Red Hat Advanced Cluster Management for Kubernetes 2.6

Applications

Read more to learn how to create applications by using Git repositories, Helm repositories, and object storage repositories.

Last Updated: 2023-10-31

Red Hat Advanced Cluster Management for Kubernetes 2.6 Applications

Read more to learn how to create applications by using Git repositories, Helm repositories, and object storage repositories.

Legal Notice

Abstract

Read more to learn how to create applications by using Git repositories, Helm repositories, and object storage repositories.

Table of Contents

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.

Important: An application name cannot exceed 37 characters.

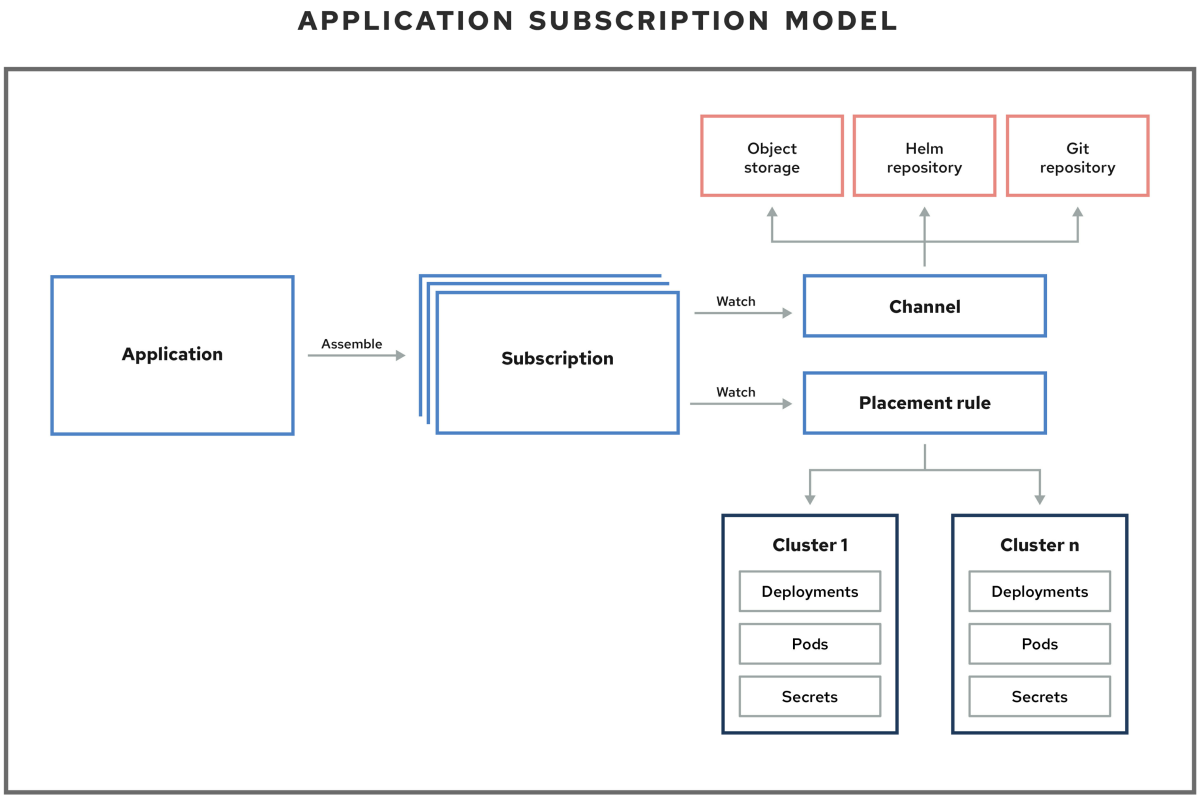
See the following topics:

- Application model and definitions
- Application console
- Subscription reports
- 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
- Granting subscription admin privilege
- Creating an allow and deny list as subscription administrator
- Adding reconcile options
- Configuring application channel and subscription for a secure Git connection
- Setting up Ansible Tower tasks
- Configuring GitOps on managed clusters
- Scheduling a deployment
- Configuring package overrides
- 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:



View the following application resource sections:

- Applications
- Subscriptions
- ApplicationSet
- Application documentation

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 specification sections. When you need to create or update an application component resource, you need to create or edit the appropriate section to include the labels for defining your resource.

You can also work with *Discovered* applications, which are applications that are discovered by the OpenShift Container Platform GitOps or an Argo CD operator that is installed in your clusters. Applications that share the same repository are grouped together in this view.

1.1.2. Subscriptions

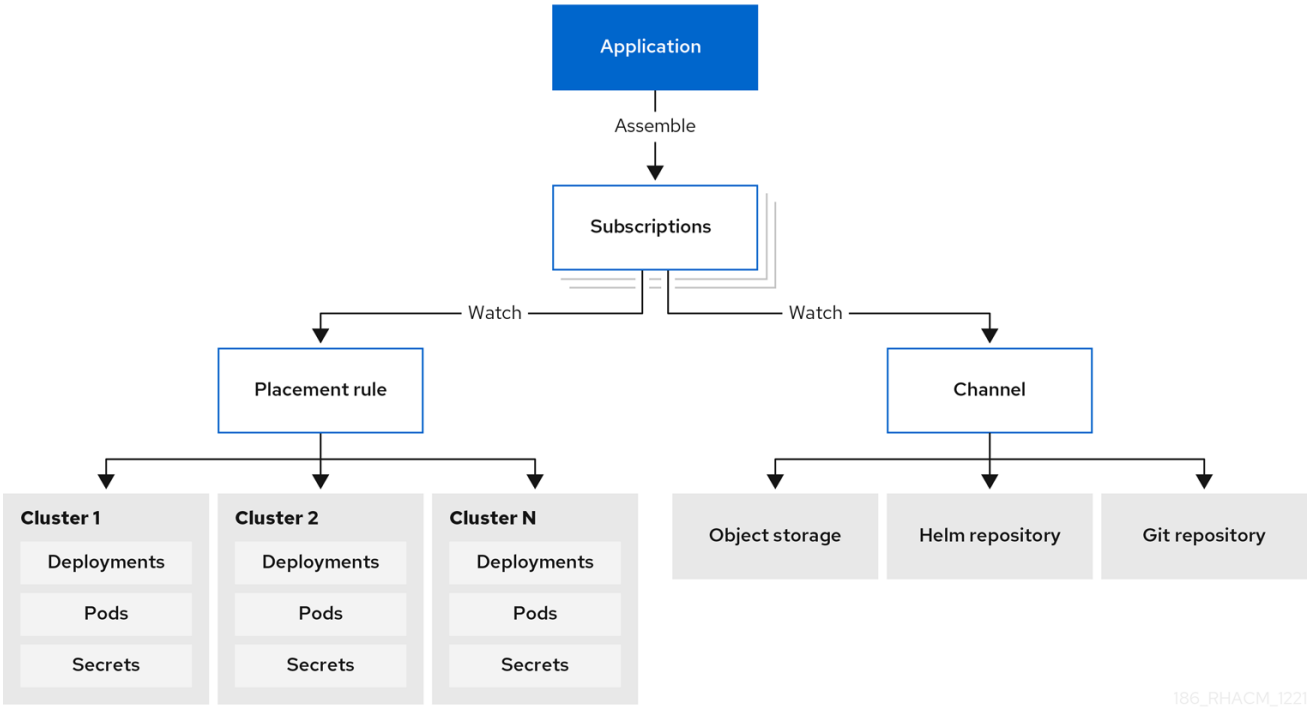
Subscriptions (**subscription.apps.open-cluster-management.io**) allow clusters to subscribe to a source repository (channel) that can be the following types: Git repository, Helm release registry, or Object storage repository.

Note: Self-managing the hub cluster is not recommended because the resources might impact the hub cluster.

Subscriptions can deploy application resources locally to the hub cluster if the hub cluster is self-managed. You can then view the **local-cluster** (the self-managed hub cluster) subscription in the topology. Resource requirements might adversely impact hub cluster performance.

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.

See the following subscription architecture image:



1.1.2.1. 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.

1.1.2.1.1. Supported Git repository servers

- GitHub
- GitLab
- Bitbucket
- Gogs

1.1.2.2. 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 rules are also used for governance and risk policies. For more information on how, see Governance.

1.1.3. ApplicationSet

ApplicationSet is a sub-project of Argo CD that is supported by the GitOps Operator. **ApplicationSet** adds multicluster support for Argo CD applications. You can create an application set from the Red Hat Advanced Cluster Management console.

Note: For more details on the prerequisites for deploying **ApplicationSet**, see Registering managed clusters to GitOps.

OpenShift Container Platform GitOps uses Argo CD to maintain cluster resources. Argo CD is an open-source declarative tool for the continuous integration and continuous deployment (CI/CD) of applications. OpenShift Container Platform GitOps implements Argo CD as a controller (OpenShift Container Platform GitOps Operator) so that it continuously monitors application definitions and configurations defined in a Git repository. Then, Argo CD compares the specified state of these configurations with their live state on the cluster.

The **ApplicationSet** controller is installed on the cluster through a GitOps operator instance and supplements it by adding additional features in support of cluster-administrator-focused scenarios. The **ApplicationSet** controller provides the following function:

- The ability to use a single Kubernetes manifest to target multiple Kubernetes clusters with the GitOps operator.
- The ability to use a single Kubernetes manifest to deploy multiple applications from one or multiple Git repositories with the GitOps operator.
- Improved support for monorepo, which is in the context of Argo CD, multiple Argo CD Application resources that are defined within a single Git repository.

- Within multitenant clusters, improved ability of individual cluster tenants to deploy applications using Argo CD without needing to involve privileged cluster administrators in enabling the destination clusters/namespaces.

The **ApplicationSet** operator leverages the cluster decision generator to interface Kubernetes custom resources that use custom resource-specific logic to decide which managed clusters to deploy to. A cluster decision resource generates a list of managed clusters, which are then rendered into the template fields of the **ApplicationSet** resource. This is done using duck-typing, which does not require knowledge of the full shape of the referenced Kubernetes resource.

