

## Configuring NiFi CR

Date published: 2024-06-11

Date modified: 2026-05-18



# Legal Notice

© Cloudera Inc. 2026. All rights reserved.

The documentation is and contains Cloudera proprietary information protected by copyright and other intellectual property rights. No license under copyright or any other intellectual property right is granted herein.

Unless otherwise noted, scripts and sample code are licensed under the Apache License, Version 2.0.

Copyright information for Cloudera software may be found within the documentation accompanying each component in a particular release.

Cloudera software includes software from various open source or other third party projects, and may be released under the Apache Software License 2.0 (“ASLv2”), the Affero General Public License version 3 (AGPLv3), or other license terms. Other software included may be released under the terms of alternative open source licenses. Please review the license and notice files accompanying the software for additional licensing information.

Please visit the Cloudera software product page for more information on Cloudera software. For more information on Cloudera support services, please visit either the Support or Sales page. Feel free to contact us directly to discuss your specific needs.

Cloudera reserves the right to change any products at any time, and without notice. Cloudera assumes no responsibility nor liability arising from the use of products, except as expressly agreed to in writing by Cloudera.

Cloudera, Cloudera Altus, HUE, Impala, Cloudera Impala, and other Cloudera marks are registered or unregistered trademarks in the United States and other countries. All other trademarks are the property of their respective owners.

Disclaimer: EXCEPT AS EXPRESSLY PROVIDED IN A WRITTEN AGREEMENT WITH CLOUDERA, CLOUDERA DOES NOT MAKE NOR GIVE ANY REPRESENTATION, WARRANTY, NOR COVENANT OF ANY KIND, WHETHER EXPRESS OR IMPLIED, IN CONNECTION WITH CLOUDERA TECHNOLOGY OR RELATED SUPPORT PROVIDED IN CONNECTION THEREWITH. CLOUDERA DOES NOT WARRANT THAT CLOUDERA PRODUCTS NOR SOFTWARE WILL OPERATE UNINTERRUPTED NOR THAT IT WILL BE FREE FROM DEFECTS NOR ERRORS, THAT IT WILL PROTECT YOUR DATA FROM LOSS, CORRUPTION NOR UNAVAILABILITY, NOR THAT IT WILL MEET ALL OF CUSTOMER’S BUSINESS REQUIREMENTS. WITHOUT LIMITING THE FOREGOING, AND TO THE MAXIMUM EXTENT PERMITTED BY APPLICABLE LAW, CLOUDERA EXPRESSLY DISCLAIMS ANY AND ALL IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO IMPLIED WARRANTIES OF MERCHANTABILITY, QUALITY, NON-INFRINGEMENT, TITLE, AND FITNESS FOR A PARTICULAR PURPOSE AND ANY REPRESENTATION, WARRANTY, OR COVENANT BASED ON COURSE OF DEALING OR USAGE IN TRADE.

# Contents

|                                                                                              |               |
|----------------------------------------------------------------------------------------------|---------------|
| <b>Configuring a NiFi instance.....</b>                                                      | <b>4</b>      |
| Resource recommendations for NiFi deployments.....                                           | 4             |
| Example NiFi cluster size.....                                                               | 4             |
| Group, version, kind, meta.....                                                              | 7             |
| Configuring environment variables.....                                                       | 7             |
| Configuring the NiFi image.....                                                              | 7             |
| Configuring volumes and mounts.....                                                          | 9             |
| Configuring cluster size.....                                                                | 9             |
| Configuring out of memory recovery.....                                                      | 10            |
| Configuring cluster scheduling.....                                                          | 11            |
| Configuring bootstrap JVM settings.....                                                      | 12            |
| Configuring persistence.....                                                                 | 13            |
| Configuring assets.....                                                                      | 13            |
| Configuring NAR providers.....                                                               | 14            |
| Configuring Kubernetes state management.....                                                 | 15            |
| Configuring node certificate generation.....                                                 | 15            |
| Configuring additional CA bundles.....                                                       | 16            |
| Securing communication between NiFi clusters using the Default Node SSL Context Service..... | 16            |
| Configuring NiFi properties.....                                                             | 18            |
| Overriding NiFi settings using ConfigMaps and Secrets.....                                   | 18            |
| Configuring scaling.....                                                                     | 19            |
| Security context configuration for Nifi.....                                                 | 19            |
| Configuring pod affinity.....                                                                | 21            |
| Configuring connections to NiFi.....                                                         | 21            |
| Configuring session affinity.....                                                            | 22            |
| Configuring arbitrary connections.....                                                       | 22            |
| Configuring NiFi Web UI connection.....                                                      | 22            |
| Configuring additional proxy hosts.....                                                      | 24            |
| Configuring authentication and authorization for NiFi.....                                   | 24            |
| Configuring the initial admin user.....                                                      | 24            |
| Configuring single user authentication.....                                                  | 25            |
| Configuring a user identity with access policies.....                                        | 26            |
| Configuring user groups.....                                                                 | 28            |
| Configuring NiFi access policy profiles.....                                                 | 31            |
| Configuring LDAP authentication.....                                                         | 33            |
| Configuring OIDC authentication.....                                                         | 34            |
| Configuring Ranger authorization for NiFi.....                                               | 35            |
| Configuring JVM security providers (FIPS).....                                               | 44            |
| Configuration.....                                                                           | 45            |
| <br><b>Example CR.....</b>                                                                   | <br><b>47</b> |

## Configuring a NiFi instance

NiFi instances are configured through the CRs used to deploy them.

A custom resource (CR) is a YAML file that describes your desired NiFi deployments. This single file contains all configuration information required for the NiFi instance, no additional configuration is required after deployment.

This documentation provides sample configuration code snippets to help you create a CR.

## Resource recommendations for NiFi deployments

Learn about the recommended resource sizes for NiFi deployments. Every NiFi deployment is unique on the basis of the purpose it serves, therefore the values here are just recommendations not requirements. Actual values may substantially differ depending on your use case.

| Resource Type | Amount                 |
|---------------|------------------------|
| CPU           | 2+ per Pod             |
| Memory        | 4Gi+ per Pod           |
| PVC/PV        | 5 per Pod              |
| Secrets       | 4 + #Pods              |
| ConfigMaps    | 13                     |
| Services      | 1 + #Connections       |
| Pods          | 1 min, 3+ recommended  |
| StatefulSet   | 1                      |
| Deployment    | 0                      |
| Ingress       | 1 + #IngressConnection |

## Example NiFi cluster size

The following list of resources represents the whole of a deployed NiFi cluster managed by the Cloudera Flow Management Operator for Kubernetes. The following example is run in kind with cert-manager and ingress-nginx deployed as dependencies.

```
apiVersion: cfm.cloudera.com/v1alpha1
kind: Nifi
metadata:
  name: mynifi
spec:
  replicas: 3
  nifiVersion: 1.0.0
  image:
    repository: container.repository.cloudera.com/cloudera/cfm-nifi-k8s
    tag: 2.8.0-b94-nifi_1.25.0.2.3.13.0-36
    pullSecret: cloudera-container-repository-credentials
    pullPolicy: IfNotPresent
  tiniImage:
    repository: container.repository.cloudera.com/cloudera/cfm-tini
    tag: 2.8.0-b94
    pullSecret: cloudera-container-repository-credentials
    pullPolicy: IfNotPresent
  persistence:
    size: 1Gi
    contentRepo:
```

```

    size: 1Gi
  flowfileRepo:
    size: 1Gi
  provenanceRepo:
    size: 2Gi
  data: {}
  security:
    initialAdminIdentity: anonymous
    nodeCertGen:
      issuerRef:
        name: self-signed-ca-issuer
        kind: ClusterIssuer
    singleUserAuth:
      enabled: true
      credentialsSecretName: creds
  hostName: nifi.io
  uiConnection:
    type: Ingress
    annotations:
      nginx.ingress.kubernetes.io/affinity: cookie
      nginx.ingress.kubernetes.io/affinity-mode: persistent
      nginx.ingress.kubernetes.io/backend-protocol: HTTPS
      nginx.ingress.kubernetes.io/ssl-passthrough: "true"
      nginx.ingress.kubernetes.io/ssl-redirect: "true"
  resources:
    nifi:
      requests:
        cpu: "1"
        memory: 2Gi
      limits:
        cpu: "4"
        memory: 4Gi
    log:
      requests:
        cpu: 50m
        memory: 128Mi

StatefulSet
$ kubectl get statefulset
NAME      READY   AGE
mynifi    3/3     24h
Pods
$ kubectl get pod
NAME      READY   STATUS    RESTARTS   AGE
mynifi-0  7/7     Running   0           23h
mynifi-1  7/7     Running   0           23h
mynifi-2  7/7     Running   0           23h
ConfigMaps
$ kubectl get configmap
NAME                                DATA   AGE
mynifi-authorizers                  1       24h
mynifi-authorizers-empty             1       24h
mynifi-bootstrap                     1       24h
mynifi-certificate-setup-script      1       24h
mynifi-decommission-script           1       24h
mynifi-identities-config              1       24h
mynifi-logback                       1       24h
mynifi-login-identity-providers       1       24h
mynifi-nifi-cli-properties            1       24h
mynifi-nifi-properties                1       24h
mynifi-start-script                  1       24h
mynifi-state-management               1       24h
mynifi-stop-script                   1       24h
Secrets

```

```

$ kubectl get secret
NAME                                TYPE                                DATA  AGE
creds                               Opaque                              2      27h
mynifi-0-node-cert                  kubernetes.io/tls                   5      24h
mynifi-1-node-cert                  kubernetes.io/tls                   5      23h
mynifi-2-node-cert                  kubernetes.io/tls                   5      23h
mynifi-keystorepassword              Opaque                              1      24h
mynifi-proxy-cert                   kubernetes.io/tls                   3      27h
mynifi-sensitive-props-key          Opaque                              1      24h
Certificates
$ kubectl get certificates
NAME                                READY  SECRET                                AGE
mynifi-0-node-cert                  True   mynifi-0-node-cert                  24h
mynifi-1-node-cert                  True   mynifi-1-node-cert                  23h
mynifi-2-node-cert                  True   mynifi-2-node-cert                  23h
mynifi-proxy-cert                   True   mynifi-proxy-cert                  24h
Services
$ kubectl get service
NAME                                TYPE                                CLUSTER-IP      EXTERNAL-IP      PORT(S)
AGE
mynifi                             ClusterIP          None             <none>           6007/TCP,5000/TCP  24h
mynifi-web                         ClusterIP          10.96.28.159    <none>           8443/TCP
24h
Ingresses
$ kubectl get ingresses
NAME                                CLASS    HOSTS    ADDRESS    PORTS    AGE
mynifi-web                         <none>   nifi.io  localhost  80       24h

PersistentVolumeClaims
$ kubectl get persistentvolumeclaim
NAME                                STATUS    VOLUME
CAPACITY  ACCESS MODES  STORAGECLASS  AGE
content-repository-mynifi-0        Bound    pvc-d5b00d05-d8ee-4b5c-abe4-2cae6161fa4b  1Gi  RWO  standard  24h
content-repository-mynifi-1        Bound    pvc-3a510ebf-2f63-409b-992b-5a08480d4b31  1Gi  RWO  standard  23h
content-repository-mynifi-2        Bound    pvc-f1baf8e-b8b2-485c-a0c8-ddb583ac994b  1Gi  RWO  standard  23h
data-mynifi-0                      Bound    pvc-67072ae8-b1ae-445c-b81a-07719417a441  1Gi  RWO  standard  24h
data-mynifi-1                      Bound    pvc-4d20e11b-0d93-4b1b-95ff-a19189506686  1Gi  RWO  standard  23h
data-mynifi-2                      Bound    pvc-71b92ed3-3a70-4c4d-a848-44f879896f59  1Gi  RWO  standard  23h
flowfile-repository-mynifi-0       Bound    pvc-c9c0ea11-0f59-4eeb-b9ac-a795b562c059  1Gi  RWO  standard  24h
flowfile-repository-mynifi-1       Bound    pvc-58200919-b8b2-43be-8d2f-16b1f7d39a75  1Gi  RWO  standard  23h
flowfile-repository-mynifi-2       Bound    pvc-869c0159-51c7-4857-aa52-8c775c709692  1Gi  RWO  standard  23h
provenance-repository-mynifi-0     Bound    pvc-97dc6f41-b8ea-4682-9f60-74c8756fb344  2Gi  RWO  standard  24h
provenance-repository-mynifi-1     Bound    pvc-bd529e22-7024-4bef-a951-0a3fb753277b  2Gi  RWO  standard  23h
provenance-repository-mynifi-2     Bound    pvc-cbcc2b50-3404-4fea-8c3c-6a2fe2bfbdb13  2Gi  RWO  standard  23h
state-mynifi-0                     Bound    pvc-d0e72637-2b7d-40b7-a147-7240942599a4  1Gi  RWO  standard  24h
state-mynifi-1                     Bound    pvc-c17f3fe2-b93d-4774-b49d-5564e794c671  1Gi  RWO  standard  23h
state-mynifi-2                     Bound    pvc-19a010f0-0397-496b-a77d-89944b94a9b0  1Gi  RWO  standard  23h

```

## Related Information

[kind documentation](#)

## Group, version, kind, meta

This is the initial section of your YAML file that you need to specify in all cases.

