Kind** CRs in the posthook folder as the posthook AnsibleJob objects. Then, it generates new posthook **AnsibleJob** instances. The new instance name is the posthook **AnsibleJob** object name and a random suffix string.

See an example instance name: **service-ticket-1-2913849**.

1.4.3.6.3. Ansible placement rules

With a valid prehook AnsibleJob, the subscription launches the prehook AnsibleJob regardless of the

decision from the placement rule. For example, you can have a prehook AnsibleJob that failed to propagate a placement rule subscription. When the placement rule decision changes, new prehook and posthook AnsibleJob instances are created.

1.4.3.7. Ansible configuration

You can configure Ansible Tower configurations with the following tasks:

1.4.3.7.1. Ansible secrets

You must create an Ansible Tower secret CR in the same subscription namespace. The Ansible Tower secret is limited to the same subscription namespace.

Create the secret from the console by filling in the **Ansible Tower secret name** section. To create the secret using terminal, edit and apply the following **yaml**:

Run the following command to add your YAML file:

```
oc apply -f
```

See the following YAML sample:

Note: The **namespace** is the same namespace as the subscription namespace. The **stringData:token** and **host** are from the Ansible Tower.

```
apiVersion: v1
kind: Secret
metadata:
  name: toweraccess
  namespace: same-as-subscription
type: Opaque
stringData:
  token: ansible-tower-api-token
  host: https://ansible-tower-host-url
```

When the app subscription controller creates prehook and posthook AnsibleJobs, if the secret from subscription **spec.hooksecretref** is available, then it is sent to the AnsibleJob CR **spec.tower_auth_secret** and the AnsibleJob can access the Ansible Tower.

1.4.3.8. Set secret reconciliation

For a main-sub subscription with prehook and posthook AnsibleJobs, the main-sub subscription should be reconciled after all prehook and posthook AnsibleJobs or main subscription are updated in the Git repository.

Prehook AnsibleJobs and the main subscription continuously reconcile and relaunch a new pre-AnsibleJob instance.

1. After the pre-AnsibleJob is done, re-run the main subscription.
2. If there is any specification change in the main subscription, re-deploy the subscription. The main subscription status should be updated to align with the redeployment procedure.
3. Reset the hub subscription status to **nil**. The subscription is refreshed along with the subscription deployment on target clusters.

When the deployment is finished on the target cluster, the subscription status on the target cluster is updated to **"subscribed"** or **"failed"**, and is synced to the hub cluster subscription status.

4. After the main subscription is done, relaunch a new post-AnsibleJob instance.
5. Verify that the DONE subscription is updated. See the following output:
 - `subscription.status == "subscribed"`
 - `subscription.status == "propagated"` with all of the target clusters **"subscribed"**

When an AnsibleJob CR is created, A Kubernetes job CR is created to launch an Ansible Tower job by communicating to the target Ansible Tower. When the job is complete, the final status for the job is returned to AnsibleJob **status.ansibleJobResult**.

Notes:

The AnsibleJob `status.conditions` is reserved by the Ansible Job operator for storing the creation of Kubernetes job result. The `status.conditions` does not reflect the actual Ansible Tower job status.

The subscription controller checks the Ansible Tower job status by the **AnsibleJob.status.ansibleJobResult** instead of **AnsibleJob.status.conditions**.

As previously mentioned in the prehook and posthook AnsibleJob workflow, when the main subscription is updated in Git repository, a new prehook and posthook AnsibleJob instance is created. As a result, one main subscription can link to multiple AnsibleJob instances.

Four fields are defined in `subscription.status.ansibleJobs`:

- `lastPrehookJobs`: The most recent prehook AnsibleJobs
- `prehookJobsHistory`: All the prehook AnsibleJobs history
- `lastPosthookJobs`: The most recent posthook AnsibleJobs
- `posthookJobsHistory`: All the posthook AnsibleJobs history

1.4.3.9. Ansible sample YAML

See the following sample of an AnsibleJob **.yaml** file in a Git prehook and posthook folder:

```
apiVersion: tower.ansible.com/v1alpha1
kind: AnsibleJob
metadata:
  generateName: demo-job-001
  namespace: default
spec:
  tower_auth_secret: toweraccess
  job_template_name: Demo Job Template
  extra_vars:
    cost: 6.88
    ghosts: ["inky","pinky","clyde","sue"]
    is_enable: false
    other_variable: foo
  pacman: mrs
  size: 8
```

```

targets_list:
- aaa
- bbb
- ccc
version: 1.23.45

```

1.4.4. Channel samples

View samples and YAML definitions that you can use to build your files. Channels (**channel.apps.open-cluster-management.io**) provide you with improved continuous integration and continuous delivery capabilities for creating and managing your Red Hat Advanced Cluster Management for Kubernetes applications.