See the following example of a **generators.clusterDecisionResource** value within an **ApplicationSet**:

```
apiVersion: argoproj.io/v1alpha1
kind: ApplicationSet
metadata:
  name: sample-application-set
  namespace: sample-gitops-namespace
spec:
  generators:
    - clusterDecisionResource:
        configMapRef: acm-placement
        labelSelector:
          matchLabels:
            cluster.open-cluster-management.io/placement: sample-application-placement
        requeueAfterSeconds: 180
  template:
    metadata:
      name: sample-application-{{name}}
    spec:
      project: default
      source:
        repoURL: https://github.com/sampleapp/apprepo.git
        targetRevision: main
        path: sample-application
      destination:
        namespace: sample-application
        server: "{{server}}"
      syncPolicy:
        syncOptions:
          - CreateNamespace=true
          - PruneLast=true
          - Replace=true
          - ApplyOutOfSyncOnly=true
          - Validate=false
        automated:
          prune: true
          allowEmpty: true
          selfHeal: true
```

See the following **Placement**:

```
apiVersion: cluster.open-cluster-management.io/v1beta1
kind: Placement
metadata:
  name: sample-application-placement
  namespace: sample-gitops-namespace
```

```
spec:
  clusterSets:
    - sampleclusterset
```

If you would like to learn more about **ApplicationSets**, see Cluster Decision Resource Generator.

1.1.4. Application documentation

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 some of the console capability in the following list and see the console for guided information about terms, actions, and how to read the Topology:

Important: Available 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. From the *Actions* menu, you can search, edit, or delete. Ensure you select **YAML:On** to view and edit the YAML as you update the fields.
- From the main *Overview* tab, you can click an application name to view details and application resources, including resource repositories, subscriptions, placements, placement rules, and deployed resources such as any optional predeployment and postdeployment hooks that are

using Ansible Tower tasks (for Git repositories). You can also create an application from the overview.

- Create and view applications, such as *ApplicationSet*, *Subscription*, *OpenShift*, *Flux*, and *Argo CD* types. An **ApplicationSet** represents Argo applications that are generated from the controller.
  - For an ArgoCD **ApplicationSet** to be created, you need to enable **Automatically sync when cluster state changes** from the **Sync policy**.
  - For Flux with the **kustomization** controller, find Kubernetes resources with the **kustomize.toolkit.fluxcd.io/name=<app_name>** label.
  - For Flux with the **helm** controller, find Kubernetes resources with the **helm.toolkit.fluxcd.io/name=<app_name>** label.
- **Note:** You need GitOps cluster resources and the GitOps operator installed to create an **ApplicationSet**. Without these prerequisites, you will see no **Argo server** options in the console to create an **ApplicationSet**.
- From the main *Overview*, when you click on an application name in the table to view a single application overview, you can see the following information:
- Cluster details, such as resource status
- Resource topology
- Subscription details
- Access to the Editor tab to edit
- Click the *Topology* tab for visual representation of all the applications and resources in your project. For Helm subscriptions, see Configuring package overrides to define the appropriate **packageName** and the **packageAlias** to get an accurate topology display.
- 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, placement rules, and channels. If you have access, you can also click multiple **Actions**, such as Edit, Search, and Delete.
- View a successful Ansible Tower deployment if you are using Ansible tasks as prehook or posthook for the deployed application.
- Click **Launch resource in Search** to search for related resources.
- Use *Search* to find 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	**Placement**

Application resource	Kind (search parameter)
Placement rule	**PlacementRule**
Application	**Application**

You can also search by other fields, including name, namespace, cluster, label, and more. For more information about using search, see Search in the console.

1.3. SUBSCRIPTION REPORTS

Subscription reports are collections of application statuses from all the managed clusters in your fleet. Specifically, the parent application resource can hold reports from a scalable amount of managed clusters.

Detailed application status is available on the managed clusters, while the **subscriptionReports** on the hub cluster are lightweight and more scalable. See the following three types of subsription status reports:

- Package-level **SubscriptionStatus**: This is the application package status on the managed cluster with detailed status for all the resources that are deployed by the application in the **appsub** namespace.
- Cluster-level **SubscriptionReport**: This is the overall status report on all the applications that are deployed to a particular cluster.
- Application-level **SubscriptionReport**: This is the overall status report on all the managed clusters to which a particular application is deployed.
  - SubscriptionStatus package-level
  - SubscriptionReport cluster-level
  - SubscriptionReport application-level
  - managedClusterView
  - CLI application-level status
  - CLI Last Update Time

1.3.1. SubscriptionStatus package-level

The package-level managed cluster status is located in **<namespace:<your-appsub-namespace>** on the managed cluster and contains detailed status for all the resources that are deployed by the application. For every **appsub** that is deployed to a managed cluster, there is a **SubscriptionStatus** CR created in the **appsub** namespace on the managed cluster. Every resource is reported with detailed errors if errors exist.

See the following **SubscriptionStatus** sample YAML file:

```
apiVersion: apps.open-cluster-management.io/v1alpha1
kind: SubscriptionStatus
metadata:
  labels:
    apps.open-cluster-management.io/cluster: <your-managed-cluster>
    apps.open-cluster-management.io/hosting-subscription: <your-appsub-namespace>.<your-
appsub-name>
  name: <your-appsub-name>
  namespace: <your-appsub-namespace>
statuses:
  packages:
  - apiVersion: v1
    kind: Service
    lastUpdateTime: "2021-09-13T20:12:34Z"
    Message: <detailed error. visible only if the package fails>
    name: frontend
    namespace: test-ns-2
    phase: Deployed
  - apiVersion: apps/v1
    kind: Deployment
    lastUpdateTime: "2021-09-13T20:12:34Z"
    name: frontend
    namespace: test-ns-2
    phase: Deployed
  - apiVersion: v1
    kind: Service
    lastUpdateTime: "2021-09-13T20:12:34Z"
    name: redis-master
    namespace: test-ns-2
    phase: Deployed
  - apiVersion: apps/v1
    kind: Deployment
    lastUpdateTime: "2021-09-13T20:12:34Z"
    name: redis-master
    namespace: test-ns-2
    phase: Deployed
  - apiVersion: v1
    kind: Service
    lastUpdateTime: "2021-09-13T20:12:34Z"
    name: redis-slave
    namespace: test-ns-2
    phase: Deployed
  - apiVersion: apps/v1
    kind: Deployment
    lastUpdateTime: "2021-09-13T20:12:34Z"
    name: redis-slave
    namespace: test-ns-2
    phase: Deployed
```

1.3.2. SubscriptionReport cluster-level

The cluster-level status is located in **<namespace:<your-managed-cluster-1>** on the the hub cluster and only contains the overall status on each application on that managed cluster. The **subscriptionReport** in each cluster namespace on the hub cluster reports one of the following statuses:

- **Deployed**
- **Failed**
- **propagationFailed**

See the following **SubscriptionStatus** sample YAML file:

```
apiVersion: apps.open-cluster-management.io/v1alpha1
kind: subscriptionReport
metadata:
  labels:
    apps.open-cluster-management.io/cluster: "true"
  name: <your-managed-cluster-1>
  namespace: <your-managed-cluster-1>
reportType: Cluster
results:
- result: deployed
  source: appsub-1-ns/appsub-1                  // appsub 1 to <your-managed-cluster-1>
  timestamp:
    nanos: 0
    seconds: 1634137362
- result: failed
  source: appsub-2-ns/appsub-2                  // appsub 2 to <your-managed-cluster-1>
  timestamp:
    nanos: 0
    seconds: 1634137362
- result: propagationFailed
  source: appsub-3-ns/appsub-3                  // appsub 3 to <your-managed-cluster-1>
  timestamp:
    nanos: 0
    seconds: 1634137362
```

1.3.3. SubscriptionReport application-level

One application-level **subscriptionReport** for each application is located in **<namespace:<your-appsub-namespace>** in **appsub** namespace on the hub cluster and contains the following information:

- The overall status of the application for each managed cluster
- A list of all resources for the application
- A report summary with the total number of total clusters
- A report summary with the total number of clusters where the application is in the status: **deployed**, **failed**, **propagationFailed**, and **inProgress**.

Note: The **inProcess** status is the total minus **deployed**, minus **failed `, and minus `propagationFailed**.

See the following **SubscriptionStatus** sample YAML file:

```
apiVersion: apps.open-cluster-management.io/v1alpha1
kind: subscriptionReport
metadata:
```

```
  labels:
    apps.open-cluster-management.io/hosting-subscription: <your-appsub-namespace>.<your-
appsub-name>
  name: <your-appsub-name>
  namespace: <your-appsub-namespace>
reportType: Application
resources:
- apiVersion: v1
  kind: Service
  name: redis-master2
  namespace: playback-ns-2
- apiVersion: apps/v1
  kind: Deployment
  name: redis-master2
  namespace: playback-ns-2
- apiVersion: v1
  kind: Service
  name: redis-slave2
  namespace: playback-ns-2
- apiVersion: apps/v1
  kind: Deployment
  name: redis-slave2
  namespace: playback-ns-2
- apiVersion: v1
  kind: Service
  name: frontend2
  namespace: playback-ns-2
- apiVersion: apps/v1
  kind: Deployment
  name: frontend2
  namespace: playback-ns-2
results:
- result: deployed
  source: cluster-1                    //cluster 1 status
  timestamp:
    nanos: 0
    seconds: 0
- result: failed
  source: cluster-3                    //cluster 2 status
  timestamp:
    nanos: 0
    seconds: 0
- result: propagationFailed
  source: cluster-4                    //cluster 3 status
  timestamp:
    nanos: 0
    seconds: 0
summary:
  deployed: 8
  failed: 1
  inProgress: 0
  propagationFailed: 1
  clusters: 10
```

1.3.4. ManagedClusterView

A **ManagedClusterView** CR is reported on the first **failed** cluster. If an application is deployed on multiple clusters with resource deployment failures, only one **managedClusterView** CR is created for the first failed cluster namespace on the hub cluster. The **managedClusterView** CR retrieves the detailed subscription status from the failed cluster so that the application owner does not need to access the failed remote cluster.

See the following command that you can run to get the status:

```
% oc get managedclusterview -n <failing-clusternamespace> "<app-name>-<app name>"
```

1.3.5. CLI application-level status

If you cannot access the managed clusters to get a subscription status, you can use the CLI. The cluster-level or the application-level subscription report provides the overall status, but not the the detailed error messages for an application.