You need to add the following section to the top of each NiFi CR you write. It defines the group “cfm.cloudera.com”, the version “v1”, the kind “Nifi”, and the name of your cluster and the NiFi nodes. It can also specify the namespace in which resources will be deployed. It is expected that a single NiFi cluster is deployed in a given namespace. You can also specify namespace during deployment. If that is what you want, omit namespace from the CR

```

apiVersion: cfm.cloudera.com/v1alpha1
kind: Nifi
metadata:
  name: [***NIFI CLUSTER NAME***]
  namespace: [***NIFI CLUSTER NAMESPACE***]

```

Replace [\*\*\*NIFI CLUSTER NAME\*\*\*] and [\*\*\*NIFI CLUSTER NAMESPACE\*\*\*] with the desired cluster name and cluster namespace respectively.

## Configuring environment variables

Environment variables can be added to the NiFi container using the following spec:

```

spec:
  statefulset:
    env:
      - name: [***VARIABLE NAME***]
        value: [***VARIABLE VALUE***]

```

## Configuring the NiFi image

Specify location of the image used for deployment.

This is how you specify the NiFi image repository, image version, and NiFi version to be used for deployment. This describes the images used for running NiFi. This also provides a way of manually upgrading the NiFi version in an existing cluster or very quickly rolling out NiFi clusters with new versions.

```

spec:
  nifiVersion: [***NIFI VERSION***]
  image:
    repository: container.repository.cloudera.com/cloudera/cfm-nifi-k8s
    tag: [***IMAGE VERSION TAG***]

```



### Note:

container.repository.cloudera.com/cloudera/cfm-nifi-k8s is the default repository for Cloudera Kubernetes images. If your Kubernetes cluster has no internet access or you want to use a self-hosted repository, replace it with the relevant path.

## Specifying the NiFi version



**Note:** Cloudera Flow Management Operator for Kubernetes uses the NiFi version to enable version-specific features and validate compatibility. The version can be supplied explicitly via the spec.nifiVersion field in the Nifi custom resource, or it can be detected automatically from the image tag.

### Automatic detection from the image tag

When the image tag contains a segment matching the pattern `nifi_X.Y.Z` (where X, Y, and Z are integers), the operator extracts the NiFi version from it automatically. This is the format used by Cloudera-produced images:

```
apiVersion: cfm.cloudera.com/v1alpha1
kind: Nifi
metadata:
  name: my-nifi
spec:
  image:
    tag: "3.1.0-b107-nifi_2.6.0.4.12.0.1-2"
```

| Image tag                        | Detected NiFi version            |
|----------------------------------|----------------------------------|
| 3.1.0-b107-nifi_2.6.0.4.12.0.1-2 | 2.6.0                            |
| my-build-nifi_1.28.0-rc1         | 1.28.0                           |
| latest                           | NiFi version cannot be detected. |

### spec.nifiVersion field

If, for some reason, it is not viable to use an image tag compatible with the automatic detection, you can provide a version in the Nifi spec:

```
apiVersion: cfm.cloudera.com/v1alpha1
kind: Nifi
metadata:
  name: my-nifi
spec:
  nifiVersion: "2.6.0"
  image:
    tag: "latest"
```

The value must be a valid SemVer string (for example, 1.28.0 or 2.6.0). Only NiFi major versions 1 and 2 are supported.

### Version resolution

When Cloudera Flow Management Operator for Kubernetes needs the NiFi version, it uses the following order of precedence:

1. `spec.nifiVersion` - if set, this value is always used
2. Image tag - if `spec.nifiVersion` is absent, the version is parsed from the image tag

If neither source provides a version, the CR is rejected to protect from misconfiguration.

### Validation and warnings

#### Mismatch warning

If both `spec.nifiVersion` is set and the image tag contains a `nifi_X.Y.Z` segment, the operator compares them. When they differ, a warning is returned on create or update actions:

```
Detected NiFi version '2.6.0' from image tag does not match the
specified NiFi version '1.0.0'
in the CR. Please ensure that the image tag correctly reflects th
e NiFi version or update the
spec.nifiVersion field to match the detected version.
```



The warning does not block the operation, but it indicates a likely misconfiguration. Check that the image being used actually contains the NiFi version specified in `spec.nifiVersion`.

### No detectable version (error)

If `spec.nifiVersion` is absent and the image tag does not contain a `nifi_X.Y.Z` segment, the CR is rejected with a similar error message:

```
NiFi version cannot be determined from either the spec.nifiVersion field or the image tag.
Please specify a valid NiFi version in the spec.nifiVersion field
or use an image with a tag
that includes the NiFi version.
```

This situation does not arise with Cloudera-produced images.

### Best Practices

- For custom images, always set `spec.nifiVersion` explicitly when using images whose tags do not follow the `nifi_X.Y.Z` pattern.
- The `nifi_X.Y.Z` segment is embedded in all Cloudera-produced image tags, adding `spec.nifiVersion` is optional. Set it explicitly if you want to be unambiguous or if you want to guard against accidental image tag changes.
- To avoid the mismatch warning, keep `spec.nifiVersion` in sync with the actual NiFi version in the image, or omit it and rely on image tag detection.

### Related Information

[Semantic versioning \(SemVer\)](#)

## Configuring volumes and mounts

Arbitrary volumes and volume mounts can be added to the NiFi container. This can be used to provide Python scripts or other artifacts to the NiFi runtime. Combined with a Cloud Storage Interface driver and Persistent Volume Claim, you can give NiFi access to files in cloud storage such as EFS or S3 buckets.

Define your custom volumes and mounts with the following spec:

```
spec:
  statefulset:
    volumes:
      - name: foo-volume
        persistentVolumeClaim: foo-pvc
    volumeMounts:
      - name: foo-volume
        mountPath: /opt/nifi/foo
```

To learn more about Kubernetes volumes, see [Persistent volumes in the Kubernetes documentation](#).

## Configuring cluster size

Specify the number of pods in your deployment.

This section configures the number and capacity of your pods in the cluster.

```
spec:
  replicas: [***NUMBER OF REPLICAS***]
  resources:
    nifi:
      requests:
        cpu: "[***CPU IN CORES***]"
```

```

    memory: [***MEMORY IN BITES***]
  limits:
    cpu: "[***CPU IN CORES***]"
    memory: [***MEMORY IN BITES***]
  log:
    requests:
      cpu: [***CPU IN CORES***]
      memory: [***MEMORY IN BITES***]

```

## Configuring out of memory recovery

You can optionally specify the step size in memory increase to prevent out of memory (OOM) crashes to your pods. You can also specify an upper bound to memory increase, to prevent infinite scaling.

The Cloudera Flow Management Operator for Kubernetes can detect an Out of Memory event in a NiFi cluster and scale up the memory footprint when configured for Out of Memory Recovery. This feature is not preventative but responsive, the NiFi cluster must first run out of memory and fail a Readiness check before the recovery attempt will be made, potentially impacting Flow performance. OOM Recovery is intended to be a safe guard and is not a replacement for good cluster sizing. If OOM Recovery has triggered, it is recommended that you reevaluate your NiFi resource sizing.

OOM Recovery has two fields to configure: `stepSize` and `upperBound`. `stepSize` defines the amount of memory that should be added for each OOM event. `upperBound` defines the maximum amount of memory to which the OOM Recovery process is allowed to grow.

```

spec:
  outOfMemoryRecovery:
    stepSize: [***DEFINES THE MEMORY INCREASE EVERY TIME PODS ARE
OOMKILLED***]
    upperBound: [***SPECIFIES THE UPPER LIMIT OF MEMORY INCREASE FOR MEMORY
PROTECTION***]

```

For example:

```

spec:
  outOfMemoryRecovery:
    stepSize: 1Gi
    upperBound: 8Gi
  resources:
    nifi:
      requests:
        cpu: "1"
        memory: 4Gi

```

The above spec starts with NiFi containers at 4Gi and will grow by 1Gi for every OOM that occurs until the NiFi container memory reaches 8Gi. When only memory requests are provided, the NiFi container memory request will grow. If memory limits are provided, only the memory limit will grow.

Note: This can break [Quality of Service](#) for the Pod, in the future the requests and limits will grow proportionately.

Once the OOM Recovery has taken effect, it will never automatically scale down. Removal of the OOM Recovery growth will occur when a NiFi resource spec change is detected or when OOM Recovery is removed from the NiFi spec.

## NiFi Resource Conditions

The following status field and condition have been added to track the OOM Recovery process:

```

status:
  conditions:
    - lastTransitionTime: "2025-04-15T16:16:15Z"
      message: NiFi has vertically scaled for OOM recovery

```

```

observedGeneration: 2
reason: OOMRecoveryScaleUp
status: "False"
type: VerticallyScaleUp
outOfMemoryRecoveryGrowth: 500Mi

```

The field `outOfMemoryRecoveryGrowth` tracks how much the NiFi memory has already grown. The `VerticallyScaleUp` condition provides the last time the cluster scaled up as well as if the scaling action is complete or not. While the status of `VerticallyScaleUp` is “True”, the scaling is in progress. Once the scaling action is complete, the status is set to “False”.

## Configuring cluster scheduling

Set a schedule for a NiFi cluster.

A field, `clusterSchedule`, defines the “up time” of your NiFi cluster. During the down time, the cluster is suspended which means all data and data flow configuration is still persisted. For more information, see *Cluster Suspension*.

The `clusterSchedule` supports two time formats for declaring a schedule: cron and time range.

### Cron

The cron scheduler takes in a standard [Cron expression](#) and a run duration. The following example unsuspends the NiFi cluster once every three hours, starting at 00:00, and runs for one hour before suspending again.

```

spec:
  clusterSchedule:
    cron:
      schedule: "0 */3 * * *"
      runDuration: 1h

```

The `runDuration` field supports setting minutes (30m) and hours (2h) or a combination (2h30m).

### TimeRange

`TimeRange` specifies two wall clock time instances between which the NiFi cluster will run. The following example runs daily between the hours of 12:00 and 13:00 UTC.

```

spec:
  clusterSchedule:
    timeRange:
      startTime: "12:00:00Z"
      stopTime: "13:00:00Z"

```

To configure `timeRange` to begin at night and finish in the morning, `timeRange` supports setting a `stopTime` that is earlier than the `startTime`. The following example begins running at 22:00 UTC and ends at 03:00 UTC the following day.

```

spec:
  clusterSchedule:
    timeRange:
      startTime: "22:00:00Z"
      stopTime: "03:00:00Z"

```

Both `startTime` and `stopTime` support numerical time offsets for timezones. For example, “12:00:00-04:00” is noon (12pm) in Eastern Standard Time and 16:00 in UTC.

## NiFi resource conditions

The NiFi Resource Status will show a ClusterScheduled condition.

When there is no schedule, the condition status will be False and will indicate that there is no schedule.

```
conditions:
- lastTransitionTime: "2025-04-15T15:51:52Z"
  message: NiFi cluster has no schedule
  observedGeneration: 2
  reason: NoClusterSchedule
  status: "False"
  type: ClusterScheduled
```

When a schedule is set and the NiFi cluster is currently not running, the ClusterScheduled condition will have status True and state the time at which the cluster will be restored.

```
conditions:
- lastTransitionTime: "2025-04-15T20:45:12Z"
  message: NiFi cluster is scheduled to be restored at 2025-04-15T20:50:00Z
  observedGeneration: 1
  reason: ClusterScheduledForRestoration
  status: "True"
  type: ClusterScheduled
```

When a schedule is set and the NiFi cluster is currently running, the ClusterScheduled condition will have status True and state the time at which the cluster will be suspended.

```
conditions:
- lastTransitionTime: "2025-04-15T20:45:12Z"
  message: NiFi cluster is scheduled to be suspended at 2025-04-15T20:48:00Z
  observedGeneration: 1
  reason: ClusterScheduledForSuspension
  status: "True"
  type: ClusterScheduled
```

## Configuring bootstrap JVM settings

Learn about Java memory calculations, defaults and how to override them with custom values.