To use the Kubernetes CLI tool, see the following procedure:

- a. Compose and save your application YAML file with your preferred editing tool.
- b. Run the following command to apply your file to an API server. Replace **filename** with the name of your file:

```
kubectl apply -f filename.yaml
```

- c. Verify that your application resource is created by running the following command:

```
kubectl get Application
```

Note: Kubernetes namespace (Namespace) channel is not available this release.

1.4.4.1. Channel YAML structure

The following YAML structures show the required fields for a channel and some of the common optional fields. Your YAML structure needs to include some required fields and values. Depending on your application management requirements, you might need to include other optional fields and values. You can compose your own YAML content with any tool.

```

apiVersion: apps.open-cluster-management.io/v1
kind: Channel
metadata:
  name:
  namespace: # Each channel needs a unique namespace, except Git channel.
spec:
  sourceNamespaces:
  type:
  pathname:
  secretRef:
    name:
  gates:
    annotations:
  labels:

```

1.4.4.2. Channel YAML table

Field	Description
apiVersion	Required. Set the value to apps.open-cluster-management.io/v1 .
kind	Required. Set the value to Channel to indicate that the resource is a channel.
metadata.name	Required. The name of the channel.
metadata.namespace	Required. The namespace for the channel; Each channel needs a unique namespace, except Git channel.
spec.sourceNamespaces	Optional. Identifies the namespace that the channel controller monitors for new or updated deployables to retrieve and promote to the channel.
spec.type	Required. The channel type. The supported types are: HelmRepo , Git , and ObjectBucket (Object storage in the console)
spec.pathname	Required for HelmRepo , Git , ObjectBucket channels. For a HelmRepo channel, set the value to be the URL for the Helm repository. For an ObjectBucket channel, set the value to be the URL for the Object storage. For a Git channel, set the value to be the HTTPS URL for the Git repository.
spec.secretRef.name	Optional. Identifies a Kubernetes Secret resource to use for authentication, such as for accessing a repository or chart. You can use a secret for authentication with only HelmRepo , ObjectBucket , and Git type channels.
spec.gates	Optional. Defines requirements for promoting a deployable within the channel. If no requirements are set, any deployable that is added to the channel namespace or source is promoted to the channel. gates do not apply for HelmRepo and Git channel types, only for ObjectBucket channel types.
spec.gates.annotations	Optional. The annotations for the channel. Deployables must have matching annotations to be included in the channel.
metadata.labels	Optional. The labels for the channel.

The definition structure for a channel can resemble the following YAML content:

```

apiVersion: apps.open-cluster-management.io/v1
kind: Channel
metadata:
  name: predev-ch
  namespace: ns-ch
  labels:
    app: nginx-app-details
spec:
  type: HelmRepo
  pathname: https://kubernetes-charts.storage.googleapis.com/

```

1.4.4.3. Object storage bucket (ObjectBucket) channel

The following example channel definition abstracts an Object storage bucket as a channel:

```

apiVersion: apps.open-cluster-management.io/v1
kind: Channel
metadata:
  name: dev
  namespace: ch-obj
spec:
  type: ObjectBucket
  pathname: [http://9.28.236.243:31311/dev] # URL is appended with the valid bucket name, which
  matches the channel name.
  secretRef:
    name: miniosecret
  gates:
    annotations:
      dev-ready: true

```

1.4.4.4. Helm repository (HelmRepo) channel

The following example channel definition abstracts a Helm repository as a channel:

```

apiVersion: v1
kind: Namespace
metadata:
  name: hub-repo
---
apiVersion: apps.open-cluster-management.io/v1
kind: Channel
metadata:
  name: Helm
  namespace: hub-repo
spec:
  pathname: [https://9.21.107.150:8443/helm-repo/charts] # URL points to a valid chart URL.
  configMapRef:
    name: insecure-skip-verify
  type: HelmRepo
---
apiVersion: v1
data:
  insecureSkipVerify: "true"
kind: ConfigMap