1. Download the CLI from multicloud-operators-subscription.
2. Run the following command to create a **managedClusterView** resource to see the managed cluster application **SubscriptionStatus** so that you can identify the error:

```
% getAppSubStatus.sh -c <your-managed-cluster> -s <your-appsub-namespace> -n <your-appsub-name>
```

1.3.6. CLI Last Update Time

You can also get the Last Update Time of an AppSub on a given managed cluster when it is not practical to log in to each managed cluster to retrieve this information. Thus, an utility script was created to simplify the retrieval of the Last Update Time of an AppSub on a managed cluster. This script is designed to run on the Hub cluster. It creates a managedClusterView resource to get the AppSub from the managed cluster, and parses the data to get the Last Update Time.

1. Download the CLI from multicloud-operators-subscription.
2. Run the following command to retriev the **Last Update Time** of an **AppSub** on a managed cluster. This script is designed to run on the hub cluster. It creates a **managedClusterView** resource to get the AppSub from the managed cluster, and parses the data to get the Last Update Time:

```
% getLastUpdateTime.sh -c <your-managed-cluster> -s <your-appsub-namespace> -n <your-appsub-name>
```

1.4. 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

1.4.1. 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 **Applications** to see listed applications and to create new applications.
2. **Optional:** After you choose the kind of application you want to create, you can select **YAML: On** to view the YAML in the console as you create and edit your application. See the YAML samples later in the topic.
3. Choose **Git** from the list of repositories that you can use and enter the values in the correct fields. Follow the guidance in the console and see the YAML editor change values based on your input.
   Notes:
   - 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.
   - Notice the reconcile option. The **merge** option is the default selection, which means that new fields are added and existing fields are updated in the resource. You can choose to **replace**. With the **replace** option, the existing resource is replaced with the Git source. When the subscription reconcile rate is set to **low**, it can take up to one hour for the subscribed application resources to reconcile. On the card on the single application view, click **Sync** to reconcile manually. If set to **off**, it never reconciles.
4. Set any optional pre-deployment and post-deployment tasks. 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.
5. You can click **Add credential** if you need to add a credential using the console. Follow the directions in the console. See more information at Managing credentials overview.
6. Click **Create**.
7. You are redirected to the *Overview* page where you can view the details and topology.

1.4.1.1. GitOps pattern

Learn best practices for organizing a Git repository to manage clusters.

1.4.1.1.1. GitOps example directory

Folders in this example are defined and named, with each folder containing applications or configurations that are run on managed clusters:

- Root folder **managed-subscriptions**: Contains subscriptions that target the **common-managed** folder.
- Subfolder **apps/**: Used to subscribe applications in the **common-managed** folder with placement to **managed-clusters**.
- Subfolder **config/**: Used to subscribe configurations in the **common-managed** folder with placement to **managed-clusters**.
- Subfolder **policies/**: Used to apply policies with placement to **managed-clusters**.
- Folder **root-subscription/**: The initial subscription for the hub cluster that subscribes the **managed-subscriptions** folder.

See the example of a directory:

```
common-managed/
   apps/
     app-name-0/
     app-name-1/
   config/
     config001/
     config002/

managed-subscriptions
   apps/
   config/
   policies/

root-subscription/
```

1.4.1.1.2. GitOps flow

Your directory structure is created for the following subscription flow: **root-subscription** > **managed-subscriptions** > **common-managed**.

1. A single subscription in **root-subscription/** is applied from the CLI terminal to the hub cluster.
2. Subscriptions and policies are downloaded and applied to the hub cluster from the **managed-subscription** folder.
   - The subscriptions and policies in the **managed-subscription** folder then perform work on the managed clusters based on the placement.
   - Placement determines which **managed-clusters** each subscription or policy affects.
   - The subscriptions or policies define what is on the clusters that match their placement.
3. Subscriptions apply content from the **common-managed** folder to **managed-clusters** that match the placement rules. This also applies common applications and configurations to all **managed-clusters** that match the placement rules.

1.4.1.1.3. More examples

- For an example of **root-subscription**/, see application-subscribe-all.
- For examples of subscriptions that point to other folders in the same repository, see subscribe-all.
- See an example of the **common-managed** folder with application artifacts in the nginx-apps repository.
- See policy examples in Policy collection.

1.4.1.2. Keeping deployed resources after deleting subscription with Git

When creating subscriptions using a Git repository, you can add a **do-not-delete** annotation to keep specific deployed resources after you delete the subscription. The **do-not-delete** annotation only works with top-level deployment resources. To add the **do-not-delete** annotation, complete the following steps:

1. Create a subscription that deploys at least one resource.
2. Add the following annotation to the resource or resources that you want to keep, even after you delete the subscription:
   apps.open-cluster-management.io/do-not-delete: 'true'

   See the following example:

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  annotations:
    apps.open-cluster-management.io/do-not-delete: 'true'
    apps.open-cluster-management.io/hosting-subscription: sub-ns/subscription-example
    apps.open-cluster-management.io/reconcile-option: merge
    pv.kubernetes.io/bind-completed: "yes"
```

After deleting the subscription, the resources with the **do-not-delete** annotation still exist, while other resources are deleted.

1.4.2. 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 **Applications** to see listed applications and to create new applications.
2. **Optional:** After you choose the kind of application you want to create, you can select **YAML: On** to view the YAML in the console as you create and edit your application. See the YAML samples later in the topic.
3. Choose **Helm** from the list of repositories that you can use and enter the values in the correct fields. Follow the guidance in the console and see the YAML editor change values based on your input.

4. Click **Create**.

5. You are redirected to the *Overview* page where you can view the details and topology.

1.4.2.1. 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 APIs, use the APIs.

1.4.2.2. Keeping deployed resources after deleting subscription with Helm

Helm provides an annotation to keep specific deployed resources after you delete a subscription. See Tell Helm Not To Uninstall a Resource for more information.

Note: The annotation must be in the Helm chart.

1.4.3. Managing apps with Object storage repositories

When you deploy Kubernetes resources using an application, the resources are located in specific repositories. 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 **Applications** to see listed applications and to create new applications.

2. **Optional:** After you choose the kind of application you want to create, you can select **YAML: On** to view the YAML in the console as you create and edit your application. See the YAML samples later in the topic.

3. Choose **Object store** from the list of repositories that you can use and enter the values in the correct fields. Follow the guidance in the console and see the YAML editor change values based on your input.

4. Click **Create**.

5. You are redirected to the *Overview* page where you can view the details and topology.

1.4.3.1. 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://sample-ip:#####/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.

1.4.3.2. Creating your Amazon Web Services (AWS) S3 object storage bucket

You can set up subscriptions to subscribe resources that are defined in the Amazon Simple Storage Service (Amazon S3) object storage service. See the following procedure:

1. Log into the AWS console with your AWS account, user name, and password.

2. Navigate to **Amazon S3** > **Buckets** to the bucket home page.

3. Click **Create Bucket** to create your bucket.

4. Select the **AWS region**, which is essential for connecting your AWS S3 object bucket.

5. Create the bucket access token.

6. Navigate to your user name in the navigation bar, then from the drop-down menu, select **My Security Credentials**.

7. Navigate to *Access keys for CLI, SDK, & API access* in the *AWS IAM credentials* tab and click on **Create access key**.

8. Save your *Access key ID* , *Secret access key*.

9. Upload your object YAML files to the bucket.

1.4.3.3. Subscribing to the object in the AWS bucket

1. Create an object bucket type channel with a secret to specify the *AccessKeyID*, *SecretAccessKey*, and *Region* for connecting the AWS bucket. The three fields are created when the AWS bucket is created.

2. Add the URL. The URL identifies the channel in a AWS S3 bucket if the URL contains **s3:**// or **s3 and aws** keywords. For example, see all of the following bucket URLs have AWS s3 bucket identifiers:

   ```
   https://s3.console.aws.amazon.com/s3/buckets/sample-bucket-1
   s3://sample-bucket-1/
   https://sample-bucket-1.s3.amazonaws.com/
   ```

   Note: The AWS S3 object bucket URL is not necessary to connect the bucket with the AWS S3 API.

1.4.3.4. Sample AWS subscription

See the following complete AWS S3 object bucket channel sample YAML file:

```
apiVersion: apps.open-cluster-management.io/v1
kind: Channel
metadata:
  name: object-dev
  namespace: ch-object-dev
spec:
  type: ObjectBucket
  pathname: https://s3.console.aws.amazon.com/s3/buckets/sample-bucket-1
  secretRef:
    name: secret-dev
---
apiVersion: v1
kind: Secret
metadata:
  name: secret-dev
  namespace: ch-object-dev
stringData:
  AccessKeyID: <your AWS bucket access key id>
  SecretAccessKey: <your AWS bucket secret access key>
  Region: <your AWS  bucket region>
type: Opaque
```

You can continue to create other AWS subscription and placement rule objects, as you see in the following sample YAML with **kind: PlacementRule** and **kind: Subscription** added:

```
apiVersion: apps.open-cluster-management.io/v1
kind: PlacementRule
metadata:
  name: towhichcluster
  namespace: obj-sub-ns
```

```
spec:
  clusterSelector: {}
---
apiVersion: apps.open-cluster-management.io/v1
kind: Subscription
metadata:
  name: obj-sub
  namespace: obj-sub-ns
spec:
  channel: ch-object-dev/object-dev
  placement:
    placementRef:
      kind: PlacementRule
      name: towhichcluster
```

You can also subscribe to objects within a specific subfolder in the object bucket. Add the **subfolder** annotation to the subscription, which forces the object bucket subscription to only apply all the resources in the subfolder path.

See the annotation with **subfolder-1** as the **bucket-path**:

```
annotations:
  apps.open-cluster-management.io/bucket-path: <subfolder-1>
```

See the following complete sample for a subfolder:

```
apiVersion: apps.open-cluster-management.io/v1
kind: Subscription
metadata:
  annotations:
    apps.open-cluster-management.io/bucket-path: subfolder1
  name: obj-sub
  namespace: obj-sub-ns
  labels:
    name: obj-sub
spec:
  channel: ch-object-dev/object-dev
  placement:
    placementRef:
      kind: PlacementRule
      name: towhichcluster
```

1.4.3.5. Keeping deployed resources after deleting subscription with Object storage

When creating subscriptions using an Object storage repository, you can add a **do-not-delete** annotation to keep specific deployed resources after you delete the subscription. The **do-not-delete** annotation only works with top-level deployment resources. To add the **do-not-delete** annotation, complete the following steps:

1. Create a subscription that deploys at least one resource.

2. Add the following annotation to the resource or resources that you want to keep, even after you delete the subscription:
   apps.open-cluster-management.io/do-not-delete: 'true'

See the following example:

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  annotations:
    apps.open-cluster-management.io/do-not-delete: 'true'
    apps.open-cluster-management.io/hosting-subscription: sub-ns/subscription-example
    apps.open-cluster-management.io/reconcile-option: merge
    pv.kubernetes.io/bind-completed: "yes"
```

After deleting the subscription, the resources with the **do-not-delete** annotation still exist, while other resources are deleted.