Cloudera Flow Management Operator for Kubernetes calculates JVM memory setting Max Direct Memory Size, Min Heap Size (xms), and Max Heap Size (mx) based on container memory limits or requests.

In bootstrap settings, java.arg.10 is DirectMem, java.arg.2 is Min Heap, and java.arg.3 is Max Heap.

Java memory is calculated and set in the following order:

### 1. Based on memory of the NiFi resource

- The minimum DirectMem allowed for NiFi is 512MB. DirectMem is set to the maximum value between 512MB and 10% of the memory limit. If you do not provide a memory limit, the same calculation is made on the memory request in the specifications.
- Min Heap Size (xms) and Max Heap Size (mx) is set to 75% of the memory limit subtracting the calculated DirectMem.

### 2. Defaults

If you do not specify memory for the NiFi resource, the following default values are automatically set:

- java.arg.2: -Xms2g

- java.arg.3: -Xmx2g
- java.arg.10: -XX:MaxDirectMemorySize=512m

### Advanced configuration: Custom values to override inbuilt memory calculations

You can set each java argument for memory as part of NiFi specifications, under the configOverride key.

```
spec:
  configOverride:
    bootstrapConf:
      upsert:
        java.arg.2: -Xms2g
        java.arg.3: -Xmx2g
        java.arg.10: -XX:MaxDirectMemorySize=512m
```

## Configuring persistence

Specify storage size and class globally, or for individual repositories.

This section specifies the storage to be used for the NiFi repositories. You can define storage globally, or have overrides for specific repositories. In case of OpenShift, the storage classes have to be specified at the OpenShift level to match the IOPS expectations for your NiFi workloads.

The Cloudera Flow Management Operator for Kubernetes can configure persistent disk storage for the following directories:

- state
- data
- FlowFile Repository
- Content Repository
- Provenance Repository

In the persistence spec, you can define a default size and StorageClass which applies to each of the directories. The spec can be further configured to define specific sizes and StorageClasses for each directory, if necessary.

```
spec:
  persistence:
    size: [***SIZE IN GIGABITES***]
    storageClass: [***STORAGE CLASS***]
    contentRepo:
      size: [***SIZE IN GIGABITES***]
      storageClass: [***STORAGE CLASS***]
    flowfileRepo:
      size: [***SIZE IN GIGABITES***]
    provenanceRepo:
      size: [***SIZE IN GIGABITES***]
```

## Configuring assets

Learn about configuring access to NiFi assets.

You can make NiFi assets, like configuration files available to your NiFi cluster using the assets field. This field allows you to specify a mount path within the NiFi Pods to which the provided pre-existing Persistent Volume Claim (PVC) is mounted. Cloudera Flow Management Operator for Kubernetes does not provide a method of loading assets into this volume. Using the example below, all files located in the volume associated with my-nifi-assets-volume-claim are accessible at the path /opt/nifi/nifi-assets/ for use within your flow.

Before you can start using assets with NiFi deployed through Cloudera Flow Management Operator for Kubernetes, you have to provide the following:

- A volume provisioner which supports creating volumes that are ReadWriteMany (RWX), for example nfs-provisioner.
- A pod running some kind of software or process (for example, an FTP server) attached to the RWX volume for loading files onto the volume. Alternatively, you can use the `kubectl cp` command to directly copy files into most containers, such as Ubuntu.

After meeting the above prerequisites, you need to create a Persistent Volume Claim (PVC) which creates the required RWX volume. For example:

```
kind: PersistentVolumeClaim
apiVersion: v1
metadata:
  name: nifi-assets
spec:
  storageClassName: [***STORAGE CLASS**]
  accessModes:
    - ReadWriteMany
  resources:
    requests:
      storage: 1Gi
```

where `[***STORAGE CLASS**]` refers to the Storage Class associated with the RWX volume provisioner.

With the PVC created and the PV provisioned, attach the pod you have created, expose the pod via Ingress or Service if required, and load the asset files.

```
spec:
  assets:
    mountPath: [***ASSETS PATH**]
    persistentVolumeClaim:
      name: nifi-assets
```

where `[***ASSETS PATH**]` is the filesystem path within the NiFi container where the assets are located, for example, `/opt/nifi/assets`.

### Related Information

[ReadWriteMany \(RWX\)](#)

[nfs-provisioner](#)

[Persistent Volume Claim \(PVC\)](#)

## Configuring NAR providers

Provide custom NARs to NiFi.

### NAR provider volumes

Custom NARs can be provided to NiFi via Kubernetes volumes. The volumes used for NARs must support RWX (Read Write Many) access mode, such as an NFS volume. You need a Persistent Volume Claim which references your RWX volume, such as:

```
kind: PersistentVolumeClaim
apiVersion: v1
metadata:
  name: [***YOUR VOLUME CLAIM NAME**]
spec:
  storageClassName: "nfs"
  accessModes:
```

```
- ReadWriteMany
resources:
  requests:
    storage: 10Mi
```

The above storage class provisions an NFS volume from the nfs-server-provisioner, if it is installed in your cluster. Cloud provider classes like EFS or S3 on AWS can be used if their CSI drivers are installed.

Finally, provide your persistent volume claim to the NiFi spec as follows. You can optionally provide a subPath for the volume if you wish to specify only a certain directory within that volume.

```
spec:
  narProvider:
    volumes:
      - volumeClaimName: [***YOUR VOLUME CLAIM NAME***]
        subPath: [***OPTIONAL SUBPATH***]
      - volumeClaimName: [***ANOTHER VOLUME CLAIM***]
```

### Related Information

[Persistent volumes | Kubernetes](#)

[NFS Server Provisioner | ArtifactHUB](#)

[CSI Driver for Amazon EFS | GitHub](#)

[Mountpoint CSI driver for Amazon S3 | GitHub](#)

## Configuring Kubernetes state management

Specify Kubernetes native state management provider as the state management provider of your cluster.

Cloudera's distribution of NiFi comes with a Kubernetes native state management provider. This is the recommended state management for use with Cloudera Flow Management Operator for Kubernetes. However, as it is not the default state management provider set by Cloudera Flow Management Operator for Kubernetes, you need to add this section to the configuration. Without this configuration, a ZooKeeper cluster is expected.

To configure the Kubernetes state management provider, use the below YAML.

```
spec:
  stateManagement:
    clusterProvider:
      id: kubernetes-provider
      class: org.apache.nifi.kubernetes.state.provider.KubernetesConfigMapsStateProvider
    configOverride:
      nifiProperties:
        upsert:
          nifi.cluster.leader.election.implementation: "KubernetesLeaderElectionManager"
```

## Configuring node certificate generation

Learn about certificate generation options.



### Note:

NiFi deployed by the Cloudera Flow Management Operator for Kubernetes is secure by default. If you do not provide an authorization method, NiFi is configured in single user auth mode with username "demonstration-user" and password "demonstration-password". If you do not specify Node Certificate Generation, node certificates are generated using a self-signed Cluster Issuer nifi-node-certs that is deployed with the operator.

Cloudera Flow Management Operator for Kubernetes provides automatic certificate generation for each NiFi node in a given cluster by way of cert-manager certificates to secure intra-cluster communication between NiFis. To configure nodeCertGen, a cert-manager Issuer or ClusterIssuer is required. A self-signed Issuer setup is sufficient for development environments. In production environments use a third-party authority, or internal signing CAs.

```
spec:
  security:
    nodeCertGen:
      issuerRef:
        name: self-signed-ca-issuer
        kind: ClusterIssuer
```

### Related Information

[Issuers and ClusterIssuers](#)

## Configuring additional CA bundles

Add custom certificates to the NiFi truststore to allow NiFi to trust third party services.

There are two methods for adding certificates to NiFi's truststore: in-line in the custom resource or through a Secret/ConfigMap. For multiple certificates, it is recommended to provide them via Secret/ConfigMap to maintain readability of the NiFi custom resource.

### In-line

```
spec:
  security:
    additionalCABundles: [***BASE64 ENCODED CERT CHAIN***]
```

### Secret/ConfigMap

First create a Secret with the needed Certificates. The referenced files may have multiple certificates in them.

```
kubectl create secret generic nifi-additional-cas --from-file=cert1.crt=
[***A CERTIFICATE FILE***] --from-file=cert2.crt=[***ANOTHER CERTIFICATE
FILE***]
```

Then supply the Secret/ConfigMap name to the following spec:

```
spec:
  security:
    additionalCABundlesRef:
      name: nifi-additional-cas
      kind: Secret
```

## Securing communication between NiFi clusters using the Default Node SSL Context Service

The Cloudera Flow Management Operator for Kubernetes automatically provisions a DefaultNodeSSLContextService controller service in the root process group of each NiFi cluster it manages. This controller service is an instance of StandardSSLContextService that references the node keystore and truststore generated by the operator, making TLS credentials available to data flows without requiring manual SSL configuration.



## About this task

The operator creates the `DefaultNodeSSLContextService` on initial deployment and checks it on every subsequent reconciliation.

- If the service is missing, the operator recreates it.
- If the service is present, the operator validates its configuration.

The service is configured with the following values:

**Table 1: Service configuration reference**

| Property                         | Value                                                                                |
|----------------------------------|--------------------------------------------------------------------------------------|
| Service type                     | <code>StandardSSLContextService</code>                                               |
| Keystore path                    | <code>/opt/nifi/nifi-current/data/keystore.jks</code>                                |
| Truststore path                  | <code>/opt/nifi/nifi-current/data/truststore.jks</code>                              |
| Keystore and truststore password | Retrieved from the <code>&lt;nifi-name&gt;-keystorepassword</code> Kubernetes Secret |
| Default keystore type            | JKS                                                                                  |
| TLS protocol                     | TLS                                                                                  |



**Note:** On FIPS-enabled clusters where a custom keystore type such as BCFKS is configured using the `KEYSTORE_TYPE` environment variable, the `DefaultNodeSSLContextService` uses that type instead of the default JKS. For more information, see *Configuring JVM security providers (FIPS)*.

You can reference the `DefaultNodeSSLContextService` in any processor that requires an SSL Context Service. For example, you can use it to secure communication between two NiFi clusters managed by the same operator instance, using an `InvokeHTTP` processor on the sending cluster and a `ListenHTTP` processor on the receiving cluster.

## Before you begin

Ensure that:

- Both NiFi clusters are managed by the same Cloudera Flow Management Operator for Kubernetes instance.
- The node certificates for both clusters are signed by the same root certificate authority, establishing a shared chain of trust.
- The Kubernetes service for the receiving cluster exposes the port used by the `ListenHTTP` processor.

## Procedure

1. On the receiving cluster, add a `ListenHTTP` processor to the flow and configure the following properties.

### Listening Port

Enter the port on which the processor listens for incoming requests.

For example: 9999

### Base Path

Enter the base path for the listener endpoint.

For example: `contentListener`

### SSL Context Service

Select `DefaultNodeSSLContextService`.

2. On the sending cluster, add an `InvokeHTTP` processor to the flow and configure the following properties.

### HTTP URL

Enter the internal Kubernetes Service URL of the receiving cluster, using the following format:

```
https://[***NIFI-NAME***].[***NAMESPACE***].svc.cluster.local:[***PORT***]/[***BASEPATH***]
```

Replace [\*\*\*NIFI-NAME\*\*\*], [\*\*\*NAMESPACE\*\*\*], [\*\*\*PORT\*\*\*], and [\*\*\*BASEPATH\*\*\*] with the actual values for the receiving cluster.

### SSL Context Service

Select DefaultNodeSSLContextService.

3. Start both processors.
4. Send a FlowFile through the InvokeHTTP processor and verify that the ListenHTTP processor on the receiving cluster receives the request.

## Configuring NiFi properties

Learn how to override default NiFi configuration settings provided by Cloudera Flow Management Operator for Kubernetes from the CR file.

NiFi settings are available as part of the specification, under the configOverride key. They can be provided in one of the following ways:

- inline,
- as a ConfigMap
- as a Secret

```
spec:
  configOverride:
    nifiProperties:
      upsert:
        nifi.cluster.load.balance.connections.per.node: "1"
        nifi.cluster.load.balance.max.thread.count: "4"
        nifi.cluster.node.connection.timeout: "60 secs"
        nifi.cluster.node.read.timeout: "60 secs"
    bootstrapConf:
      upsert:
        java.arg.2: -Xms2g
        java.arg.3: -Xmx2g
        java.arg.13: -XX:+UseConcMarkSweepGC
```

## Overriding NiFi settings using ConfigMaps and Secrets

Learn about overriding default NiFi settings using ConfigMaps and Secrets.