```

```

metadata:
  name: insecure-skip-verify
  namespace: hub-repo

```

The following channel definition shows another example of a Helm repository channel:

Note: For Helm, all Kubernetes resources contained within the Helm chart must have the label `release.{{ .Release.Name }}` for the application topology to be displayed properly.

```

apiVersion: apps.open-cluster-management.io/v1
kind: Channel
metadata:
  name: predev-ch
  namespace: ns-ch
  labels:
    app: nginx-app-details
spec:
  type: HelmRepo
  pathname: https://kubernetes-charts.storage.googleapis.com/

```

1.4.4.5. Git (Git) repository channel

The following example channel definition shows an example of a channel for the Git Repository. In the following example, **secretRef** refers to the user identity used to access the Git repo that is specified in the **pathname**. If you have a public repo, you do not need the **secretRef**:

```

apiVersion: apps.open-cluster-management.io/v1
kind: Channel
metadata:
  name: hive-cluster-gitrepo
  namespace: gitops-cluster-lifecycle
spec:
  type: Git
  pathname: https://github.com/open-cluster-management/gitops-clusters.git
  secretRef:
    name: github-gitops-clusters
---
apiVersion: v1
kind: Secret
metadata:
  name: github-gitops-clusters
  namespace: gitops-cluster-lifecycle
data:
  user: dXNlcgo= # Value of user and accessToken is Base 64 coded.
  accessToken: cGFzc3dvcmQ

```

1.4.5. Secret samples

Secrets (**Secret**) are Kubernetes resources that you can use to store authorization and other sensitive information, such as passwords, OAuth tokens, and SSH keys. By storing this information as secrets, you can separate the information from the application components that require the information to improve your data security.

To use the Kubernetes CLI tool, see the following procedure:

- a. Compose and save your application YAML file with your preferred editing tool.
- b. Run the following command to apply your file to an API server. Replace **filename** with the name of your file:

```
kubectl apply -f filename.yaml
```

- c. Verify that your application resource is created by running the following command:

```
kubectl get Application
```

The definition structure for a secret can resemble the following YAML content:

1.4.5.1. Secret YAML structure

```
apiVersion: v1
kind: Secret
metadata:
  annotations:
    apps.open-cluster-management.io/deployables: "true"
  name: [secret-name]
  namespace: [channel-namespace]
data:
  AccessKeyID: [ABCdeF1=] #Base64 encoded
  SecretAccessKey: [gHljk2lmnoPQRST3uvw==] #Base64 encoded
```

1.4.6. Subscription samples

View samples and YAML definitions that you can use to build your files. As with channels, subscriptions (**subscription.apps.open-cluster-management.io**) provide you with improved continuous integration and continuous delivery capabilities for application management.

To use the Kubernetes CLI tool, see the following procedure:

- a. Compose and save your application YAML file with your preferred editing tool.
- b. Run the following command to apply your file to an apiserver. Replace **filename** with the name of your file:

```
kubectl apply -f filename.yaml
```

- c. Verify that your application resource is created by running the following command:

```
kubectl get Application
```

1.4.6.1. Subscription YAML structure

The following YAML structure shows the required fields for a subscription and some of the common optional fields. Your YAML structure needs to include certain required fields and values.

Depending on your application management requirements, you might need to include other optional fields and values. You can compose your own YAML content with any tool:

```

apiVersion: apps.open-cluster-management.io/v1
kind: Subscription
metadata:
  name:
  namespace:
  labels:
spec:
  sourceNamespace:
  source:
  channel:
  name:
  packageFilter:
    version:
    labelSelector:
      matchLabels:
        package:
        component:
  annotations:
  packageOverrides:
  - packageName:
    packageAlias:
  - path:
    value:
  placement:
    local:
    clusters:
      name:
    clusterSelector:
  placementRef:
    name:
    kind: PlacementRule
  overrides:
    clusterName:
    clusterOverrides:
      path:
      value:

```

1.4.6.2. Subscription YAML table

Field	Description
apiVersion	Required. Set the value to apps.open-cluster-management.io/v1 .
kind	Required. Set the value to Subscription to indicate that the resource is a subscription.
metadata.name	Required. The name for identifying the subscription.
metadata.namespace	Required. The namespace resource to use for the subscription.