1.5. APPLICATION ADVANCED CONFIGURATION

Within Red Hat Advanced Cluster Management for Kubernetes, applications are composed of multiple application resources. You can use channel, subscription, placements, 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 specification sections. When you need to create or update an application component resource, you need to create or edit the appropriate section to include the labels for defining your resource.

View the following application advanced configuration topics:

- Subscribing Git resources
- Granting subscription admin privilege
- Creating an allow and deny list as subscription administrator
- Adding reconcile options
- Configuring application channel and subscription for a secure Git connection
- Setting up Ansible Tower tasks
- Configuring Helm to watch namespace resources
- Configuring GitOps on managed clusters
- Configuring package overrides
- Channel samples overview
- Subscription samples overview
- Placement rule samples overview
- Application samples overview

1.5.1. Subscribing Git resources

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. A *subscription administrator* can change default behavior, as described in Granting subscription admin privilege.

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

- Creating application resources in Git
- Subscribing specific Git elements
- Application namespace example
- Resource overwrite example

1.5.1.1. Creating application resources in Git

You need to specify the full group and version for **apiVersion** in resource YAML when you subscribe. For example, if you subscribe to **apiVersion: v1**, the subscription controller fails to validate the subscription and you receive an error: **Resource /v1, Kind=ImageStream is not supported**.

If the **apiVersion** is changed to **image.openshift.io/v1**, as in the following sample, it passes the validation in the subscription controller and the resource is applied successfully.

```
apiVersion: `image.openshift.io/v1`
kind: ImageStream
metadata:
  name: default
  namespace: default
spec:
  lookupPolicy:
    local: true
  tags:
    - name: 'latest'
      from:
        kind: DockerImage
        name: 'quay.io/repository/open-cluster-management/multicluster-operators-subscription:community-latest'
```

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

1.5.1.2. Application namespace example

In this following examples, you are logged in as a subscription administrator.

1.5.1.2.1. Application to different namespaces

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.

1.5.1.2.2. Application to same namespace

As a subscription administrator, you might want to deploy all application resources into the same namespace.

You can deploy all application resources into the subscription namespace by Creating an allow and deny list as subscription administrator.

Add **apps.open-cluster-management.io/current-namespace-scoped: true** annotation to the subscription YAML. For example, when a subscription administrator creates the following subscription, all three ConfigMaps in the previous example are created in **subscription-ns** namespace.

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

```
  namespace: subscription-ns
  annotations:
    apps.open-cluster-management.io/git-path: sample-resources
    apps.open-cluster-management.io/reconcile-option: merge
    apps.open-cluster-management.io/current-namespace-scoped: "true"
spec:
  channel: channel-ns/somechannel
  placement:
    placementRef:
      name: dev-clusters
```

1.5.1.3. Resource overwrite example

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

Note: The resource overwrite option is not applicable to **helm** charts from the Git repository because the **helm** chart resources are managed by Helm.

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. See the change in the **data** specification:

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

1.5.1.3.1. Default merge option

See the following sample resource YAML file from a Git repository with the default **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.

1.5.1.3.2. mergeAndOwn option

With **mergeAndOwn**, entries from subscribed resource are either created or updated in the existing resource. Log in as a subscription administrator and create a subscription with **apps.open-cluster-management.io/reconcile-option: mergeAndOwn** 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: mergeAndOwn
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
  annotations:
    apps.open-cluster-management.io/hosting-subscription: sub-ns/subscription-example
```

```
data:
  name: user1
  age: 20
```

As previosly mentioned, when the **mergeAndOwn** option is used, entries from subscribed resource are either created or updated in the existing resource. No entry is removed from the existing resource. It also adds the **apps.open-cluster-management.io/hosting-subscription** annotation to indicate that the resource is now owned by the subscription. Deleting the subscription deletes the ConfigMap.

1.5.1.3.3. Replace option

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
```

1.5.1.4. Subscribing specific Git elements

You can subscribe to a specific Git branch, commit, or tag.

1.5.1.4.1. Subscribing to a specific branch

The subscription operator that is included in the **multicloud-operators-subscription** repository subscribes to the default branch of a Git repository. If you want to subscribe to a different branch, you need to specify the branch name annotation in the subscription.

The following example, the YAML file displays how to specify a different branch with **apps.open-cluster-management.io/git-branch: <branch1>**:

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

```
metadata:
  name: git-mongodb-subscription
  annotations:
    apps.open-cluster-management.io/git-path: stable/ibm-mongodb-dev
    apps.open-cluster-management.io/git-branch: <branch1>
```

1.5.1.4.2. Subscribing to a specific commit

The subscription operator that is included in the **multicloud-operators-subscription** repository subscribes to the latest commit of specified branch of a Git repository by default. If you want to subscribe to a specific commit, you need to specify the desired commit annotation with the commit hash in the subscription.

The following example, the YAML file displays how to specify a different commit with **apps.open-cluster-management.io/git-desired-commit: <full commit number>**:

```
apiVersion: apps.open-cluster-management.io/v1
kind: Subscription
metadata:
  name: git-mongodb-subscription
  annotations:
    apps.open-cluster-management.io/git-path: stable/ibm-mongodb-dev
    apps.open-cluster-management.io/git-desired-commit: <full commit number>
    apps.open-cluster-management.io/git-clone-depth: 100
```

The **git-clone-depth** annotation is optional and set to **20** by default, which means the subscription controller retrieves the previous 20 commit histories from the Git repository. If you specify a much older **git-desired-commit**, you need to specify **git-clone-depth** accordingly for the desired commit.

1.5.1.4.3. Subscribing to a specific tag

The subscription operator that is included in the **multicloud-operators-subscription** repository subscribes to the latest commit of specified branch of a Git repository by default. If you want to subscribe to a specific tag, you need to specify the tag annotation in the subscription.

The following example, the YAML file displays how to specify a different tag with **apps.open-cluster-management.io/git-tag: <v1.0>**:

```
apiVersion: apps.open-cluster-management.io/v1
kind: Subscription
metadata:
  name: git-mongodb-subscription
  annotations:
    apps.open-cluster-management.io/git-path: stable/ibm-mongodb-dev
    apps.open-cluster-management.io/git-tag: <v1.0>
    apps.open-cluster-management.io/git-clone-depth: 100
```

Note: If both Git desired commit and tag annotations are specified, the tag is ignored.

The **git-clone-depth** annotation is optional and set to **20** by default, which means the subscription controller retrieves the previous **20** commit history from the Git repository. If you specify much older **git-tag**, you need to specify **git-clone-depth** accordingly for the desired commit of the tag.

1.5.2. Granting subscription administrator privilege

Learn how to grant subscription administrator access. A *subscription* administrator can change default behavior. Learn more in the following process:

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. You can also prepare groups or service accounts.
   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. If **open-cluster-management:subscription-admin** ClusterRoleBinding does not exist, you need to create it. See the following example:

```
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRoleBinding
metadata:
  name: open-cluster-management:subscription-admin
roleRef:
  apiGroup: rbac.authorization.k8s.io
  kind: ClusterRole
  name: open-cluster-management:subscription-admin
```

5. 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
- kind: ServiceAccount
  name: my-service-account
  namespace: my-service-account-namespace
# Service Account can be used as a user subject as well
- apiGroup: rbac.authorization.k8s.io
  kind: User
  name: 'system:serviceaccount:my-service-account-namespace:my-service-account'
```

1.5.3. Creating an allow and deny list as subscription administrator

As a subscription administrator, you can create an application from a Git repository application subscription that contains an **allow** list to allow deployment of only specified Kubernetes **kind**

resources. You can also create a **deny** list in the application subscription to deny deployment of specific Kubernetes **kind** resources.

By default, **policy.open-cluster-management.io/v1** resources are not deployed by an application subscription. To avoid this default behavior, application subscription needs deployed by a subscription administrator.

See the following example of **allow** and **deny** specifications:

```
apiVersion: apps.open-cluster-management.io/v1
kind: Subscription
metadata:
  annotations:
    apps.open-cluster-management.io/github-path: sub2
  name: demo-subscription
  namespace: demo-ns
spec:
  channel: demo-ns/somechannel
  allow:
  - apiVersion: policy.open-cluster-management.io/v1
    kinds:
    - Policy
  - apiVersion: v1
    kinds:
    - Deployment
  deny:
  - apiVersion: v1
    kinds:
    - Service
    - ConfigMap
  placement:
    local: true
```

The following application subscription YAML specifies that when the application is deployed from the **myapplication** directory from the source repository, it deploys only **v1**/**Deployment** resources, even if there are other resources in the source repository:

```
apiVersion: apps.open-cluster-management.io/v1
kind: Subscription
metadata:
  annotations:
    apps.open-cluster-management.io/github-path: myapplication
  name: demo-subscription
  namespace: demo-ns
spec:
  channel: demo-ns/somechannel
  deny:
  - apiVersion: v1
     kinds:
      - Service
      - ConfigMap
  placement:
    placementRef:
      name: demo-placement
      kind: PlacementRule
```

This example application subscription YAML specifies deployments of all valid resources except **v1**/**Service** and **v1**/**ConfigMap** resources. Instead of listing individual resource kinds within an API group, you can add **"*"** to allow or deny all resource kinds in the API Group.

1.5.4. Adding reconcile options

You can use the **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 is merged on the target cluster while other resources are replaced.

1.5.4.1. Reconcile frequency Git channel

You can select reconcile frequency options: **high**, **medium**, **low**, and **off** in channel configuration to avoid unnecessary resource reconciliations and therefore prevent overload on subscription operator.

Required access: Administrator and cluster administrator

See the following definitions of the **settings:attribute:<value>**:

- **Off**: The deployed resources are not automatically reconciled. A change in the **Subscription** custom resource initiates a reconciliation. You can add or update a label or annotation.
- **Low**: The deployed resources are automatically reconciled every hour, even if there is no change in the source Git repository.
- **Medium**: This is the default setting. The subscription operator compares the currently deployed commit ID to the latest commit ID of the source repository every 3 minutes, and applies changes to target clusters. Every 15 minutes, all resources are reapplied from the source Git repository to the target clusters, even if there is no change in the repository.
- **High**: The deployed resources are automatically reconciled every two minutes, even if there is no change in the source Git repository.

You can set this by using the **apps.open-cluster-management.io/reconcile-rate** annotation in the channel custom resource that is referenced by subscription.

See the following **name: git-channel** example:

```
apiVersion: apps.open-cluster-management.io/v1
kind: Channel
metadata:
  name: git-channel
  namespace: sample
  annotations:
    apps.open-cluster-management.io/reconcile-rate: <value from the list>
spec:
  type: GitHub
  pathname: <Git URL>
---
apiVersion: apps.open-cluster-management.io/v1
kind: Subscription
metadata:
```

```
  name: git-subscription
  annotations:
    apps.open-cluster-management.io/git-path: <application1>
    apps.open-cluster-management.io/git-branch: <branch1>
spec:
  channel: sample/git-channel
  placement:
    local: true
```

In the previous example, all subscriptions that use **sample/git-channel** are assigned **low** reconciliation frequency.

a. When the subscription reconcile rate is set to **low**, it can take up to one hour for the subscribed application resources to reconcile. On the card on the single application view, click **Sync** to reconcile manually. If set to **off**, it never reconciles.

Regardless of the **reconcile-rate** setting in the channel, a subscription can turn the auto-reconciliation **off** by specifying **apps.open-cluster-management.io/reconcile-rate: off** annotation in the **Subscription** custom resource.

See the following **git-channel** example:

```
apiVersion: apps.open-cluster-management.io/v1
kind: Channel
metadata:
  name: git-channel
  namespace: sample
  annotations:
    apps.open-cluster-management.io/reconcile-rate: high
spec:
  type: GitHub
  pathname: <Git URL>
---
apiVersion: apps.open-cluster-management.io/v1
kind: Subscription
metadata:
  name: git-subscription
  annotations:
    apps.open-cluster-management.io/git-path: application1
    apps.open-cluster-management.io/git-branch: branch1
    apps.open-cluster-management.io/reconcile-rate: "off"
spec:
  channel: sample/git-channel
  placement:
    local: true
```

See that the resources deployed by **git-subscription** are never automatically reconciled even if the **reconcile-rate** is set to **high** in the channel.

1.5.4.2. Reconcile frequency Helm channel

Every 15 minutes, the subscription operator compares currently deployed hash of your Helm chart to the hash from the source repository. Changes are applied to target clusters. The frequency of resource reconciliation impacts the performance of other application deployments and updates.

For example, if there are hundreds of application subscriptions and you want to reconcile all subscriptions more frequently, the response time of reconciliation is slower.

Depending on the Kubernetes resources of the application, appropriate reconciliation frequency can improve performance.

- **Off**: The deployed resources are not automatically reconciled. A change in the Subscription custom resource initiates a reconciliation. You can add or update a label or annotation.
- **Low**: The subscription operator compares currently deployed hash to the hash of the source repository every hour and apply changes to target clusters when there is change.
- **Medium**: This is the default setting. The subscription operator compares currently deployed hash to the hash of the source repository every 15 minutes and apply changes to target clusters when there is change.
- **High**: The subscription operator compares currently deployed hash to the hash of the source repository every 2 minutes and apply changes to target clusters when there is change.

You can set this using **apps.open-cluster-management.io/reconcile-rate** annotation in the **Channel** custom resource that is referenced by subscription. See the following **helm-channel** example:

See the following **helm-channel** example:

```
apiVersion: apps.open-cluster-management.io/v1
kind: Channel
metadata:
  name: helm-channel
  namespace: sample
  annotations:
    apps.open-cluster-management.io/reconcile-rate: low
spec:
  type: HelmRepo
  pathname: <Helm repo URL>
---
apiVersion: apps.open-cluster-management.io/v1
kind: Subscription
metadata:
  name: helm-subscription
spec:
  channel: sample/helm-channel
  name: nginx-ingress
  packageOverrides:
  - packageName: nginx-ingress
    packageAlias: nginx-ingress-simple
    packageOverrides:
    - path: spec
      value:
        defaultBackend:
          replicaCount: 3
  placement:
    local: true
```

In this example, all subscriptions that uses **sample/helm-channel** are assigned a **low** reconciliation frequency.

Regardless of the reconcile-rate setting in the channel, a subscription can turn the auto-reconciliation **off** by specifying **apps.open-cluster-management.io/reconcile-rate: off** annotation in the **Subscription** custom resource, as displayed in the following example:

```
apiVersion: apps.open-cluster-management.io/v1
kind: Channel
metadata:
  name: helm-channel
  namespace: sample
  annotations:
    apps.open-cluster-management.io/reconcile-rate: high
spec:
  type: HelmRepo
  pathname: <Helm repo URL>
---
apiVersion: apps.open-cluster-management.io/v1
kind: Subscription
metadata:
  name: helm-subscription
  annotations:
    apps.open-cluster-management.io/reconcile-rate: "off"
spec:
  channel: sample/helm-channel
  name: nginx-ingress
  packageOverrides:
  - packageName: nginx-ingress
    packageAlias: nginx-ingress-simple
    packageOverrides:
    - path: spec
      value:
        defaultBackend:
          replicaCount: 3
  placement:
    local: true
```

In this example, the resources deployed by **helm-subscription** are never automatically reconciled, even if the **reconcile-rate** is set to **high** in the channel.

1.5.5. Configuring application channel and subscription for a secure Git connection

Git channels and subscriptions connect to the specified Git repository through HTTPS or SSH. The following application channel configurations can be used for secure Git connections:

- Connecting to a private repo with user and access token
- Making an insecure HTTPS connection to a Git server
- Using custom CA certificates for a secure HTTPS connection
- Making an SSH connection to a Git server
- Updating certificates and SSH keys

1.5.5.1. Connecting to a private repo with user and access token

You can connect to a Git server using channel and subscription. See the following procedures for connecting to a private repository with a user and access token:

1. Create a secret in the same namespace as the channel. Set the **user** field to a Git user ID and the **accessToken** field to a Git personal access token. The values should be base64 encoded. See the following sample with user and accessToken populated:

```
apiVersion: v1
kind: Secret
metadata:
  name: my-git-secret
  namespace: channel-ns
data:
  user: dXNlcgo=
  accessToken: cGFzc3dvcmQK
```

2. Configure the channel with a secret. See the following sample with the **secretRef** populated:

```
apiVersion: apps.open-cluster-management.io/v1
kind: Channel
metadata:
  name: sample-channel
  namespace: channel-ns
spec:
    type: Git
    pathname: <Git HTTPS URL>
    secretRef:
      name: my-git-secret
```

1.5.5.2. Making an insecure HTTPS connection to a Git server

You can use the following connection method in a development environment to connect to a privately-hosted Git server with SSL certificates that are signed by custom or self-signed certificate authority. However, this procedure is not recommended for production:

Specify **insecureSkipVerify: true** in the channel specification. Otherwise, the connection to the Git server fails with an error similar to the following:

```
x509: certificate is valid for localhost.com, not localhost
```

See the following sample with the channel specification addition for this method:

```
apiVersion: apps.open-cluster-management.io/v1
ind: Channel
metadata:
labels:
  name: sample-channel
  namespace: sample
spec:
  type: GitHub
  pathname: <Git HTTPS URL>
  insecureSkipVerify: true
```

1.5.5.3. Using custom CA certificates for a secure HTTPS connection

You can use this connection method to securely connect to a privately-hosted Git server with SSL certificates that are signed by custom or self-signed certificate authority.

1. Create a ConfigMap to contain the Git server root and intermediate CA certificates in PEM format. The ConfigMap must be in the same namespace as the channel CR. The field name must be **caCerts** and use |. From the following sample, notice that **caCerts** can contain multiple certificates, such as root and intermediate CAs:

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: git-ca
  namespace: channel-ns
data:
  caCerts: |
    # Git server root CA

    -----BEGIN CERTIFICATE-----
    MIIF5DCCA8wCCQDInYMol7LSDTANBgkqhkiG9w0BAQsFADCBszELMAkGA1UEBhMC

Q0ExCzAJBgNVBAgMAk9OMRAwDgYDVQQHDAdUb3JvbnRvMQ8wDQYDVQQKDAZSZW
RI

YXQxDDAKBgNVBAsMA0FDTTFFMEMGA1UEAww8Z29ncy1zdmMtZGVmYXVsdC5hcHBz
    LnJqdW5nLWh1YjEzLmRldjA2LnJlZC1jaGVzdGVyZmllbGQuY29tMR8wHQYJKoZI
    hvcNAQkBFhByb2tlakByZWRoYXQuY29tMB4XDTIwMTIwMzE4NTMxMloXDTIzMDky

MzE4NTMxMlowgbMxCzAJBgNVBAYTAkNBMQswCQYDVQQIDAJPTjEQMA4GA1UEBwwH


VG9yb250bzEPMA0GA1UECgwGUmVkSGF0MQwwCgYDVQQLDANBQ00xRTBDBgNVBA
MM
    PGdvZ3Mtc3ZjLWRlZmF1bHQuYXBwcy5yanVuZy1odWIxMy5kZXYwNi5yZWQtY2hl
    c3RlcmZpZWxkLmNvbTEfMB0GCSqGSIb3DQEJARYQcm9rZWpAcmVkaGF0LmNvbTCC
    AiIwDQYJKoZIhvcNAQEBBQADggIPADCCAgoCggIBAM3nPK4mOQzaDAo6S3ZJ0Ic3
    U9p/NLodnoTIC+cn0q8qNCAjf13zbGB3bfN9Zxl8Q5fv+wYwHrUOReCp6U/InyQy
    6OS3gj738F635inz1KdyhKtlWW2p9Ye9DUtx1IlfHkDVdXtynjHQbsFNIdRHcpQP
    upM5pwPC3BZXqvXChhlfAy2m4yu7vy0hO/oTzWIwNsoL5xt0Lw4mSyhlEip/t8lU
    xn2y8qhm7MiIUpXuwWhSYgCrEVqmTcB70Pc2YRZdSFolMN9Et70MjQN0TXjoktH8
    PyASJIKIRd+48yROIbUn8rj4aYYBsJuoSCjJNwujZPbqseqUr42+v+Qp2bBj1Sjw
    +SEZfHTvSv8AqX0T6eo6njr578+DgYlwsS1A1zcAdzp8qmDGqvJDzwcnQVFmvaoM
    gGHCdJihfy3vDhxuZRDse0V4Pz6tl6iklM+tHrJL/bdL0NdfJXNCqn2nKrM51fpw
    diNXs4Zn3QSStC2x2hKnK+Q1rwCSEg/lBawgxGUslTboFH77a+Kwu4Oug9ibtm5z
    ISs/JY4Kiy4C2XJOltOR2XZYkdKaX4x3ctbrGaD8Bj+QHiSAxaaSXIX+VbzkHF2N
    aD5ijFUopjQEKFrYh3O93DB/URIQ+wHVa6+Kvu3uqE0cg6pQsLpbFVQ/I8xHvt9L
    kYy6z6V/nj9ZYKQbq/kPAgMBAAEwDQYJKoZIhvcNAQELBQADggIBAKZuc+lewYAv
    jaaSeRDRoToTb/yN0Xsi69UfK0aBdvhCa7/0rPHcv8hmUBH3YgkZ+CSA5ygajtL4
    g2E8CwIO9ZjZ6l+pHCuqmNYoX1wdjaaDXlpwk8hGTSgy1LsOoYrC5ZysCi9Jilu9
    PQVGs/vehQRqLV9uZBigG6oZqdUqEimaLHrOcEAHB5RVcnFurz0qNbT+UySjsD63
    9yJdCeQbeKAR9SC4hG13EbM/RZh0lgFupkmGts7QYULzT+oA0cCJpPLQl6m6qGyE
    kh9aBB7FLykK1TeXVuANlNU4EMyJ/e+uhNkS9ubNJ3vuRuo+ECHsha058yi16JC9
    NkZqP+df4Hp85sd+xhrgYieq7QGX2KOXAjqAWo9htoBhOyW3mm783A7WcOiBMQv0
    2UGZxMsRjlP6UqB08LsV5ZBAefElR344sokJR1de/Sx2J9J/am7yOoqbtKpQotIA
    XSUkATuuQw4ctyZLDkUpzrDzgd2Bt+aawF6sD2YqycaGFwv2YD9t1YlD6F4Wh8Mc
    20Qu5EGrkQTCWZ9pOHNSa7YQdmJzwbxJC4hqBpBRAJFl2fAlqFtyum6/8ZN9nZ9K
    FSEKdlu+xeb6Y6xYt0mJJWF6mCRi4i7IL74EU/VNXwFmfP6IadliUOST3w5t92cB
```

```
    M26t73UCExXMXTCQvnp0ki84PeR1kRk4
    -----END CERTIFICATE-----

    # Git server intermediate CA 1

    -----BEGIN CERTIFICATE-----
    MIIF5DCCA8wCCQDInYMol7LSDTANBgkqhkiG9w0BAQsFADCBszELMAkGA1UEBhMC

Q0ExCzAJBgNVBAgMAk9OMRAwDgYDVQQHDAdUb3JvbnRvMQ8wDQYDVQQKDAZSZW
RI

YXQxDDAKBgNVBAsMA0FDTTFFMEMGA1UEAww8Z29ncy1zdmMtZGVmYXVsdC5hcHBz
    LnJqdW5nLWh1YjEzLmRldjA2LnJlZC1jaGVzdGVyZmllbGQuY29tMR8wHQYJKoZI
    hvcNAQkBFhByb2tlakByZWRoYXQuY29tMB4XDTIwMTIwMzE4NTMxMloXDTIzMDky

MzE4NTMxMlowgbMxCzAJBgNVBAYTAkNBMQswCQYDVQQIDAJPTjEQMA4GA1UEBwwH


VG9yb250bzEPMA0GA1UECgwGUmVkSGF0MQwwCgYDVQQLDANBQ00xRTBDBgNVBA
MM
    PGdvZ3Mtc3ZjLWRlZmF1bHQuYXBwcy5yanVuZy1odWIxMy5kZXYwNi5yZWQtY2hl
    c3RlcmZpZWxkLmNvbTEfMB0GCSqGSIb3DQEJARYQcm9rZWpAcmVkaGF0LmNvbTCC
    AiIwDQYJKoZIhvcNAQEBBQADggIPADCCAgoCggIBAM3nPK4mOQzaDAo6S3ZJ0Ic3
    U9p/NLodnoTIC+cn0q8qNCAjf13zbGB3bfN9Zxl8Q5fv+wYwHrUOReCp6U/InyQy
    6OS3gj738F635inz1KdyhKtlWW2p9Ye9DUtx1IlfHkDVdXtynjHQbsFNIdRHcpQP
    upM5pwPC3BZXqvXChhlfAy2m4yu7vy0hO/oTzWIwNsoL5xt0Lw4mSyhlEip/t8lU
    xn2y8qhm7MiIUpXuwWhSYgCrEVqmTcB70Pc2YRZdSFolMN9Et70MjQN0TXjoktH8
    PyASJIKIRd+48yROIbUn8rj4aYYBsJuoSCjJNwujZPbqseqUr42+v+Qp2bBj1Sjw
    +SEZfHTvSv8AqX0T6eo6njr578+DgYlwsS1A1zcAdzp8qmDGqvJDzwcnQVFmvaoM
    gGHCdJihfy3vDhxuZRDse0V4Pz6tl6iklM+tHrJL/bdL0NdfJXNCqn2nKrM51fpw
    diNXs4Zn3QSStC2x2hKnK+Q1rwCSEg/lBawgxGUslTboFH77a+Kwu4Oug9ibtm5z
    ISs/JY4Kiy4C2XJOltOR2XZYkdKaX4x3ctbrGaD8Bj+QHiSAxaaSXIX+VbzkHF2N
    aD5ijFUopjQEKFrYh3O93DB/URIQ+wHVa6+Kvu3uqE0cg6pQsLpbFVQ/I8xHvt9L
    kYy6z6V/nj9ZYKQbq/kPAgMBAAEwDQYJKoZIhvcNAQELBQADggIBAKZuc+lewYAv
    jaaSeRDRoToTb/yN0Xsi69UfK0aBdvhCa7/0rPHcv8hmUBH3YgkZ+CSA5ygajtL4
    g2E8CwIO9ZjZ6l+pHCuqmNYoX1wdjaaDXlpwk8hGTSgy1LsOoYrC5ZysCi9Jilu9
    PQVGs/vehQRqLV9uZBigG6oZqdUqEimaLHrOcEAHB5RVcnFurz0qNbT+UySjsD63
    9yJdCeQbeKAR9SC4hG13EbM/RZh0lgFupkmGts7QYULzT+oA0cCJpPLQl6m6qGyE
    kh9aBB7FLykK1TeXVuANlNU4EMyJ/e+uhNkS9ubNJ3vuRuo+ECHsha058yi16JC9
    NkZqP+df4Hp85sd+xhrgYieq7QGX2KOXAjqAWo9htoBhOyW3mm783A7WcOiBMQv0
    2UGZxMsRjlP6UqB08LsV5ZBAefElR344sokJR1de/Sx2J9J/am7yOoqbtKpQotIA
    XSUkATuuQw4ctyZLDkUpzrDzgd2Bt+aawF6sD2YqycaGFwv2YD9t1YlD6F4Wh8Mc
    20Qu5EGrkQTCWZ9pOHNSa7YQdmJzwbxJC4hqBpBRAJFl2fAlqFtyum6/8ZN9nZ9K
    FSEKdlu+xeb6Y6xYt0mJJWF6mCRi4i7IL74EU/VNXwFmfP6ladliUOST3w5t92cB
    M26t73UCExXMXTCQvnp0ki84PeR1kRk4
    -----END CERTIFICATE-----

    # Git server intermediate CA 2

    -----BEGIN CERTIFICATE-----
    MIIF5DCCA8wCCQDInYMol7LSDTANBgkqhkiG9w0BAQsFADCBszELMAkGA1UEBhMC

Q0ExCzAJBgNVBAgMAk9OMRAwDgYDVQQHDAdUb3JvbnRvMQ8wDQYDVQQKDAZSZW
RI

YXQxDDAKBgNVBAsMA0FDTTFFMEMGA1UEAww8Z29ncy1zdmMtZGVmYXVsdC5hcHBz
```

```
    LnJqdW5nLWh1YjEzLmRldjA2LnJlZC1jaGVzdGVyZmllbGQuY29tMR8wHQYJKoZI
    hvcNAQkBFhByb2tlakByZWRoYXQuY29tMB4XDTIwMTIwMzE4NTMxMloXDTIzMDky

MzE4NTMxMlowgbMxCzAJBgNVBAYTAkNBMQswCQYDVQQIDAJPTjEQMA4GA1UEBwwH


VG9yb250bzEPMA0GA1UECgwGUmVkSGF0MQwwCgYDVQQLDANBQ00xRTBDBgNVBA
MM
    PGdvZ3Mtc3ZjLWRlZmF1bHQuYXBwcy5yanVuZy1odWIxMy5kZXYwNi5yZWQtY2hl
    c3RlcmZpZWxkLmNvbTEfMB0GCSqGSIb3DQEJARYQcm9rZWpAcmVkaGF0LmNvbTCC
    AiIwDQYJKoZIhvcNAQEBBQADggIPADCCAgoCggIBAM3nPK4mOQzaDAo6S3ZJ0Ic3
    U9p/NLodnoTIC+cn0q8qNCAjf13zbGB3bfN9Zxl8Q5fv+wYwHrUOReCp6U/InyQy
    6OS3gj738F635inz1KdyhKtlWW2p9Ye9DUtx1IlfHkDVdXtynjHQbsFNIdRHcpQP
    upM5pwPC3BZXqvXChhlfAy2m4yu7vy0hO/oTzWIwNsoL5xt0Lw4mSyhlEip/t8lU
    xn2y8qhm7MiIUpXuwWhSYgCrEVqmTcB70Pc2YRZdSFoIMN9Et70MjQN0TXjoktH8
    PyASJIKIRd+48yROIbUn8rj4aYYBsJuoSCjJNwujZPbqseqUr42+v+Qp2bBj1Sjw
    +SEZfHTvSv8AqX0T6eo6njr578+DgYlwsS1A1zcAdzp8qmDGqvJDzwcnQVFmvaoM
    gGHCdJihfy3vDhxuZRDse0V4Pz6tl6iklM+tHrJL/bdL0NdfJXNCqn2nKrM51fpw
    diNXs4Zn3QSStC2x2hKnK+Q1rwCSEg/lBawgxGUslTboFH77a+Kwu4Oug9ibtm5z
    ISs/JY4Kiy4C2XJOltOR2XZYkdKaX4x3ctbrGaD8Bj+QHiSAxaaSXIX+VbzkHF2N
    aD5ijFUopjQEKFrYh3O93DB/URIQ+wHVa6+Kvu3uqE0cg6pQsLpbFVQ/l8xHvt9L
    kYy6z6V/nj9ZYKQbq/kPAgMBAAEwDQYJKoZIhvcNAQELBQADggIBAKZuc+lewYAv
    jaaSeRDRoToTb/yN0Xsi69UfK0aBdvhCa7/0rPHcv8hmUBH3YgkZ+CSA5ygajtL4
    g2E8CwIO9ZjZ6l+pHCuqmNYoX1wdjaaDXlpwk8hGTSgy1LsOoYrC5ZysCi9Jilu9
    PQVGs/vehQRqLV9uZBigG6oZqdUqEimaLHrOcEAHB5RVcnFurz0qNbT+UySjsD63
    9yJdCeQbeKAR9SC4hG13EbM/RZh0lgFupkmGts7QYULzT+oA0cCJpPLQl6m6qGyE
    kh9aBB7FLykK1TeXVuANlNU4EMyJ/e+uhNkS9ubNJ3vuRuo+ECHsha058yi16JC9
    NkZqP+df4Hp85sd+xhrgYieq7QGX2KOXAjqAWo9htoBhOyW3mm783A7WcOiBMQv0
    2UGZxMsRjlP6UqB08LsV5ZBAefElR344sokJR1de/Sx2J9J/am7yOoqbtKpQotIA
    XSUkATuuQw4ctyZLDkUpzrDzgd2Bt+aawF6sD2YqycaGFwv2YD9t1YlD6F4Wh8Mc
    20Qu5EGrkQTCWZ9pOHNSa7YQdmJzwbxJC4hqBpBRAJFl2fAlqFtyum6/8ZN9nZ9K
    FSEKdlu+xeb6Y6xYt0mJJWF6mCRi4i7IL74EU/VNXwFmfP6IadliUOST3w5t92cB
    M26t73UCExXMXTCQvnp0ki84PeR1kRk4
    -----END CERTIFICATE-----
```