The ConfigMap or Secret values are available to inject into the environment for the following files:

- authorizers.xml
- bootstrap.conf
- logback.xml
- login-identity-providers.xml
- nifi.properties
- state-management.xml

Each of these config overrides must be in an individual ConfigMap with the key being the filename to be replaced. Using this ConfigMap or Secret reference method entirely overrides the defaults provided by the Cloudera Flow Management Operator for Kubernetes, which may impact cluster operation.

NiFiSpec

```
spec:
  configOverride:
    authorizersObjectReference:
      kind: "ConfigMap"
      name: "custom-authorizers"

ConfigMapSpec
data:
  authorizers.xml: |
    <authorizers>
      <authorizer>
        <identifier>single-user-authorizer</identifier>
        <class>org.apache.nifi.authorization.single.user.SingleUserAuth
orizer</class>
      </authorizer>
    </authorizers>
```

## Configuring scaling

Learn about scaling NiFi clusters either manually or automatically, using HPA.

It is possible to manually scale up and down the NiFi cluster size by editing the replicas value in the deployment file and applying the changes. It is also possible to specify an HPA to automatically scale the NiFi cluster (replica count) based on the Kubernetes resources (CPU/memory).

To manually scale the cluster, simply edit the replicas field to your desired replica count.

For autoscaling, apply a Horizontal Pod Autoscaling (HPA) resource targeting the NiFi CR, as follows:

```
apiVersion: autoscaling/v2
kind: HorizontalPodAutoscaler
metadata:
  name: nifi-hpa
spec:
  maxReplicas: 3
  minReplicas: 1
  metrics:
    - type: Resource
      resource:
        name: cpu
        target:
          type: Utilization
          averageUtilization: 75
  scaleTargetRef:
    apiVersion: cfm.cloudera.com/v1alpha1
    kind: Nifi
    name: [***NIFI CLUSTER NAME***]
```

## Security context configuration for Nifi

Cloudera Flow Management Operator for Kubernetes sets hardened security contexts on all pods and containers by default. Three StatefulSetSpec fields let you override pod, main container, and tini container defaults on the Nifi CRD without replacing the operator defaults wholesale.

### API fields

| Field              | Scope                | API path                            |
|--------------------|----------------------|-------------------------------------|
| podSecurityContext | Pod (all containers) | spec.statefulset.podSecurityContext |

| Field                    | Scope                                            | API path                                  |
|--------------------------|--------------------------------------------------|-------------------------------------------|
| containerSecurityContext | Main application container only (nifi)           | spec.statefulset.containerSecurityContext |
| tiniSecurityContext      | Log-tailer sidecars and kerberize init container | spec.statefulset.tiniSecurityContext      |

### Merge semantics

Each field is applied with Kubernetes strategic-merge-patch semantics (the same algorithm used by `kubectl apply`):

- A field that is set in the override takes precedence over the operator default.
- A field that is not set in the override keeps the operator default.
- Unsetting a defaulted field is not supported — omitting a field in the override leaves the operator value in place; you cannot remove a field the operator always sets.
- List fields (capabilities.add, capabilities.drop) behave atomically: when the override supplies the list, the override list replaces the operator list entirely.

### Operator defaults

#### Pod Security Standards

The default security context settings of Cloudera Flow Management Operator for Kubernetes and NiFi pods are compliant with the restricted [Pod Security Standard](#).

#### Pod security context

Applied to every pod in the StatefulSet.

```
runAsNonRoot: true
seccompProfile:
  type: RuntimeDefault
fsGroup: 0          # standard Kubernetes clusters only; omitted on OpenShift
```

On OpenShift clusters, fsGroup is not set, letting the platform assign the supplemental group automatically. On standard clusters it defaults to 0 (root group) so that volumes are accessible to the NiFi process. Use podSecurityContext.fsGroup to change it.

#### Main container security context

Applied to the nifi container.

```
allowPrivilegeEscalation: false
capabilities:
  drop: [ALL]
```

runAsNonRoot is inherited from the pod security context.

#### Tini container security context

Applied to the log-tailer sidecar containers and the kerberize init container (when Kerberos is enabled).

```
runAsNonRoot: true
allowPrivilegeEscalation: false
capabilities:
  drop: [ALL]
```

NiFi log-tailers: app-log, user-log, bootstrap-log, gc-log, request-log, and deprecation-log.

## Examples

Override fsGroup so that the kubelet does not change ownership of shared volumes to root:root, and set a specific UID for the main container and tini containers:

```
spec:
  statefulset:
    podSecurityContext:
      fsGroup: 1000
    containerSecurityContext:
      runAsUser: 1000
    tiniSecurityContext:
      runAsUser: 1000
```

A sample patch is provided at `config/samples/nifi/patches/pod-security-context.yaml`.

## Configuring pod affinity

Pod affinity controls where pods are deployed based on node configuration and placement of other pods.

To learn more about Pod Affinity, read *Assigning Pods to Nodes* in the Kubernetes documentation.

You can configure the affinity settings of the NiFi pod in the NiFi Custom Resource under `spec.statefulset`. The following example represents the default configuration which will be added to the Custom Resource in the defaulting webhook.



### Note:

If any affinity is provided in `spec.statefulset`, the default in the example will not be applied.

```
spec:
  statefulset:
    affinity:
      podAntiAffinity:
        preferredDuringSchedulingIgnoredDuringExecution:
          - podAffinityTerm:
              labelSelector:
                matchExpressions:
                  - key: app.kubernetes.io/instance
                    operator: In
                    values:
                      - mynifi
              topologyKey: kubernetes.io/hostname
            weight: 1
```

This default configuration attempts to spread NiFi cluster pods to different nodes of the Kubernetes cluster. If there are more NiFi pods than available Kubernetes nodes, then some pods will coexist on the same node.

### Related Information

[Assigning Pods to Nodes](#) | [Kubernetes documentation](#)

## Configuring connections to NiFi

Learn about configuring connections for your NiFi cluster.

Cloudera Flow Management Operator for Kubernetes provides a flexible method of configuring connections to NiFi called Connections. Using Connections, a Service, Ingress, or Route can be configured to route to a specific port on NiFi. For defining Connections targeting an arbitrary port on NiFi, use the `spec.connections` array. For configuring connection to the NiFi Web UI, use the `spec.uiConnection` field. This documentation provides a full reference for Connections.

## Configuring session affinity

Learn about configuring session affinity. It makes possible to keep connection to the web UI alive in clusters with several nodes.

Regardless of your connection type, a NiFi cluster with more than one node requires session affinity of some type for the Web UI to operate. This is because each NiFi node can supply its own web UI and if a LoadBalancer shifts you to another instance, your authentication tokens become invalid. The best method of applying session affinity varies greatly depending on the Kubernetes cluster provider. In the simplest case, defining session affinity on the web Service resource itself is sufficient:

```
spec:
  uiConnection:
    serviceConfig:
      sessionAffinity: ClientIP
```

In certain clouds, for example AWS, the backing LoadBalancer resources do not support session affinity, and cause provisioning to break.

## Configuring arbitrary connections

Learn about configuring a connections array.

You can use the connections array to flexibly define routing to ports on NiFi. The below example configures an Ingress resource with some annotations and labels provided. The Ingress will expose a URL <https://nifi.io/listenTCP> which routes to port 9432 on NiFi. Additionally, the backing Service is configured to have two extra ports, 8496 and 8495.

```
spec:
  connections:
  - type: Ingress
    name: someConnection
    annotations:
      someanno: myanno
    labels:
      somelabel: mylabel
    ingressConfig:
      hostname: nifi.io
      paths:
      - port: 9432
        path: /listenTCP
        name: listentcp
    serviceConfig:
      ports:
      - port: 8496
        protocol: TCP
        name: porta
      - port: 8495
        protocol: UDP
        name: portb
```

## Configuring NiFi Web UI connection

Learn about configuring a connection to the NiFi web UI.

You can configure a connection to the NiFi Web UI using the `spec.uiConnection` field. It is a standard connection field with special validation and handling. The name of this connection is always ignored and set to `[***CR NAME***]-web`. For Ingress type connections, a maximum of one path may be specified. When you configure a `uiConnection`, the `spec.hostname` field is required.

The uiConnection can support hostname routing with and without an additional context path. It is not recommended to use a context path for routing as NiFi does not support it well, but it is possible. For more information, see NiFi documentation on proxy configuration. An example using ingress-nginx is included in this section.

### Related Information

[NiFi proxy configuration](#)

### Hostname-only ingress example

Learn about configuring an Ingress resource using TLS files generated by Cloudera Flow Management Operator for Kubernetes.

This YAML snippet configures an Ingress resource for accessing the NiFi Web UI. It uses the TLS files generated by a Cloudera Flow Management Operator for Kubernetes created Certificate as defined in spec.security.ingressCertGen. The supplied annotations are for the ingress-nginx Ingress controller. The affinity settings enable a persistent session so that UI interactions go to the same NiFi node in the cluster. The backend-protocol setting is needed for when NiFi is configured to be secure, as it will reject any non-HTTPS connection attempts.

```
spec:
  uiConnection:
    type: Ingress
    ingressConfig:
      ingressClassName: myIngressClass
      ingressTLS:
        - hosts:
            - nifi.localhost
          secretName: mynifi-ingress-cert
    annotations:
      nginx.ingress.kubernetes.io/affinity: cookie
      nginx.ingress.kubernetes.io/affinity-mode: persistent
      nginx.ingress.kubernetes.io/backend-protocol: HTTPS
```

### Hostname-only route example

Learn about configuring a Route resource to access the NiFi web UI.

This YAML snippet configures a Route resource for accessing the NiFi web UI.

```
spec:
  uiConnection:
    type: Route
    routeConfig:
      tls:
        termination: passthrough
```

### Ingress with context path example

Learn about configuring an Ingress resource that rewrites the connection path in incoming requests and does a reverse-rewrite on UI calls going to the backend.

This YAML code snippet configures an ingress UI Connection with a path. The annotations here are for the ingress-nginx ingress controller and all are required for NiFi to correctly understand the incoming requests.

In the example the path includes some regex at the end: (/|\$)(.\*). This regex informs the rewrite directives in the configuration-snippet and rewrite-target annotations. NiFi does not handle proxy paths well, it does not understand that https://nifi.localhost/some/path/to/nifi coming through the defined Ingress is intended to call the /nifi API to load the UI. The rewrite-target annotation addresses this by capturing the /nifi and anything that comes after and sends that as the path to the NiFi pod. It translates /some/path/to/nifi/ to /nifi/. Similarly, the NiFi web UI does not correctly form API calls going to the backend, attempting to call /nifi/ instead of /some/path/to/nifi/. This is addressed by the configuration-snippet rewrite instruction. It does the reverse of the rewrite-target, reapplying the removed context

path /some/path/to. The remaining configuration-snippet lines are headers required by a NiFi behind a proxy. For more information, see the *NiFi System Administrator's Guide*.

```
spec:
  uiConnection:
    type: Ingress
    ingressConfig:
      ingressClassName: myIngressClass
      ingressTLS:
        - hosts:
            - nifi.localhost
          secretName: mynifi-ingress-cert
      paths:
        - port: 8443
          path: "/some/path/to(/|$)(.*)"
    annotations:
      nginx.ingress.kubernetes.io/affinity: cookie
      nginx.ingress.kubernetes.io/affinity-mode: persistent
      nginx.ingress.kubernetes.io/backend-protocol: HTTPS
      nginx.ingress.kubernetes.io/configuration-snippet: |-
        proxy_set_header X-ProxyScheme $scheme;
        proxy_set_header X-ProxyHost $host;
        proxy_set_header X-ProxyPort $server_port;
        proxy_set_header X-ProxyContextPath /some/path/to;
        rewrite (.*\nifi)$ $1/ redirect;
        proxy_ssl_name mynifi.default.svc.cluster.local;
      nginx.ingress.kubernetes.io/rewrite-target: /$2
```

## Configuring additional proxy hosts

Learn about adding a list of expected proxy hosts. NiFi will reject API requests sent through proxies if it is not aware of those proxy hosts.

Provide a list of expected proxy hosts to NiFi beyond the hostname provided in `hostName`. To add additional proxy hosts, add the following to your NiFi YAML:

```
spec:
  additionalProxyHosts:
    - [***YOUR PROXY***)
    - [***ANOTHER PROXY***)
```

## Configuring authentication and authorization for NiFi

Learn about configuring the type of authentication appropriate for your use case.