Field	Description
metadata.labels	Optional. The labels for the subscription.
spec.channel	Optional. The namespace name ("Namespace/Name") that defines the channel for the subscription. Define either the channel , or the source , or the sourceNamespace field. In general, use the channel field to point to the channel instead of using the source or sourceNamespace fields. If more than one field is defined, the first field that is defined is used.
spec.sourceNamespace	Optional. The source namespace where deployables are stored on the hub cluster. Use this field only for namespace channels. Define either the channel , or the source , or the sourceNamespace field. In general, use the channel field to point to the channel instead of using the source or sourceNamespace fields.
spec.source	Optional. The path name ("URL") to the Helm repository where deployables are stored. Use this field for only Helm repository channels. Define either the channel , or the source , or the sourceNamespace field. In general, use the channel field to point to the channel instead of using the source or sourceNamespace fields.
spec.name	Required for HelmRepo type channels, but optional for ObjectBucket type channels. The specific name for the target Helm chart or deployable within the channel. If neither the name or packageFilter are defined for channel types where the field is optional, all deployables are found and the latest version of each deployable is retrieved.
spec.packageFilter	Optional. Defines the parameters to use to find target deployables or a subset of a deployables. If multiple filter conditions are defined, a deployable must meet all filter conditions.
spec.packageFilter.version	Optional. The version or versions for the deployable. You can use a range of versions in the form >1.0 , or <3.0 . By default, the version with the most recent "creationTimestamp" value is used.
spec.packageFilter.annotations	Optional. The annotations for the deployable.

Field	Description
spec.packageOverrides	Optional. Section for defining overrides for the Kubernetes resource that is subscribed to by the subscription, such as a Helm chart, deployable, or other Kubernetes resource within a channel.
spec.packageOverrides.packageName	Optional, but required for setting an override. Identifies the Kubernetes resource that is being overwritten.
spec.packageOverrides.packageAlias	Optional. Gives an alias to the Kubernetes resource that is being overwritten.
spec.packageOverrides.packageOverrides	Optional. The configuration of parameters and replacement values to use to override the Kubernetes resource.
spec.placement	Required. Identifies the subscribing clusters where deployables need to be placed, or the placement rule that defines the clusters. Use the placement configuration to define values for multicluster deployments.
spec.placement.local	Optional, but required for a stand-alone cluster or cluster that you want to manage directly. Defines whether the subscription must be deployed locally. Set the value to true to have the subscription synchronize with the specified channel. Set the value to false to prevent the subscription from subscribing to any resources from the specified channel. Use this field when your cluster is a stand-alone cluster or you are managing this cluster directly. If your cluster is part of a multicluster and you do not want to manage the cluster directly, use only one of clusters , clusterSelector , or placementRef to define where your subscription is to be placed. If your cluster is the Hub of a multicluster and you want to manage the cluster directly, you must register the Hub as a managed cluster before the subscription operator can subscribe to resources locally.
spec.placement.clusters	Optional. Defines the clusters where the subscription is to be placed. Only one of clusters , clusterSelector , or placementRef is used to define where your subscription is to be placed for a multicluster. If your cluster is a stand-alone cluster that is not your hub cluster, you can also use local cluster .

Field	Description
spec.placement.clusters.name	Optional, but required for defining the subscribing clusters. The name or names of the subscribing clusters.
spec.placement.clusterSelector	Optional. Defines the label selector to use to identify the clusters where the subscription is to be placed. Use only one of clusters , clusterSelector , or placementRef to define where your subscription is to be placed for a multicluster. If your cluster is a stand-alone cluster that is not your hub cluster, you can also use local cluster .
spec.placement.placementRef	Optional. Defines the placement rule to use for the subscription. Use only one of clusters , clusterSelector , or placementRef to define where your subscription is to be placed for a multicluster. If your cluster is a stand-alone cluster that is not your Hub cluster, you can also use local cluster .
spec.placement.placementRef.name	Optional, but required for using a placement rule. The name of the placement rule for the subscription.
spec.placement.placementRef.kind	Optional, but required for using a placement rule. Set the value to PlacementRule to indicate that a placement rule is used for deployments with the subscription.
spec.overrides	Optional. Any parameters and values that need to be overridden, such as cluster-specific settings.
spec.overrides.clusterName	Optional. The name of the cluster or clusters where parameters and values are being overridden.
spec.overrides.clusterOverrides	Optional. The configuration of parameters and values to override.
spec.timeWindow	Optional. Defines the settings for configuring a time window when the subscription is active or blocked.
spec.timeWindow.type	Optional, but required for configuring a time window. Indicates whether the subscription is active or blocked during the configured time window. Deployments for the subscription occur only when the subscription is active.