2. Configure the channel with this ConfigMap. See the following sample with the **git-ca** name from the previous step:

```
apiVersion: apps.open-cluster-management.io/v1
kind: Channel
metadata:
  name: my-channel
  namespace: channel-ns
spec:
  configMapRef:
    name: git-ca
  pathname: <Git HTTPS URL>
  type: Git
```

1.5.5.4. Making an SSH connection to a Git server

1. Create a secret to contain your private SSH key in **sshKey** field in **data**. If the key is passphrase-protected, specify the password in **passphrase** field. This secret must be in the same namespace as the channel CR. Create this secret using a **oc** command to create a secret generic

git-ssh-key --from-file=sshKey=./.ssh/id_rsa, then add base64 encoded **passphrase**. See the following sample:

```
apiVersion: v1
kind: Secret
metadata:
  name: git-ssh-key
  namespace: channel-ns
data:
  sshKey:
LS0tLS1CRUdJTiBPUEVOU1NIIFBSSVZBVEUgS0VZLS0tLS0KYjNCbGJuTnphQzFyWlhrdG
RqRUFBQUFBQ21GbGN6STFOaTFqZEhJQUFBQUdZbU55ZVhCMEFBQUFHQUFBQUJD
K3YySHhWSIwCm8zejh1endzV3NWODMvSFVkOEtGeVBmWk5OeE5TQUgcFA3Yk1yR2tlRF
FPd3J6MGIKOUlRM0tKVXQzWEE0Zmd6NVlrVFVhcTJsZWxxVk1HcXI2WHF2UVJ5Mkc0NkRI
RVlYUGpabVZMcGVuaGtRYU5HYmpaMmZOdQpWUGpiOVhZRmd4bTNnYUpJU3BNeTFL
WjQ5MzJvOFByaDZEdzRYVUF1a28wZGdBaDdndVpPaE53b0pVYnNmYlZRc0xMS1RrCnQw
blZ1anRvd2NEVGx4TlpIUjcwbGVUSHdGQTYwekM0elpMNkRPc3RMYjV2LzZhMjFHRlMwVm
VXQ3YvMlpMOE1sbjVUZWwKSytoUWtxRnJBL3BUc1ozVXNjSG1GUi9PV25FPQotLS0tLUVO
RCBPUEVOU1NIIFBSSVZBVEUgS0VZLS0tLS0K
  passphrase: cGFzc3cwcmQK
type: Opaque
```

2. Configure the channel with the secret. See the following sample:

```
apiVersion: apps.open-cluster-management.io/v1
kind: Channel
metadata:
  name: my-channel
  namespace: channel-ns
spec:
  secretRef:
    name: git-ssh-key
  pathname: <Git SSH URL>
  type: Git
```

The subscription controller does an **ssh-keyscan** with the provided Git hostname to build the **known_hosts** list to prevent an Man-in-the-middle (MITM) attack in the SSH connection. If you want to skip this and make insecure connection, use **insecureSkipVerify: true** in the channel configuration. This is not best practice, especially in production environments.

```
apiVersion: apps.open-cluster-management.io/v1
kind: Channel
metadata:
  name: my-channel
  namespace: channel-ns
spec:
  secretRef:
    name: git-ssh-key
  pathname: <Git SSH URL>
  type: Git
  insecureSkipVerify: true
```

1.5.5.5. Updating certificates and SSH keys

If a Git channel connection configuration requires an update, such as CA certificates, credentials, or SSH key, you need to create a new secret and ConfigMap in the same namespace and update the channel to reference that new secret and ConfigMap. For more information, see Using custom CA certificates for a secure HTTPS connection.

1.5.6. Setting up Ansible Tower tasks

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

- Prerequisites
- Install Ansible Automation Platform Resource Operator
- Set up credential
- Ansible integration
- Ansible operator components
- Ansible configuration
- Set secret reconciliation
- Ansible sample YAML

1.5.6.1. Prerequisites

- OpenShift Container Platform 4.6 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.

1.5.6.2. 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*. **Note:** To submit prehook and posthook **AnsibleJobs**, install Ansible Automation Platform (AAP) Resource Operator with corresponding version available on different OpenShift Container Platform

versions:

- OpenShift Container Platform 4.6 needs (AAP) Resource Operator early-access
- OpenShift Container Platform 4.7 needs (AAP) Resource Operator early-access, stable-2.1
- OpenShift Container Platform 4.8 needs (AAP) Resource Operator early-access, stable-2.1, stable-2.2
- OpenShift Container Platform 4.9 needs (AAP) Resource Operator early-access, stable-2.1, stable-2.2
- OpenShift Container Platform 4.10 needs (AAP) Resource Operator stable-2.1, stable-2.2

1.5.6.3. Set up credential

You can create the credential you need from the *Credentials* page in the console. Click **Add credential** or access the page from the navigation. See Creating a credential for Ansible Automation Platform for credential information.

1.5.6.4. 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.

1.5.6.5. 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.

1.5.6.5.1. 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.

1.5.6.5.2. 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**.

1.5.6.5.3. 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.

1.5.6.6. Ansible configuration

You can configure Ansible Tower configurations with the following tasks:

1.5.6.6.1. 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.

1.5.6.7. 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

1.5.6.8. 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:
  name: 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
```

1.5.7. Configuring Helm to watch namespace resources

By default, when a subscription deploys subscribed Helm resources to target clusters, the application resources are watched. You can configure the Helm channel type to watch namespace-scoped resources. When enabled, manual changes to those watched namespace-scoped resources are reverted.

1.5.7.1. Configuring

Required access: Cluster administrator

To configure the Helm application to watch namespace scoped resources, set the value for the **watchHelmNamespaceScopedResources** field in your subscription definition to **true**. See the following sample.