### Note:

NiFi requires all web and API traffic be over HTTPS to support user authentication and authorization. For information on adding an auto-generated certificate to each node, see [Node certificate generation](#).



### Note:

NiFi deployed by the Cloudera Flow Management Operator for Kubernetes is secure by default. If you do not provide an authorization method, NiFi is configured in single user auth mode with username "demonstration-user" and password "demonstration-password". If you do not specify Node Certificate Generation, node certificates are generated using a self-signed Cluster Issuer `nifi-node-certs` that is deployed with the operator.

## Configuring the initial admin user

When you set up a secured NiFi instance for the first time, you must manually designate an "Initial Admin Identity". This initial admin user is granted access to the UI and given the ability to create additional users, groups, and policies.



NiFi requires an initial admin user which will be given sufficient privileges to configure other users and policies. When configuring an authentication method other than single user authentication, an initial admin user is required.

Specify the initial admin user with the following YAML snippet:

```
spec:
  security:
    initialAdminIdentity: [***INITIAL ADMIN IDENTITY***]
```

Replace `[***INITIAL ADMIN IDENTITY***]` with a username, LDAP distinguished name (DN), or a Kerberos principal.

### Related Concepts

[Configuring single user authentication](#)

[Configuring a user identity with access policies](#)

[Configuring user groups](#)

[Configuring NiFi access policy profiles](#)

[Configuring LDAP authentication](#)

[Configuring OIDC authentication](#)

[Configuring Ranger authorization for NiFi](#)

## Configuring single user authentication

Single user authentication is NiFi's most basic authentication option, sufficient for individual development clusters and also production clusters where flows are deployed in a controlled manner, such as continuous integration (CI) or site reliability engineering (SRE). A single user is granted all permissions on the NiFi cluster, no other users can be configured.

- Configuration snippet for letting NiFi generate the password.

```
spec:
  security:
    singleUserAuth:
      enabled: true
```

You find the generated username and password in the app-log container logs.

- Configuration snippet for setting NiFi username and password using a Secret:

```
spec:
  security:
    singleUserAuth:
      enabled: true
      credentialsSecretName: [***YOUR CREDENTIALS SECRET***]
```

Replace:

`[***YOUR CREDENTIALS SECRET***]`

Create your credentials secret with the following command:

```
kubectl create secret generic [***YOUR CREDENTIALS SECRET***] --
from-literal=username=[***YOUR USER NAME***] --from-literal=password=[***YOUR PASSWORD***]
```

Replace:

`[***YOUR CREDENTIALS SECRET***]`

with the desired credentials secret name

`[***YOUR USER NAME***]`

with the generated username in the app-log container logs

**[\*\*\*YOUR PASSWORD\*\*\*]**

with the generated password in the app-log container logs

### Related Concepts

[Configuring the initial admin user](#)

[Configuring a user identity with access policies](#)

[Configuring user groups](#)

[Configuring NiFi access policy profiles](#)

[Configuring LDAP authentication](#)

[Configuring OIDC authentication](#)

[Configuring Ranger authorization for NiFi](#)

## Configuring a user identity with access policies

The User custom resource creates and manages a user in a NiFi instance. Cloudera Flow Management Operator for Kubernetes reconciles the user in the NiFi API and optionally provisions a client certificate and access policies for that user.

Each User resource targets a single Nifi instance. When created, the operator registers the user in the NiFi API and writes the NiFi-assigned UUID back to status.identifier, which indicates the User was created successfully. Future reconciliations will update user identity, if it is changed, and keep access policies in sync. When the resource is deleted, the operator removes the user from NiFi and from all of its access policies before releasing the finalizer.

### Spec fields

| Field                  | Type              | Required | Description                                                             |
|------------------------|-------------------|----------|-------------------------------------------------------------------------|
| instanceTarget         | InstanceTarget    | Yes      | The Nifi instance to manage the user in                                 |
| identity               | string            | Yes      | The username displayed in NiFi and used as the certificate CN / DNS SAN |
| certificate            | CertificateSpec   | No       | Controls client certificate generation; omit to disable                 |
| accessPolicies         | []AccessPolicy    | No       | Inline access policies granted to this user                             |
| accessPolicyProfileRef | []ObjectReference | No       | References to AccessPolicyProfile resources                             |

### instanceTarget

| Field     | Type   | Required | Description                                               |
|-----------|--------|----------|-----------------------------------------------------------|
| kind      | Nifi   | Yes      | Kind of the target resource                               |
| name      | string | Yes      | Name of the target resource                               |
| namespace | string | No       | Namespace of the target; defaults to the User's namespace |

### accessPolicies

Each entry specifies one or more NiFi resource paths and the actions to grant on them.

| Field     | Type     | Description                                                      |
|-----------|----------|------------------------------------------------------------------|
| resources | []string | NiFi resource paths, e.g. /process-groups/root, /flow, /policies |

| Field   | Type     | Description          |
|---------|----------|----------------------|
| actions | []string | read, write, or both |

Policies specified inline in accessPolicies take precedence over any conflicting policy for the same resource inherited from an AccessPolicyProfile.

## Examples

### Minimal user

```
apiVersion: cfm.cloudera.com/v1alpha1
kind: User
metadata:
  name: alice
spec:
  instanceTarget:
    kind: Nifi
    name: my-nifi
    identity: alice
```

### User with read access to the root process group

```
apiVersion: cfm.cloudera.com/v1alpha1
kind: User
metadata:
  name: alice
spec:
  instanceTarget:
    kind: Nifi
    name: my-nifi
    identity: alice
  accessPolicies:
    - resources:
        - /process-groups/root
        - /flow
      actions:
        - read
```

### User with a generated client certificate

Requires the target NiFi to have spec.security.nodeCertGen configured. The operator creates a cert-manager Certificate in the same namespace, using spec.identity as both the CommonName and DNS SAN. The resulting Secret will contain a valid client certificate that can be used to authenticate as the user to the NiFi API.

```
apiVersion: cfm.cloudera.com/v1alpha1
kind: User
metadata:
  name: alice
spec:
  instanceTarget:
    kind: Nifi
    name: my-nifi
    identity: alice
  certificate:
    generate: true
```

### User with policies from an AccessPolicyProfile

```
apiVersion: cfm.cloudera.com/v1alpha1
kind: User
```

```
metadata:
  name: alice
spec:
  instanceTarget:
    kind: Nifi
    name: my-nifi
  identity: alice
  accessPolicyProfileRef:
    - name: read-only-profile
      namespace: default
```

## Lifecycle

### Creation

The operator registers the user in the NiFi API. The NiFi-assigned UUID is written to `status.identifier` and used by the operator when making additional changes.

### Identity Rename

Updating `spec.identity` renames the user in the NiFi API on the next reconciliation. The `status.identifier` is preserved, so NiFi-internal references (policy memberships, group memberships) are not affected.

### Access policy changes

On each reconciliation the operator computes the desired policy set by merging inline access policies with any referenced profiles and diffs it against the current NiFi state. Missing policies are added and stale policies are removed. Removing a policy only removes the user from that policy; the policy itself is never deleted.

### Deletion

Deleting the User

resource removes the user from the NiFi API and from all access policies, then deletes the generated client certificate (if any) before releasing the finalizer. If the target Nifi resource is not found at deletion time, the NiFi-side cleanup is skipped and the finalizer is still removed.

## Relationship to UserGroup

Users can be collected into UserGroup resources by direct reference (`spec.userRefs`). This will assign those users to the relevant user group in NiFi.

### Related Concepts

[Configuring the initial admin user](#)

[Configuring single user authentication](#)

[Configuring LDAP authentication](#)

[Configuring OIDC authentication](#)

[Configuring Ranger authorization for NiFi](#)

[Configuring NiFi access policy profiles](#)

[Configuring user groups](#)

## Configuring user groups

The UserGroup custom resource creates and manages a user group in a NiFi instance. Cloudera Flow Management Operator for Kubernetes reconciles the group in the NiFi API, keeps its membership in sync with referenced User resources, and optionally provisions access policies for the group.

Each UserGroup resource targets a single Nifi instance. When created, the operator registers the group in the NiFi API and writes the NiFi-assigned UUID to `status.identifier`. Subsequent reconciliations detect identity renames, sync

group membership from `spec.userRefs`, and diff access policies against the current NiFi state. When the resource is deleted, the operator removes the group from NiFi before releasing the finalizer.

### Spec Fields

| Field                       | Type                           | Required | Description                                                        |
|-----------------------------|--------------------------------|----------|--------------------------------------------------------------------|
| <code>instanceTarget</code> | <code>InstanceTarget</code>    | Yes      | The NiFi instance to manage the group in                           |
| <code>identity</code>       | <code>string</code>            | Yes      | The group name displayed in NiFi                                   |
| <code>userRefs</code>       | <code>[]ObjectReference</code> | No       | References to User resources whose members are added to this group |
| <code>accessPolicies</code> | <code>[]AccessPolicy</code>    | No       | Inline access policies granted to this group                       |

### instanceTarget

| Field                  | Type                | Required | Description                                                    |
|------------------------|---------------------|----------|----------------------------------------------------------------|
| <code>kind</code>      | <code>Nifi</code>   | Yes      | Kind of the target resource                                    |
| <code>name</code>      | <code>string</code> | Yes      | Name of the target resource                                    |
| <code>namespace</code> | <code>string</code> | No       | Namespace of the target; defaults to the UserGroup's namespace |

### userRefs

Each entry is an object reference to a User resource in the same cluster. The namespace defaults to the UserGroup's own namespace if omitted.

```
userRefs:
  - name: alice
  - name: bobnamespace: other-ns
```

Users that have not yet been reconciled by the User controller (no `status.identifier`) are silently skipped and will be added on the next reconciliation once their identifier is available.

### accessPolicies

Each entry specifies one or more NiFi resource paths and the actions to grant on them.

| Field                  | Type                  | Description                                                                                               |
|------------------------|-----------------------|-----------------------------------------------------------------------------------------------------------|
| <code>resources</code> | <code>[]string</code> | NiFi resource paths, e.g. <code>/process-groups/root</code> , <code>/flow</code> , <code>/policies</code> |
| <code>actions</code>   | <code>[]string</code> | read, write, or both                                                                                      |

The special token `/process-groups/root` is resolved at runtime to the root process group UUID of the target NiFi instance.

### Status Fields

| Field                   | Type                | Description                                                                  |
|-------------------------|---------------------|------------------------------------------------------------------------------|
| <code>identifier</code> | <code>string</code> | NiFi-assigned UUID for this group; set after first successful reconciliation |

## Examples

### Minimal group

A minimal UserGroup with no users or access policies assigned to it.

```

apiVersion: cfm.cloudera.com/v1alpha1
kind: UserGroup
metadata:
  name: operators
spec:
  instanceTarget:
    kind: Nifi
    name: my-nifi
  identity: operators

```

### Group with members

This group contains users represented by Users alice and bob, both User custom resources in the same namespace as this operators UserGroup. Users alice and bob will be added to the operators UserGroup. This UserGroup does not have any access policies associated with it.

```

apiVersion: cfm.cloudera.com/v1alpha1
kind: UserGroup
metadata:
  name: operators
spec:
  instanceTarget:
    kind: Nifi
    name: my-nifi
  identity: operators
  userRefs:
    - name: alice
    - name: bob

```

### Group with read access to the root process group

This group contains a single User alice and has read access to the root process group and the UI.

```

apiVersion: cfm.cloudera.com/v1alpha1
kind: UserGroup
metadata:
  name: readers
spec:
  instanceTarget:
    kind: Nifi
    name: my-nifi
  identity: readers
  userRefs:
    - name: alice
  accessPolicies:
    - resources:
      - /process-groups/root
      - /flowactions:
      - read

```

## Lifecycle

### Creation

The operator registers the group in the NiFi API on the first successful reconciliation. The NiFi-assigned UUID is written to status.identifier and used to identify the group in all subsequent reconciliations.

**Identity rename**

Updating `spec.identity` renames the group in the NiFi API on the next reconciliation. The `status.identifier` is preserved, so NiFi-internal references (policy memberships) are not affected.

**Membership changes**

On each reconciliation the operator resolves `spec.userRefs` to NiFi tenant UUIDs via `user.status.identifier` and calls the NiFi API to set the group's member list. Removing a User from `spec.userRefs` removes them from the group on the next reconciliation.