Field	Description
spec.timeWindow.location	Optional, but required for configuring a time window. The time zone of the configured time range for the time window. All time zones must use the Time Zone (tz) database name format. For more information, see Time Zone Database .
spec.timeWindow.daysOfWeek	Optional, but required for configuring a time window. Indicates the days of the week when the time range is applied to create a time window. The list of days must be defined as an array, such as daysOfWeek: ["Monday", "Wednesday", "Friday"] .
spec.timeWindow.hours	Optional, but required for configuring a time window. Defined the time range for the time window. A start time and end time for the hour range must be defined for each time window. You can define multiple time window ranges for a subscription.
spec.timeWindow.hours.start	Optional, but required for configuring a time window. The timestamp that defines the beginning of the time window. The timestamp must use the Go programming language Kitchen format "hh:mmpm" . For more information, see Constants .
spec.timeWindow.hours.end	Optional, but required for configuring a time window. The timestamp that defines the ending of the time window. The timestamp must use the Go programming language Kitchen format "hh:mmpm" . For more information, see Constants .

Notes:

- When you are defining your YAML, a subscription can use **packageFilters** to point to multiple Helm charts, deployables, or other Kubernetes resources. The subscription, however, only deploys the latest version of one chart, or deployable, or other resource.
- For time windows, when you are defining the time range for a window, the start time must be set to occur before the end time. If you are defining multiple time windows for a subscription, the time ranges for the windows cannot overlap. The actual time ranges are based on the **subscription-controller** container time, which can be set to a different time and location than the time and location that you are working within.
- Within your subscription spec, you can also define the placement of a Helm release as part of the subscription definition. Each subscription can reference an existing placement rule, or define a placement rule directly within the subscription definition.
- When you are defining where to place your subscription in the **spec.placement** section, use only one of **clusters**, **clusterSelector**, or **placementRef** for a multicluster environment.
- If you include more than one placement setting, one setting is used and others are ignored. The following priority is used to determine which setting the subscription operator uses:

- a. **placementRef**
- b. **clusters**
- c. **clusterSelector**

Your subscription can resemble the following YAML content:

```
apiVersion: apps.open-cluster-management.io/v1
kind: Subscription
metadata:
  name: nginx
  namespace: ns-sub-1
  labels:
    app: nginx-app-details
spec:
  channel: ns-ch/predev-ch
  name: nginx-ingress
  packageFilter:
    version: "1.36.x"
  placement:
    placementRef:
      kind: PlacementRule
      name: towhichcluster
  overrides:
    - clusterName: "/"
      clusterOverrides:
        - path: "metadata.namespace"
          value: default
```

1.4.6.3. Subscription file samples

```
apiVersion: apps.open-cluster-management.io/v1
kind: Subscription
metadata:
  name: nginx
  namespace: ns-sub-1
  labels:
    app: nginx-app-details
spec:
  channel: ns-ch/predev-ch
  name: nginx-ingress
```

1.4.6.3.1. Subscription time window example

The following example subscription includes multiple configured time windows. A time window occurs between 10:20 AM and 10:30 AM every Monday, Wednesday, and Friday. A time window also occurs between 12:40 PM and 1:40 PM every Monday, Wednesday, and Friday. The subscription is active only during these six weekly time windows for deployments to begin.

```
apiVersion: apps.open-cluster-management.io/v1
kind: Subscription
metadata:
  name: nginx
```

```

namespace: ns-sub-1
labels:
  app: nginx-app-details
spec:
  channel: ns-ch/predev-ch
  name: nginx-ingress
  packageFilter:
    version: "1.36.x"
  placement:
    placementRef:
      kind: PlacementRule
      name: towwhichcluster
  timewindow:
    windowtype: "active" #Enter active or blocked depending on the purpose of the type.
    location: "America/Los_Angeles"
    daysofweek: ["Monday", "Wednesday", "Friday"]
    hours:
      - start: "10:20AM"
        end: "10:30AM"
      - start: "12:40PM"
        end: "1:40PM"