```
apiVersion: apps.open-cluster-management.io/v1
kind: Subscription
metadata:
  name: nginx
  namespace: ns-sub-1
spec:
  watchHelmNamespaceScopedResources: true
  channel: ns-ch/predev-ch
  name: nginx-ingress
  packageFilter:
    version: "1.36.x"
```

1.5.8. Configuring Managed Clusters for OpenShift GitOps operator

To configure GitOps, you can register a set of one or more Red Hat Advanced Cluster Management for Kubernetes managed clusters to an instance of Red Hat OpenShift Container Platform GitOps operator. After registering, you can deploy applications to those clusters. Set up a continuous GitOps environment to automate application consistency across clusters in development, staging, and production environments.

1.5.8.1. Prerequisites

1. You need to install the Red Hat OpenShift GitOps operator on your Red Hat Advanced Cluster Management for Kubernetes.

2. Import one or more managed clusters.

1.5.8.2. Registering managed clusters to GitOps

1. Create managed cluster sets and add managed clusters to those managed cluster sets. See the example for managed cluster sets in the multicloud-integrations managedclusterset.
   See the Creating and managing ManagedClusterSets documentation for more information.

2. Create managed cluster set *binding* to the namespace where Red Hat OpenShift Container Platform GitOps is deployed.
   See the example in the repository at multicloud-integrations managedclustersetbinding, which binds to the **openshift-gitops** namespace.

   See the Creating a ManagedClusterSetBinding resource documentation for more information.

3. In the namespace that is used in managed cluster set binding, create a placement custom resource to select a set of managed clusters to register to an OpenShift Container Platform GitOps operator instance. You can use the example in the repository at multicloud-integration placement
   See Using ManagedClusterSets with Placement for placement information.

   Note: Only OpenShift Container Platform clusters are registered to an Red Hat OpenShift Container Platform GitOps operator instance, not other Kubernetes clusters.

4. Create a **GitOpsCluster** custom resource to register the set of managed clusters from the placement decision to the specified instance of Red Hat OpenShift Container Platform GitOps. This enables the Red Hat OpenShift Container Platform GitOps instance to deploy applications to any of those Red Hat Advanced Cluster Management managed clusters.
   Use the example in the repository at multicloud-integrations gitops cluster.

   Note: The referenced **Placement** resource must be in the same namespace as the **GitOpsCluster** resource.

   See from the following sample that **placementRef.name** is **all-openshift-clusters**, and is specified as target clusters for the GitOps instance that is installed in **argoNamespace: openshift-gitops**. The **argoServer.cluster** specification requires the **local-cluster** value.

```
apiVersion: apps.open-cluster-management.io/v1beta1
kind: GitOpsCluster
metadata:
  name: gitops-cluster-sample
  namespace: dev
spec:
  argoServer:
    cluster: local-cluster
    argoNamespace: openshift-gitops
  placementRef:
    kind: Placement
    apiVersion: cluster.open-cluster-management.io/v1beta1
    name: all-openshift-clusters
```

5. Save your changes. You can now follow the GitOps workflow to manage your applications. See About GitOps to learn more.

1.5.8.3. GitOps token

When you integrate with the GitOps operator for every managed cluster that is bound to the GitOps namespace through the placement and **ManagedClusterSetBinding** custom resources, a secret with a token to access the **ManagedCluster** is created in the namespace. This is required for the GitOps controller to sync resources to the managed cluster. When a user is given administrator access to a GitOps namespace to perform application lifecycle operations, the user also gains access to this secret and **admin** level to the managed cluster.

If this is not desired, instead of binding the user to the namespace-scoped **admin** role, use a more restrictive custom role with permissions required to work with application resources that can be created and used to bound the user. See the following **ClusterRole** example:

```
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRole
metadata:
  name: application-set-admin
rules:
- apiGroups:
  - argoproj.io
  resources:
  - applicationsets
  verbs:
  - get
  - list
  - watch
  - update
  - delete
  - deletecollection
  - patch
```

1.5.9. Scheduling a deployment

If you need to deploy new or change Helm charts or other resources during only specific times, you can define subscriptions for those resources to begin deployments during only those specific times. Alternatively, you can restrict deployments.

For instance, you can define time windows between 10:00 PM and 11:00 PM each Friday to serve as scheduled maintenance windows for applying patches or other application updates to your clusters.

You can restrict or block deployments from beginning during specific time windows, such as to avoid unexpected deployments during peak business hours. For instance, to avoid peak hours you can define a time window for a subscription to avoid beginning deployments between 8:00 AM and 8:00 PM.

By defining time windows for your subscriptions, you can coordinate updates for all of your applications and clusters. For instance, you can define subscriptions to deploy only new application resources between 6:01 PM and 11:59 PM and define other subscriptions to deploy only updated versions of existing resources between 12:00 AM to 7:59 AM.

When a time window is defined for a subscription, the time ranges when a subscription is active changes. As part of defining a time window, you can define the subscription to be *active* or *blocked* during that window.

The deployment of new or changed resources begins only when the subscription is active. Regardless of whether a subscription is active or blocked, the subscription continues to monitor for any new or changed resource. The active and blocked setting affects only deployments.

When a new or changed resource is detected, the time window definition determines the next action for the subscription.

- For subscriptions to **HelmRepo**, **ObjectBucket**, and **Git** type channels:
- If the resource is detected during the time range when the subscription is *active*, the resource deployment begins.
- If the resource is detected outside the time range when the subscription is blocked from running deployments, the request to deploy the resource is cached. When the next time range that the subscription is active occurs, the cached requests are applied and any related deployments begin.
- When a time window is *blocked*, all resources that were previously deployed by the application subscription remain. Any new update is blocked until the time window is active again.

End user may wrongly think when the app sub time window is blocked, all deployed resources will be removed. And they will be back when the app sub time window is active again.

If a deployment begins during a defined time window and is running when the defined end of the time window elapses, the deployment continues to run to completion.

To define a time window for a subscription, you need to add the required fields and values to the subscription resource definition YAML.

- As part of defining a time window, you can define the days and hours for the time window.
- You can also define the time window type, which determines whether the time window when deployments can begin occurs during, or outside, the defined time frame.
- If the time window type is **active**, deployments can begin only during the defined time frame. You can use this setting when you want deployments to occur within only specific maintenance windows.
- If the time window type is **block**, deployments cannot begin during the defined time frame, but can begin at any other time. You can use this setting when you have critical updates that are required, but still need to avoid deployments during specific time ranges. For instance, you can use this type to define a time window to allow security-related updates to be applied at any time except between 10:00 AM and 2:00 PM.
- You can define multiple time windows for a subscription, such as to define a time window every Monday and Wednesday.

1.5.10. 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
```

1.5.11. Channel samples overview

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 OpenShift 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:

```
oc apply -f filename.yaml
```

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

```
oc get application.app
```

- Channel YAML structure
- Channel YAML table
- Object storage bucket (ObjectBucket) channel

- Helm repository (**HelmRepo**) channel
- Git (**Git**) repository channel

1.5.11.1. Channel YAML structure

For application samples that you can deploy, see the **stolostron** repository.

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 and in the product console.

```
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:
```

1.5.11.2. Channel YAML table

Field	Optional or required	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 the 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.

Field	Optional or required	Description
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. The **gates** value is only for **ObjectBucket** channel types and does not apply to **HelmRepo** and **Git** 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.
spec.insecureSkipVerify	Optional	Default value is **false**, if set **true**, the channel connection is built by skipping the authentication

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/
```

1.5.11.3. 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:xxxx/dev] # URL is appended with the valid bucket name, which
matches the channel name.
 secretRef:
   name: miniosecret
 gates:
   annotations:
     dev-ready: true
```

1.5.11.4. Helm repository (HelmRepo) channel

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

Deprecation notice: For 2.6, specifying **insecureSkipVerify: "true"** in channel **ConfigMap** reference to skip Helm repo SSL certificate is deprecated. See the replacement in the following current sample, with **spec.insecureSkipVerify: true** that is used in the channel instead:

```
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.
    insecureSkipVerify: true
    type: HelmRepo
```

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 display 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/
```

1.5.11.5. Git (Git) repository channel

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

```
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
```

1.5.12. Subscription samples overview

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 OpenShift 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:

```
oc apply -f filename.yaml
```

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

```
oc get application.app
```

- Subscription YAML structure
- Subscription YAML table
- Subscription file samples
  - Subscription time window example
  - Subscription with overrides example
  - Helm repository subscription example
  - Git repository subscription example

1.5.12.1. 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:
```

1.5.12.2. Subscription YAML table

Field	Required or Optional	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.
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.

Field	Required or Optional	Description
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, 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.
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 override	Identifies the Kubernetes resource that is being overwritten.

Field	Required or Optional	Description
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.

Field	Required or Optional	Description
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**.
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.

Field	Required or Optional	Description
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.
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.

Field	Required or Optional	Description
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 specification, 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
```

1.5.12.3. Subscription file samples

For application samples that you can deploy, see the **stolostron** repository.

```
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
```

1.5.12.4. Secondary channel sample

If there is a mirrored channel (application source repository), you can specify a **secondaryChannel** in the subscription YAML. When an application subscription fails to connect to the repository server using the primary channel, it connects to the repository server using the secondary channel. Ensure that the application manifests stored in the secondary channel are in sync with the primary channel. See the following sample subscription YAML with the **secondaryChannel**.

```
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
  secondaryChannel: ns-ch-2/predev-ch-2
  name: nginx-ingress
```

1.5.12.4.1. 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: towhichcluster
  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"
```

1.5.12.4.2. 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
```

1.5.12.4.3. 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
```

1.5.12.4.4. Git repository subscription example

1.5.12.4.4.1. 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.

Note: When you are using a Git channel subscription that subscribes to Helm charts, the resource topology view might show an additional **Helmrelease** resource. This resource is an internal application management resource and can be safely ignored.

1.5.12.4.4.2. 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.

1.5.12.4.4.3. 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.

1.5.12.4.4.4. 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.

1.5.12.4.4.4.1. 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**.

1.5.12.4.4.4.2. 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.

1.5.12.4.4.4.3. Configuring WebHook in Git repository

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

1.5.12.4.4.4.4. 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>"
```

No webhook specific configuration is needed in subscriptions.

1.5.13. Placement rule samples overview

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 OpenShift 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:

```
oc apply -f filename.yaml
```

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

```
oc get application.app
```

- Placement rule YAML structure
- Placement rule YAML values table
- Placement rule sample files

1.5.13.1. 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 and in the product console

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

1.5.13.2. Placement rule YAML values table

Field	Required or Optional	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.
spec.clusterSelector.matchExpressions	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.

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

1.5.13.3. Placement rule sample files

For application samples that you can deploy, see the **stolostron** repository.

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: towhichcluster
  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
```

1.5.14. 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 OpenShift 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:

```
oc apply -f filename.yaml
```

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

```
oc get application.app
```

- Application YAML structure
- Application YAML table
- Application file samples

1.5.14.1. 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
```

1.5.14.2. 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		

Field	Value	Description
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.

1.5.14.3. Application file samples

For application samples that you can deploy, see the **stolostron** repository.

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
```