**Access policy changes**

On each reconciliation the operator diffs the desired policy set against the current NiFi state. Missing policies are added and stale policies are removed. Removing a policy only removes the group from that policy; the policy object itself is never deleted from NiFi.

**Deletion**

Deleting the UserGroup resource removes the group from the NiFi API before releasing the finalizer. If the target NiFi resource is not found at deletion time, the NiFi-side cleanup is skipped and the finalizer is still removed.

**Relationship to User**

User resources are added to a group by listing them in `spec.userRefs`. The operator watches User resources and reconciles any UserGroup that references them, so membership updates propagate automatically when a User is created or updated.

**Related Concepts**

[Configuring the initial admin user](#)

[Configuring single user authentication](#)

[Configuring NiFi access policy profiles](#)

[Configuring LDAP authentication](#)

[Configuring OIDC authentication](#)

[Configuring Ranger authorization for NiFi](#)

[Configuring a user identity with access policies](#)

**Configuring NiFi access policy profiles**

An `AccessPolicyProfile` is a reusable collection of NiFi access policies that can be shared across multiple User resources. Instead of repeating the same policy list on every user, define the policies once in an `AccessPolicyProfile` and reference it from each user.

**Defining an AccessPolicyProfile**

```
apiVersion: cfm.cloudera.com/v1alpha1
kind: AccessPolicyProfile
metadata:
  name: read-only-profile
  namespace: nifi
spec:
  accessPolicies:
    - actions:
        - "read"
      resources:
        - /flow
        - /process-groups/root
    - actions:
        - "read"
      resources:
```

```
- /data/process-groups/root
```

Each entry in `accessPolicies` pairs one or more actions (read, write) with one or more NiFi resource paths.

### Referencing a profile from a User

Use `accessPolicyProfileRef` on the User to attach one or more profiles:

```
apiVersion: cfm.cloudera.com/v1alpha1
kind: User
metadata:
  name: alice
  namespace: nifi
spec:
  instanceTarget:
    name: my-nifi
    namespace: nifi
  identity: alice@example.com
  accessPolicyProfileRef:
    - name: read-only-profile
      namespace: nifi
```

Multiple profiles can be listed and their policies are unioned together:

```
accessPolicyProfileRef:
  - name: read-only-profile
    namespace: nifi
  - name: component-admin-profile
    namespace: nifi
```

The namespace field in each entry defaults to the User resource's own namespace when omitted.

### Inline policies override profile policies

Policies defined directly in `spec.accessPolicies` take precedence over policies from referenced profiles. When the same NiFi resource appears in both places, the inline policy wins and the policy from the `AccessPolicyProfile` for that resource is ignored entirely.

```
spec:
  accessPolicies:
    # This write policy for /flow overrides any /flow policy from
    # the profile.
    - actions: ["read", "write"]
      resources:
        - /flow
  accessPolicyProfileRef:
    # has read-only /flow — ignored for /flow
    - name: read-only-profile
      namespace: nifi
```

The override is per-resource: other resources in the profile that are not covered by inline policies are still applied.

### Reconciliation behaviour

The user controller watches `AccessPolicyProfile` resources. When a profile is updated, every User that references it is automatically re-reconciled, so NiFi access policies are kept in sync without manual intervention.

All referenced `AccessPolicyProfile` resources must exist before the User is reconciled. If any profile is not found, the reconciliation fails without updating the NiFi User and is retried until all profiles are available.



**Related Concepts**

- [Configuring the initial admin user](#)
- [Configuring single user authentication](#)
- [Configuring user groups](#)
- [Configuring LDAP authentication](#)
- [Configuring OIDC authentication](#)
- [Configuring Ranger authorization for NiFi](#)
- [Configuring a user identity with access policies](#)

**Configuring LDAP authentication**

Learn how to configure an LDAP server for user authentication in your NiFi or NiFi Registry cluster.

Cloudera Flow Management Operator for Kubernetes can configure NiFi to connect to an LDAP server for user authentication.

Prerequisites:

- Full LDAP URL, i.e. `ldap://[***LDAP SERVER URL***]:[***LDAP PORT***]`
- Desired authentication strategy
- Authentication credentials and key/trust stores if using LDAPS.
- User search filters

For LDAP servers protected with any authentication, a Secret must be created containing the correct authentication credentials and TLS resources (if applicable). The Secret must contain the following data fields:

- `managerPassword`
- `keystore` (if TLS is configured)
- `keystorePassword` (if TLS is configured)
- `truststore` (if TLS is configured)
- `truststorePassword` (if TLS is configured)

Create the secret using the `kubectl` CLI utility:

```
kubectl create secret generic my-ldap-creds \
  --from-literal=managerPassword=myManagerPassword \
  --from-file=keystore=/path/to/keystore \
  --from-literal=keystorePassword=myKeystorePassword \
  --from-file=truststore=/path/to/truststore \
  --from-literal=truststorePassword=myTruststorePassword
```

The following example shows a connection to an LDAP server protected with basic authentication with TLS.

```
spec:
  security:
    initialAdminIdentity: mynifiadmin
    ldap:
      authenticationStrategy: SIMPLE
      managerDN: "cn=admin,dc=example,dc=org"
      secretName: my-ldap-creds
      referralStrategy: FOLLOW
      connectTimeout: 3 secs
      readTimeout: 10 secs
      url: ldap://my-ldap-url:389
      userSearchBase: "dc=example,dc=org"
      userSearchFilter: "(uid={0})"
      identityStrategy: USE_USERNAME
      authenticationExpiration: 12 hours
  tls:
    keystoreType: jks
```

```
truststoreType: jks
clientAuth: NONE
protocol: TLSv1.2
```

By default, Cloudera Flow Management Operator for Kubernetes does not deploy a UserGroupProvider using the LDAP target. This means NiFi does not pull down any users, only queries the LDAP server for authentication. This impedes configuring user access, requiring the NiFi administrator to create each user manually.

The following example shows configuring user synchronization with the LDAP server:

```
spec:
  security:
    ldap:
      sync:
        interval: 30 min
        userObjectClass: inetOrgPerson
        userSearchScope: SUBTREE
        userIdentityAttribute: cn
        userGroupNameAttribute: ou
        userGroupNameReferencedGroupAttribute: ou
        groupSearchBase: "dc=example,dc=org"
        groupObjectClass: organizationalUnit
        groupSearchScope: OBJECT
        groupNameAttribute: ou
```

### Related Concepts

- [Configuring the initial admin user](#)
- [Configuring single user authentication](#)
- [Configuring a user identity with access policies](#)
- [Configuring user groups](#)
- [Configuring NiFi access policy profiles](#)
- [Configuring OIDC authentication](#)
- [Configuring Ranger authorization for NiFi](#)

## Configuring OIDC authentication

NiFi supports user authentication with Open ID Connect (OIDC) providers such as Keycloak.

To configure authentication with an Open ID Connect (OIDC) provider, you need to know the Discovery URL, clientId, and clientSecret of the authenticating server.

An example of a Discovery URL from Keycloak is:

```
https://keycloak.cfmoperator.net/realms/master/.well-known/openid-configuration
```

The clientId and clientSecret fields are provided to NiFi in a Kubernetes secret. Create that secret with the following command:

```
kubectl create secret generic oidc-client-secret --from-literal=clientId=[***YOUR CLIENT ID***] --from-literal=clientSecret=[***YOUR CLIENT SECRET***]
```

The Discovery URL and client credentials secret are provided to NiFi with the below spec:

```
spec:
  security:
    openIDAuth:
      discoveryURL: [***YOUR DISCOVERY URL***]
      clientSecretName: [***OIDC CLIENT SECRET***]
```

OpenIDAuth also provides additional options:

**connectTimeout**

Specify the connection timeout when communicating with the OIDC provider.

**readTimeout**

Specify the read timeout when communicating with the OIDC provider.

**JWSAlgorithm**

JWSAlgorithm is the preferred algorithm for validating identity tokens. If this value is blank, it defaults to RS256 which is required to be supported by the OIDC provider according to the specification. If this value is HS256, HS384, or HS512, NiFi attempts to validate HMAC protected tokens using the specified client secret. If this value is none, NiFi attempts to validate unsecured/plain tokens. Other values for this algorithm attempt to parse as an RSA or EC algorithm to be used in conjunction with the JSON Web Key (JWK) provided through the jwks\_uri in the metadata found at the discovery URL.

**Note:**

For NiFi to trust the certificate presented by the OIDC server, you must add a valid CA for your OIDC server to NiFi. For information on adding a CA to NiFi, see [Additional CA Bundles](#).

**Related Concepts**

[Configuring the initial admin user](#)

[Configuring single user authentication](#)

[Configuring a user identity with access policies](#)

[Configuring user groups](#)

[Configuring NiFi access policy profiles](#)

[Configuring LDAP authentication](#)

[Configuring Ranger authorization for NiFi](#)

**Related Information**

[OpenID Connect | Apache NiFi System Administrator's Guide](#)

## Configuring Ranger authorization for NiFi

Delegate NiFi authorization to Apache Ranger so Cloudera Flow Management Operator for Kubernetes renders Ranger authorizer configuration instead of file-based access policies.

By default, NiFi manages access policies in a local `authorizations.xml` file through `FileAccessPolicyProvider` backed by `StandardManagedAuthorizer`. Cloudera Flow Management Operator for Kubernetes drives that model through its `User` and `UserGroup` controllers, which call the NiFi REST API to keep policy assignments in sync with the desired state declared in Kubernetes. If you run Cloudera on cloud, Cloudera on premises, or any deployment where Apache Ranger is the central policy engine, that file-based model is redundant and conflicts with Ranger.

When the `ranger` field is set on a NiFi resource's `spec.security`, Cloudera Flow Management Operator for Kubernetes switches to a fully Ranger-delegated authorization mode:

- `authorizers.xml` is rendered with `RangerNiFiAuthorizer` instead of `FileAccessPolicyProvider` and `StandardManagedAuthorizer`.
- `nifi.properties` sets `nifi.security.user.authorizer=ranger-nifi-authorizer`.
- Two XML configuration files consumed by the Ranger NiFi plugin, `ranger-nifi-security.xml` and `ranger-nifi-audit.xml`, are generated and mounted into the NiFi pods at `/opt/nifi/nifi-current/conf/`.
- The `User` and `UserGroup` controllers still create NiFi tenants (users and groups) so that Ranger-issued identities are recognized by NiFi, but they skip access policy management entirely.

**Note:**

The Apache Ranger NiFi plugin documentation is the authoritative reference for the underlying XML property names. For more information, see [Apache Ranger quick start guide](#) and [NiFi Ranger plugin source](#).

## Interaction with other security fields

Ranger authorization is separate from authentication and cannot be combined with a custom authorizer configuration.

`spec.security.ranger` is an authorizer configuration and is orthogonal to authentication configuration. You can combine it with any authentication method (Kerberos, LDAP, OIDC, SAML, certificate-based, or single user). The authorizer determines what an authenticated user may do; the authenticator determines who the user is.



### Important:

`spec.security.customAuthorizer` must not be set when `ranger` is set. They are mutually exclusive authorizer configurations.

## Related Concepts

[Configuring the initial admin user](#)

[Configuring single user authentication](#)

[Configuring a user identity with access policies](#)

[Configuring user groups](#)

[Configuring NiFi access policy profiles](#)

[Configuring LDAP authentication](#)

[Configuring OIDC authentication](#)

## Ranger authorization specification

The `ranger` block on a NiFi custom resource defines how the Ranger NiFi plugin connects to Ranger Admin and optional audit and TLS settings.

```
spec:
  security:
    ranger:
      serviceName: <string>           # required
      adminURL: <string>              # required
      adminIdentity: <string>         # required
      policyPollIntervalMs: <int>     # optional, default 30000
      audit:                          # optional
        solr:
          url: <string>               # optional
          zookeepersURL: <string>     # optional
          async: <bool>               # optional, default true
          maxQueueSize: <int>         # optional, default 10240
          maxFlushIntervalMs: <int>   # optional, default 30000
      tls:                            # optional
        secretName: <string>
        configSecretName: <string>   # optional escape hatch
```

This is a reference for each field under the Ranger authorization specification, including the Ranger plugin XML properties the operator generates from each value:

### **serviceName (string, required)**

Specifies the name of the NiFi service as registered in Ranger Admin's service repository. Ranger uses this name to look up the applicable policies when a request arrives. It must exactly match the service name shown on the Ranger Admin Service Manager page under the NiFi service type.

Stored in `ranger-nifi-security.xml` as `ranger.plugin.nifi.service.name`.