```

1.4.6.3.2. Subscription with overrides example

The following example includes package overrides to define a different release name of the Helm release for Helm chart. A package override setting is used to set the name **my-nginx-ingress-releaseName** as the different release name for the **nginx-ingress** Helm release.

```

apiVersion: apps.open-cluster-management.io/v1
kind: Subscription
metadata:
  name: simple
  namespace: default
spec:
  channel: ns-ch/predev-ch
  name: nginx-ingress
  packageOverrides:
    - packageName: nginx-ingress
      packageAlias: my-nginx-ingress-releaseName
  packageOverrides:
    - path: spec
      value:
        defaultBackend:
          replicaCount: 3
  placement:
    local: false

```

1.4.6.3.3. Helm repository subscription example

The following subscription automatically pulls the latest **nginx** Helm release for the version **1.36.x**. The Helm release deployable is placed on the **my-development-cluster-1** cluster when a new version is available in the source Helm repository.

The **spec.packageOverrides** section shows optional parameters for overriding values for the Helm release. The override values are merged into the Helm release **values.yaml** file, which is used to retrieve the configurable variables for the Helm release.

```

apiVersion: apps.open-cluster-management.io/v1
kind: Subscription
metadata:
  name: nginx
  namespace: ns-sub-1
  labels:
    app: nginx-app-details
spec:
  channel: ns-ch/predev-ch
  name: nginx-ingress
  packageFilter:
    version: "1.36.x"
  placement:
    clusters:
      - name: my-development-cluster-1
  packageOverrides:
  - packageName: my-server-integration-prod
  packageOverrides:
  - path: spec
    value:
      persistence:
        enabled: false
        useDynamicProvisioning: false
      license: accept
      tls:
        hostname: my-mcm-cluster.icp
      sso:
        registrationImage:
          pullSecret: hub-repo-docker-secret

```

1.4.6.3.4. Git repository subscription example

1.4.6.3.4.1. Subscribing specific branch and directory of Git repository

```

apiVersion: apps.open-cluster-management.io/v1
kind: Subscription
metadata:
  name: sample-subscription
  namespace: default
  annotations:
    apps.open-cluster-management.io/git-path: sample_app_1/dir1
    apps.open-cluster-management.io/git-branch: branch1
spec:
  channel: default/sample-channel
  placement:
    placementRef:
      kind: PlacementRule
      name: dev-clusters

```

In this example subscription, the annotation **apps.open-cluster-management.io/git-path** indicates that the subscription subscribes to all Helm charts and Kubernetes resources within the **sample_app_1/dir1**

directory of the Git repository that is specified in the channel. The subscription subscribes to **master** branch by default. In this example subscription, the annotation **apps.open-cluster-management.io/git-branch: branch1** is specified to subscribe to **branch1** branch of the repository.

1.4.6.3.4.2. Adding a `.kubernetesignore` file

You can include a `.kubernetesignore` file within your Git repository root directory, or within the **apps.open-cluster-management.io/git-path** directory that is specified in subscription's annotations.

You can use this `.kubernetesignore` file to specify patterns of files or subdirectories, or both, to ignore when the subscription deploys Kubernetes resources or Helm charts from the repository.

You can also use the `.kubernetesignore` file for fine-grain filtering to selectively apply Kubernetes resources. The pattern format of the `.kubernetesignore` file is the same as a `.gitignore` file.

If the **apps.open-cluster-management.io/git-path** annotation is not defined, the subscription looks for a `.kubernetesignore` file in the repository root directory. If the **apps.open-cluster-management.io/git-path** field is defined, the subscription looks for the `.kubernetesignore` file in the **apps.open-cluster-management.io/git-path** directory. Subscriptions do not search in any other directory for a `.kubernetesignore` file.

1.4.6.3.4.3. Applying Kustomize

If there is `kustomization.yaml` or `kustomization.yml` file in a subscribed Git folder, kustomize is applied.