Example: `nifi-production`

### **adminURL (string, required)**

Specifies the base REST API URL of the Ranger Admin server. The Ranger NiFi plugin contacts this endpoint to download the policy set on startup and on each poll cycle. The URL must be reachable from inside the NiFi pods; include the port and scheme.

Stored in `ranger-nifi-security.xml` as `ranger.plugin.nifi.policy.rest.url`.

The plugin always uses the `RangerAdminRESTClient` implementation (`ranger.plugin.nifi.policy.source.impl`); that value is hardcoded by the operator and does not need to be set by you.

Example: `https://ranger-admin.example.com:6182`

**adminIdentity (string, required)**

Specifies the TLS distinguished name (DN) of the Ranger Admin server's certificate. NiFi's security model identifies all principals — including services — by their certificate DN. NiFi must recognize the Ranger Admin identity in order to trust policy download responses signed by that server.

This value is written into `authorizers.xml` as the Ranger Admin Identity property on the `RangerNiFiAuthorizer` block.

See the [Authorizers.xml Setup](#) in NiFi Administration Guide.

Example: `CN=ranger-admin.example.com, OU=Security, O=Acme, L=San Jose, ST=CA, C=US`

**policyPollIntervalMs (integer, optional, default 30000)**

Controls how frequently (in milliseconds) the Ranger NiFi plugin polls Ranger Admin for policy updates. Lower values reduce the lag between a policy change in Ranger and its effect in NiFi at the cost of more frequent HTTP calls to Ranger Admin. The default of 30 000 ms (30 seconds) is the Ranger plugin default and is appropriate for most deployments.

A local policy cache is written to `/tmp/ranger-nifi-security-cache` and is used as a fallback if Ranger Admin is temporarily unreachable.

Stored in `ranger-nifi-security.xml` as `ranger.plugin.nifi.policy.pollIntervalMs`.

Example: 15000 (poll every 15 seconds)

**audit (object, optional)**

Configures where the Ranger NiFi plugin writes access audit events. Audit records capture every authorization decision — allowed or denied — including the user identity, resource path, and action. When this section is omitted, Solr auditing is disabled (`xasecure.audit.solr.enable=false`).

Currently the only supported destination is Apache Solr.

**audit.solr (object, optional)**

Configures Apache Solr as the audit event destination.

Ranger's audit framework uses the `xasecure.audit.*` property namespace, inherited from the original XASecure project that became Ranger. For the full Solr audit configuration reference, see the [Apache Ranger quick start guide](#).

**audit.solr.url (string, optional)**

Specifies a direct HTTP or HTTPS URL pointing to the Solr collection that receives audit events. This is the simplest connectivity option when Solr is accessible at a known stable address.

Stored in `ranger-nifi-audit.xml` as `xasecure.audit.solr.url`.

Mutually exclusive with `zookeepersURL`: set one or the other, not both.

Example: `http://solr.example.com:6083/solr/ranger_audits`

**audit.solr.zookeepersURL (string, optional)**

Specifies a ZooKeeper connection string used for Solr service discovery. When the Solr cluster uses SolrCloud mode with ZooKeeper-based routing, this is the preferred option because it accommodates Solr node additions and removals without reconfiguring NiFi.

Stored in `ranger-nifi-audit.xml` as `xasecure.audit.solr.zookeepers`.

Example: `zk1.example.com:2181,zk2.example.com:2181/solr`

**audit.solr.async (bool, optional, default true)**

When true, audit events are written to an in-memory queue and flushed to Solr in batches. Asynchronous mode prevents a slow or temporarily unavailable Solr from adding latency to NiFi request processing. When false, each audit write blocks until Solr acknowledges receipt, providing stronger durability guarantees at the cost of throughput.

Stored in ranger-nifi-audit.xml as xasecure.audit.solr.async.

**audit.solr.maxQueueSize (integer, optional, default 10240)**

Specifies the maximum number of audit events held in the asynchronous buffer before the oldest events are dropped to make room for new ones. Applies only when async is true. Increase this value if audit events are being dropped during bursty authorization activity (for example, large batch jobs that touch many NiFi resources rapidly).

Stored in ranger-nifi-audit.xml as xasecure.audit.solr.async.max.queue.size.

**audit.solr.maxFlushIntervalMs (integer, optional, default 30000)**

Specifies the maximum time in milliseconds between forced flushes of the asynchronous audit queue to Solr. Even if the queue has not reached maxQueueSize, a flush is triggered once this interval elapses. Lower values reduce audit event lag at the cost of more frequent Solr writes. Applies only when async is true.

Stored in ranger-nifi-audit.xml as xasecure.audit.solr.async.max.flush.interval.ms.

**tls (object, optional)**

Configures mutual TLS (mTLS) between the Ranger NiFi plugin running inside NiFi pods and the Ranger Admin server. When the Ranger Admin endpoint is TLS-protected — which is strongly recommended in production — both a keystore (client certificate for NiFi to authenticate to Ranger Admin) and a truststore (CA certificate to verify Ranger Admin's server certificate) are typically required.

If this section is omitted entirely, the plugin will use Java's default truststore and will not present a client certificate. This is acceptable when Ranger Admin is configured to not require mutual TLS and uses a publicly trusted certificate, but is not recommended for production environments.

The underlying XML property namespace is xasecure.policymgr.clientssl.\*.

**tls.secretName (string, optional)**

Specifies the name of a Kubernetes Secret in the same namespace that contains the keystore and truststore NiFi uses to authenticate itself to Ranger Admin. The operator expects the Secret to contain the following keys:

| Secret key          | Description                               |
|---------------------|-------------------------------------------|
| keystore.jks        | The JKS-format keystore file (binary)     |
| keystore.password   | The plaintext password for the keystore   |
| truststore.jks      | The JKS-format truststore file (binary)   |
| truststore.password | The plaintext password for the truststore |

The keystore file is mounted read-only at /home/nifi/ranger-tls-keystore/keystore.jks inside the NiFi container. The password is injected at runtime; it does not appear in ranger-nifi-security.xml in cleartext — the placeholder ?RANGER\_KEYSTORE\_PASSWORD? is replaced by the plugin's credential provider mechanism.

Stored in ranger-nifi-security.xml as xasecure.policymgr.clientssl.keystore.

The truststore file is mounted read-only at /home/nifi/ranger-tls-truststore/truststore.jks.

Stored in ranger-nifi-security.xml as xasecure.policymgr.clientssl.truststore.

**configSecretName (string, optional)**

Provides an escape hatch for advanced Ranger configurations not expressible through the structured fields above. When set, the operator skips generating ranger-nifi-security.xml and ranger-nifi-audit.xml entirely and instead mounts the files from this Kubernetes Secret.

The Secret must be in the same namespace as the NiFi resource and must contain one or both of the following keys:

| Secret key               | Mount path                                           |
|--------------------------|------------------------------------------------------|
| ranger-nifi-security.xml | /opt/nifi/nifi-current/conf/ranger-nifi-security.xml |
| ranger-nifi-audit.xml    | /opt/nifi/nifi-current/conf/ranger-nifi-audit.xml    |

Use cases include:

- Properties not yet modeled in the operator schema (for example, Kafka or HDFS audit destinations).
- Ranger plugin versions with non-standard property names.
- Environments requiring credential provider integration (jceks:// URIs for passwords).

When configSecretName is set, all structured fields under audit and tls are still validated by the API server but their values are not used, the Secret contents take complete precedence.

**Behavior under Ranger mode**

When Ranger authorization is enabled, User and UserGroup resources still sync NiFi tenants but skip access policy management, and authorizer selection follows a fixed precedence.

**User and UserGroup controllers**

When ranger is configured, User and UserGroup resources still create and update NiFi tenant identities (users and groups) via the NiFi REST API. This is necessary so that NiFi can resolve identity strings returned from Ranger policy checks to internal user objects.

However, the controllers skip all access policy operations — they do not call CreateAccessPolicy, UpdateAccessPolicy, or add users to policy resource entries. The accessPolicies and accessPolicyProfileRef fields on User and UserGroup resources are accepted by the API server but have no effect when Ranger is enabled. Policy management is Ranger's responsibility.

For more information, see [Configuring a user identity with access policies](#) on page 26 and [Configuring user groups](#) on page 28.

**Authorizer selection**

The nifi.security.user.authorizer property in nifi.properties is set based on the following precedence:

1. single-user-authorizer — when Single User Auth is enabled.
2. ranger-nifi-authorizer — when spec.security.ranger is set.
3. managed-authorizer — the default file-based authorizer.

Consider the following three configuration scenarios:

1. security.singleUserAuth and security.ranger
2. security.ldap and security.ranger
3. security.ldap

For case 1, single user authentication configures NiFi to ignore all authorizations because there is only the single super user.

For case 2, user authentication is handled by LDAP and authorization is handled by Ranger.

For case 3, user authentication is handled by LDAP and authorization is handled by managed-authorizer (file-based access policies).

### Ranger authorization examples

Example NiFi custom resource configurations for Ranger authorization, from minimal setups to production deployments with mTLS, Solr audit, and custom XML overrides.

#### Minimal configuration (no TLS, no audit)

```
apiVersion: cfm.cloudera.com/v1alpha1
kind: Nifi
metadata:
  name: nifi-prod
  namespace: nifi
spec:
  security:
    ranger:
      serviceName: nifi-prod
      adminURL: https://ranger-admin.example.com:6182
      adminIdentity: "CN=ranger-admin.example.com, OU=Security, O=Acme, L=San Jose, ST=CA, C=US"
```

#### Production configuration (mTLS + Solr audit)

```
apiVersion: cfm.cloudera.com/v1alpha1
kind: Nifi
metadata:
  name: nifi-prod
  namespace: nifi
spec:
  security:
    ranger:
      serviceName: nifi-prod
      adminURL: https://ranger-admin.example.com:6182
      adminIdentity: "CN=ranger-admin.example.com, OU=Security, O=Acme, L=San Jose, ST=CA, C=US"
      policyPollIntervalMs: 15000
      audit:
        solr:
          url: http://solr.example.com:6083/solr/ranger_audits
          async: true
          maxQueueSize: 20480
          maxFlushIntervalMs: 10000
      tls:
        secretName: nifi-ranger-tls
```

#### ZooKeeper-based Solr discovery

```
spec:
  security:
    ranger:
      serviceName: nifi-prod
      adminURL: https://ranger-admin.example.com:6182
      adminIdentity: "CN=ranger-admin.example.com, OU=Security, O=Acme, L=San Jose, ST=CA, C=US"
      audit:
        solr:
          zookeepersURL: "zk1.example.com:2181,zk2.example.com:2181/solr"
```



**Override with custom XML (escape hatch)**

```
spec:
  security:
    ranger:
      serviceName: nifi-prod
      adminURL: https://ranger-admin.example.com:6182
      adminIdentity: "CN=ranger-admin.example.com, OU=Security, O=Acme, L=San Jose, ST=CA, C=US"
      configSecretName: nifi-ranger-custom-config
```

```
apiVersion: v1
kind: Secret
metadata:
  name: nifi-ranger-custom-config
  namespace: nifi
stringData:
  ranger-nifi-security.xml: |
    <?xml version="1.0"?>
    <configuration>
      <property>
        <name>ranger.plugin.nifi.service.name</name>
        <value>nifi-prod</value>
      </property>
      <property>
        <name>ranger.plugin.nifi.policy.source.impl</name>
        <value>org.apache.ranger.admin.client.RangerAdminRESTClient</value>
      </property>
      <property>
        <name>ranger.plugin.nifi.policy.rest.url</name>
        <value>https://ranger-admin.example.com:6182</value>
      </property>
      <property>
        <name>ranger.plugin.nifi.policy.pollIntervalMs</name>
        <value>30000</value>
      </property>
      <property>
        <name>ranger.plugin.nifi.policy.cache.dir</name>
        <value>/tmp/ranger-nifi-security-cache</value>
      </property>
      <property>
        <name>xasecure.policymgr.clientssl.keystore</name>
        <value>/home/nifi/ranger-tls-keystore/keystore.jks</value>
      </property>
      <property>
        <name>xasecure.policymgr.clientssl.keystore.credential.file</name>
        <value>jceks://file/home/nifi/ranger-tls-keystore/keystore-credential.jceks</value>
      </property>
      <property>
        <name>xasecure.policymgr.clientssl.truststore</name>
        <value>/home/nifi/ranger-tls-truststore/truststore.jks</value>
      </property>
      <property>
        <name>xasecure.policymgr.clientssl.truststore.credential.file</name>
        <value>jceks://file/home/nifi/ranger-tls-truststore/truststore-credential.jceks</value>
      </property>
    </configuration>
  ranger-nifi-audit.xml: |
    <?xml version="1.0"?>
    <configuration>
      <property>
```

```

    <name>xasecure.audit.solr.enable</name>
    <value>true</value>
  </property>
  <property>
    <name>xasecure.audit.solr.async</name>
    <value>true</value>
  </property>
  <property>
    <name>xasecure.audit.solr.url</name>
    <value>http://solr.example.com:6083/solr/ranger_audits</value>
  </property>
  <property>
    <name>xasecure.audit.solr.async.max.queue.size</name>
    <value>10240</value>
  </property>
  <property>
    <name>xasecure.audit.solr.async.max.flush.interval.ms</name>
    <value>30000</value>
  </property>
</configuration>

```

### Generated Ranger XML files

When you do not set a config secret, Cloudera Flow Management Operator for Kubernetes generates the Ranger security and audit XML files from the structured spec fields.

When configSecretName is not set, the operator generates the two XML configuration files from the structured spec fields. The following examples show what the operator produces for a production configuration.

### ranger-nifi-security.xml (generated)

```

<?xml version="1.0"?>
<configuration>
  <!-- Service identity and policy source -->
  <property>
    <name>ranger.plugin.nifi.service.name</name>
    <value>nifi-prod</value>
  </property>
  <property>
    <name>ranger.plugin.nifi.policy.source.impl</name>
    <value>org.apache.ranger.admin.client.RangerAdminRESTClient</value>
  </property>
  <property>
    <name>ranger.plugin.nifi.policy.rest.url</name>
    <value>https://ranger-admin.example.com:6182</value>
  </property>
  <property>
    <name>ranger.plugin.nifi.policy.pollIntervalMs</name>
    <value>15000</value>
  </property>

  <!-- Local policy cache used as fallback when Ranger Admin is unreachable -->
  <property>
    <name>ranger.plugin.nifi.policy.cache.dir</name>
    <value>/tmp/ranger-nifi-security-cache</value>
  </property>

  <!-- mTLS: keystore (present when tls.secretName is set) -->
  <property>
    <name>xasecure.policymgr.clientssl.keystore</name>
    <value>/home/nifi/ranger-tls-keystore/keystore.jks</value>
  </property>

```

```

<property>
  <name>xasecure.policymgr.clientssl.keystore.password</name>
  <value>?RANGER_KEYSTORE_PASSWORD?</value>
</property>

<!-- mTLS: truststore (present when tls.secretName is set) -->
<property>
  <name>xasecure.policymgr.clientssl.truststore</name>
  <value>/home/nifi/ranger-tls-truststore/truststore.jks</value>
</property>
<property>
  <name>xasecure.policymgr.clientssl.truststore.password</name>
  <value>?RANGER_TRUSTSTORE_PASSWORD?</value>
</property>
</configuration>

```

### ranger-nifi-audit.xml (generated, Solr enabled)

```

<?xml version="1.0"?>
<configuration>
  <property>
    <name>xasecure.audit.solr.enable</name>
    <value>true</value>
  </property>
  <property>
    <name>xasecure.audit.solr.async</name>
    <value>true</value>
  </property>
  <property>
    <name>xasecure.audit.solr.url</name>
    <value>http://solr.example.com:6083/solr/ranger_audits</value>
  </property>
  <property>
    <name>xasecure.audit.solr.async.max.queue.size</name>
    <value>20480</value>
  </property>
  <property>
    <name>xasecure.audit.solr.async.max.flush.interval.ms</name>
    <value>10000</value>
  </property>
</configuration>

```

### ranger-nifi-audit.xml (generated, Solr disabled)

When spec.security.ranger.audit is omitted:

```

<?xml version="1.0"?>
<configuration>
  <property>
    <name>xasecure.audit.solr.enable</name>
    <value>false</value>
  </property>
</configuration>

```

### TLS Secret format for Ranger authorization

Cloudera Flow Management Operator for Kubernetes expects a Kubernetes Secret with keystore and truststore files for mutual TLS between NiFi and Ranger Admin.

## Keystore Secret

```
apiVersion: v1
kind: Secret
metadata:
  name: nifi-ranger-keystore
  namespace: nifi
type: Opaque
data:
  # base64-encoded JKS keystore file
  keystore.jks: <base64>
  # base64-encoded plaintext password string
  keystore.password: <base64>
  # base64-encoded JKS truststore file
  truststore.jks: <base64>
  # base64-encoded plaintext password string
  truststore.password: <base64>
```

## Configuring JVM security providers (FIPS)

NiFi and NiFi Registry are not FIPS compliant out of the box. When booting cfm-nifi-k8s for NiFi version 1 on a FIPS enabled cluster, the Pod will enter a CrashLoop attempting to load JKS keystores. NiFi version 2 will boot but not necessarily be compliant. Follow the instructions here to add additional security providers to the NiFi JVM to enable FIPS compliance.

### Prerequisites

FIPS compliance requires special security providers to be given to the NiFi and NiFi Registry containers. To fully configure these new providers, the operator requires a few pieces of information:

1. Security provider jars.
2. Keystore provider class.
3. Preferred keystore format.
4. Security providers definition.
5. Java policy for providers. (optional)

#### Security provider jars

These are Java jar files containing FIPS compliant security providers that you have obtained from [Cloudera](#) (CCJ and BCTLS) or another vendor, such as Safelogic. The jars should be referred to by the environment variable PROVIDER\_JAR\_PATH.

The rest of this document will show examples using ccj and bctls from Cloudera's archive mirror.

#### Keystore provider class

The provider class that should be used for constructing keystores and truststores. Using ccj, this would be com.safelogic.cryptocomply.jcajce.provider.CryptoComplyFipsProvider. This will be provided to NiFi by environment variable KEYSTORE\_PROVIDER\_CLASS.

#### Preferred keystore format

The default keystore format JKS is a weak format and generally not FIPS compliant. Your security provider may provide a different format, such as Bouncy Castle FIPS KeyStore (BCFKS). This will be supplied to NiFi by environment variable KEYSTORE\_TYPE.

#### Security providers definition

The security providers to add to the JVM must be provided in a file with one provider per line.

CCJ example:

```
$ cat additional-security-providers.txt
com.safelogic.cryptocomply.jcajce.provider.CryptoComplyFipsProvider
org.bouncycastle.jsse.provider.BouncyCastleJsseProvider fips:CCJ
```

A path reference to this file must be provided with an environment variable `SECURITY_PROVIDERS_PATH`.

### Java policy for providers

For some providers, additional permissions may need to be given via Java policy. A standard Java policy file can be provided, see this CCJ example:

```
$ cat additional-java-policy.txt
grant {
    //CCJ Java Permissions
    permission java.lang.RuntimePermission "getProtectionDomain";
    permission java.lang.RuntimePermission "accessDeclaredMembers";
    permission java.util.PropertyPermission "java.runtime.name",
    "read";
    permission java.security.SecurityPermission "putProviderProperty.CCJ";
    //CCJ Key Export and Translation
    permission com.safelogic.cryptocomply.crypto.CryptoServicesPermission "exportKeys";
    //CCJ SSL
    permission com.safelogic.cryptocomply.crypto.CryptoServicePermission "tlsAlgorithmsEnabled";
    //CCJ Setting of Default SecureRandom
    permission com.safelogic.cryptocomply.crypto.CryptoServicePermission "defaultRandomConfig";
    //CCJ Setting CryptoServicesRegistrar Properties
    permission com.safelogic.cryptocomply.crypto.CryptoServicesPermission "globalConfig";
    //CCJ Enable JKS
    permission com.safelogic.cryptocomply.jca.enable_jks "true";
};
```

A path reference to this file must be provided with an environment variable `JAVA_POLICY_PATH`.

## Configuration

The Cloudera Flow Management Kubernetes Operator for Apache NiFi has two methods of providing FIPS compliant security providers to the NiFi JVM: image rebuild or with volumes.

### Image rebuild



**Note:**

This option requires access to an internal container registry.

This is the recommended method of enabling FIPS if you've got the infrastructure to utilize, as this requires no runtime configuration, Flow developer teams will simply reference the new FIPS enabled image.

You can provide all required JVM Security Provider Information directly to the `cfm-nifi-k8s` and `cfm-nifiregistry-k8s` images via an image rebuild. With this method, you will create a Dockerfile that modifies the images you've pulled from Cloudera prior to pushing them to your internal registries.

1. In a directory, place the provider jars, provider definition file, and optional java policy file.

```
$ ls
additional-java-policy.txt  additional-security-providers.txt  bctls.jar
ccj-3.0.2.1.jar
```

2. Create a Dockerfile.

```
# Use args to parameterize this Dockerfile for reuse
ARG CFM_NIFI_K8S_BASE_IMAGE=container.repository.cloudera.com/cloudera/cfm-nifi-k8s
ARG CFM_NIFI_K8S_BASE_TAG=2.9.0-b96-nifi_1.27.0.2.3.14.0-14

FROM ${CFM_NIFI_K8S_BASE_IMAGE}:${CFM_NIFI_K8S_BASE_TAG} AS nifi-k8s

# Copy the required files
COPY bctls.jar ccj-3.0.2.1.jar $NIFI_HOME/lib/
COPY additional-java-policy.txt additional-security-providers.txt $NIFI_HOME/conf/
# Configure environment variables to point to the provided files
ENV PROVIDER_JAR_PATH="$NIFI_HOME/lib/ccj-3.0.2.1.jar:$NIFI_HOME/lib/bctls.jar"
ENV JAVA_POLICY_PATH="$NIFI_HOME/conf/additional-java-policy.txt"
ENV SECURITY_PROVIDERS_PATH="$NIFI_HOME/conf/additional-security-providers.txt"
# Configure the keystore type
ENV KEYSTORE_TYPE=BCFKS

# Specify the security provider classe
ENV KEYSTORE_PROVIDER_CLASS=com.safelogic.cryptocomply.jcajce.provider.CryptoComplyFipsProvider
```

3. Build the new image.

```
docker build -t <your-registry>/cloudera/cfm-nifi-k8s:2.9.0-b96-nifi_1.27.0.2.3.14.0-14-fips .
docker push <your-registry>/cloudera/cfm-nifi-k8s:2.9.0-b96-nifi_1.27.0.2.3.14.0-14-fips
```

### Using volumes

Using volumes, Security Providers can be configured at deploy time using the standard cfm-nifi-k8s and cfm-nifiregistry-k8s images provided by Cloudera. Prior to deploying NiFi or NiFi Registry, a volume that supports RWX should be created and populated with the required files:

- Security provider jars
- Security provider definition file
- Additional Java policy

1. In your Nifi or NifiRegistry yamls, add the following to mount the volume:

```
spec:
  statefulset:
    volumes:
      - name: fips-providers
        persistentVolumeClaim:
          claimName: [***RWX VOLUME CLAIM***]
    volumeMounts:
      - name: fips-providers
        mountPath: /opt/nifi/fips-providers
```

2. Reference the provided files, keystore type, and keystore provider class:

```
spec:
  security:
    jvmSecurityProviderInfo:
      # List of provider jars in classpath format
      providerJarPath: "/opt/nifi/fips-providers/ccj-3.0.2.1.jar:/opt/nifi/fips-providers/bctls.jar"
      # Class providing the keystore implementation
      providerClass: com.safelogic.cryptocomply.jcajce.provider.CryptoComplyFipsProvider
      # Keystore format
      keystoreType: BCFKS
      # Path to security providers definition
      securityProvidersPath: /opt/nifi/fips-providers/additional-security-providers.txt
      # Path to additional Java policy
      javaPolicyPath: /opt/nifi/fips-providers/additional-java-policy.txt
```

## Example CR

The following example NiFi CR deploys a 3 node cluster with Kubernetes-based state management and leader election, and a Route to access the NiFi UI.

```
apiVersion: cfm.cloudera.com/v1
kind: Nifi
metadata:
  name: mynifi
spec:
  nifiVersion: [***NIFI VERSION***]
  replicas: 3
  image:
    repository: container.repository.cloudera.com/cloudera/cfm-nifi-k8s
    tag: [***NIFI TAG***]
    pullSecret: docker-pull-secret
  tiniImage:
    repository: container.repository.cloudera.com/cloudera/cfm-tini
    tag: [***CFM TINI TAG***]
    pullSecret: docker-pull-secret
  hostName: mynifi.[***OPENSIFT ROUTER DOMAIN***]
  uiConnection:
    type: Route
    serviceConfig:
      sessionAffinity: ClientIP
```