You can use **spec.packageOverrides** to override **kustomization** at the subscription deployment time.

```
apiVersion: apps.open-cluster-management.io/v1
kind: Subscription
metadata:
  name: example-subscription
  namespace: default
spec:
  channel: some/channel
  packageOverrides:
    - packageName: kustomization
      packageOverrides:
        - value: |
patchesStrategicMerge:
  - patch.yaml
```

In order to override `kustomization.yaml` file, **packageName: kustomization** is required in **packageOverrides**. The override either adds new entries or updates existing entries. It does not remove existing entries.

1.4.6.3.4.4. Enabling Git WebHook

By default, a Git channel subscription clones the Git repository specified in the channel every minute and applies changes when the commit ID has changed. Alternatively, you can configure your subscription to apply changes only when the Git repository sends repo PUSH and PULL webhook event notifications.

In order to configure webhook in a Git repository, you need a target webhook payload URL and optionally a secret.

1.4.6.3.4.4.1. Payload URL

Create a route (ingress) in the hub cluster to expose the subscription operator's webhook event listener service.

```
oc create route passthrough --service=multicluster-operators-subscription -n open-cluster-management
```

Then, use **oc get route multicluster-operators-subscription -n open-cluster-management** command to find the externally-reachable hostname. The webhook payload URL is <https://<externally-reachable-hostname>/webhook>.

1.4.6.3.4.4.2. Webhook secret

Webhook secret is optional. Create a Kubernetes secret in the channel namespace. The secret must contain **data.secret**. See the following example:

```
apiVersion: v1
kind: Secret
metadata:
  name: my-github-webhook-secret
data:
  secret: BASE64_ENCODED_SECRET
```

The value of **data.secret** is the base-64 encoded WebHook secret you are going to use.

Best practice: Use a unique secret for each Git repository.

1.4.6.3.4.4.3. Configuring WebHook in Git repository

Use the payload URL and webhook secret to configure WebHook in your Git repository.

1.4.6.3.4.4.4. Enable WebHook event notification in channel

Annotate the subscription channel. See the following example:

```
oc annotate channel.apps.open-cluster-management.io <channel name> apps.open-cluster-management.io/webhook-enabled="true"
```

If you used a secret to configure WebHook, annotate the channel with this as well where **<the_secret_name>** is the kubernetes secret name containing webhook secret.

```
oc annotate channel.apps.open-cluster-management.io <channel name> apps.open-cluster-management.io/webhook-secret="<the_secret_name>"
```

1.4.6.3.4.4.5. Subscriptions of webhook-enabled channel

No webhook specific configuration is needed in subscriptions.

1.4.7. Placement rule samples

Placement rules (**placementrule.apps.open-cluster-management.io**) define the target clusters where deployables can be deployed. Use placement rules to help you facilitate the multicluster deployment of your deployables.

To use the Kubernetes CLI tool, see the following procedure:

- a. Compose and save your application YAML file with your preferred editing tool.
- b. Run the following command to apply your file to an API server. Replace **filename** with the name of your file:

```
kubectl apply -f filename.yaml
```

- c. Verify that your application resource is created by running the following command:

```
kubectl get Application
```

1.4.7.1. Placement rule YAML structure

The following YAML structure shows the required fields for a placement rule and some of the common optional fields. Your YAML structure needs to include some required fields and values. Depending on your application management requirements, you might need to include other optional fields and values. You can compose your own YAML content with any tool.

```
apiVersion: apps.open-cluster-management.io/v1
kind: PlacementRule
name:
namespace:
resourceVersion:
labels:
  app:
  chart:
  release:
  heritage:
selfLink:
uid:
spec:
  clusterSelector:
    matchLabels:
      datacenter:
      environment:
  clusterReplicas:
  clusterConditions:
ResourceHint:
  type:
  order:
Policies:
```

1.4.7.2. Placement rule YAML values table

Field	Description
-------	-------------

Field	Description
apiVersion	Required. Set the value to apps.open-cluster-management.io/v1 .
kind	Required. Set the value to PlacementRule to indicate that the resource is a placement rule.
metadata.name	Required. The name for identifying the placement rule.
metadata.namespace	Required. The namespace resource to use for the placement rule.
metadata.resourceVersion	Optional. The version of the placement rule resource.
metadata.labels	Optional. The labels for the placement rule.
spec.clusterSelector	Optional. The labels for identifying the target clusters
spec.clusterSelector.matchLabels	Optional. The labels that must exist for the target clusters.
status.decisions	Optional. Defines the target clusters where deployables are placed.
status.decisions.clusterName	Optional. The name of a target cluster
status.decisions.clusterNamespace	Optional. The namespace for a target cluster.
spec.clusterReplicas	Optional. The number of replicas to create.
spec.clusterConditions	Optional. Define any conditions for the cluster.
spec.ResourceHint	Optional. If more than one cluster matches the labels and values that you provided in the previous fields, you can specify a resource specific criteria to select the clusters. For example, you can select the cluster with the most available CPU cores.
spec.ResourceHint.type	Optional. Set the value to either cpu to select clusters based on available CPU cores or memory to select clusters based on available memory resources.
spec.ResourceHint.order	Optional. Set the value to either asc for ascending order, or desc for descending order.

Field	Description
spec.Policies	Optional. The policy filters for the placement rule.

1.4.7.3. Placement rule sample files

Existing placement rules can include the following fields that indicate the status for the placement rule. This status section is appended after the **spec** section in the YAML structure for a rule.

```
status:
  decisions:
    clusterName:
    clusterNamespace:
```

Field	Description
status	The status information for the placement rule.
status.decisions	Defines the target clusters where deployables are placed.
status.decisions.clusterName	The name of a target cluster
status.decisions.clusterNamespace	The namespace for a target cluster.

- Example 1

```
apiVersion: apps.open-cluster-management.io/v1
kind: PlacementRule
metadata:
  name: gbapp-gbapp
  namespace: development
  labels:
    app: gbapp
spec:
  clusterSelector:
    matchLabels:
      environment: Dev
  clusterReplicas: 1
status:
  decisions:
    - clusterName: local-cluster
      clusterNamespace: local-cluster
```

- Example 2

```
apiVersion: apps.open-cluster-management.io/v1
kind: PlacementRule
metadata:
  name: towwhichcluster
```

```

namespace: ns-sub-1
labels:
  app: nginx-app-details
spec:
  clusterReplicas: 1
  clusterConditions:
    - type: ManagedClusterConditionAvailable
      status: "True"
  clusterSelector:
    matchExpressions:
      - key: environment
        operator: In
        values:
          - dev

```

1.4.8. Application samples

View samples and YAML definitions that you can use to build your files. Applications (**Application.app.k8s.io**) in Red Hat Advanced Cluster Management for Kubernetes are used for viewing the application components.

To use the Kubernetes CLI tool, see the following procedure:

- a. Compose and save your application YAML file with your preferred editing tool.
- b. Run the following command to apply your file to an API server. Replace **filename** with the name of your file:

```
kubectl apply -f filename.yaml
```

- c. Verify that your application resource is created by running the following command:

```
kubectl get Application
```

1.4.8.1. Application YAML structure

To compose the application definition YAML content for creating or updating an application resource, your YAML structure needs to include some required fields and values. Depending on your application requirements or application management requirements, you might need to include other optional fields and values.

The following YAML structure shows the required fields for an application and some of the common optional fields.

```

apiVersion: app.k8s.io/v1beta1
kind: Application
metadata:
  name:
  namespace:
spec:
  selector:
    matchLabels:
      label_name: label_value

```

1.4.8.2. Application YAML table

Field	Value	Description
apiVersion	app.k8s.io/v1beta1	Required
kind	Application	Required
metadata		
	name: The name for identifying the application resource.	Required
	namespace: The namespace resource to use for the application.	
spec		
selector.matchLabels	key:value pair that are a Kubernetes label and value found on the subscription or subscriptions this application will be associated with. The label allows the application resource to find the related subscriptions by performing a label name and value match.	Required

The spec for defining these applications is based on the Application metadata descriptor custom resource definition that is provided by the Kubernetes Special Interest Group (SIG). Only the values shown in the table are required.

You can use this definition to help you compose your own application YAML content. For more information about this definition, see [Kubernetes SIG Application CRD community specification](#).

1.4.8.3. Application file samples

The definition structure for an application can resemble the following example YAML content:

```
apiVersion: app.k8s.io/v1beta1
kind: Application
metadata:
  name: my-application
  namespace: my-namespace
spec:
  selector:
    matchLabels:
      my-label: my-label-value
```