Cloudera Flow Management Operator for Kubernetes 2.11.0

Configuring NiFi Registry CR

Date published: 2024-06-11
Date modified: 2025-09-30

https://docs.cloudera.com/

Legal Notice

Contents

Configuring a NiFi Registry cluster

Cloudera Flow Management Operator for Kubernetes can deploy NiFi Registry instances using the NiFiRegistry custom resource. NiFi Registry instances are configured through these CRs. No additional configuration is required after deployment.

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

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

Configuring group, version, kind, and 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 on top of each NiFi Registry custom resource (CR) you write. It defines the group “cfm.cloudera.com”, the version “v1alpha1”, the kind “NifiRegistry”, and the name of your cluster and the 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/v1alpha
kind: NifiRegistry
metadata:
  name:[***NIFI REGISTRY NAME***]
```

Configuring images

This section describes the images used for running NiFi Registry. This provides a way of manually upgrading the NiFi version in an existing cluster or very quickly rolling out NiFi clusters with new versions.

A CFM NiFi Registry deployment includes two container images: cfm-nifiregistry-k8s and cfm-tini. The cfm-nifiregistry-k8s image is the actual registry image itself. The cfm-tini image is a small utility image used for aggregating logs.

Pulling images from Cloudera’s registries requires a pull secret containing your Cloudera credentials. Create this pull secret with

```
kubectl create secret docker-registry my-pull-secret \
--docker-username=[***CLOUDERA USER***] \
--docker-password=[***CLOUDERA PASSWORD***] \
--docker-server=container.repository.cloudera.com
```

```
spec:
    image:
        repository: [***CFM-NIFI-REGISTRY-K8S REPOSITORY***]
        tag: 2.8.0-bXX
        imagePullPolicy: IfNotPresent
        pullSecret: my-pull-secret
    tiniImage:
        Repository: [***CFM-TINI REPOSITORY***]
        tag: 2.8.0-bXX
        imagePullPolicy: IfNotPresent
```

```
        pullSecret: [***PULL SECRET***]
```

- The default *[***CFM-NIFI-REGISTRY-K8S REPOSITORY***]* is container.repository.cloudera.com/cloudera/cfm-nifiregistry-k8s
- The default *[***CFM-TINI REPOSITORY***]* is container.repository.cloudera.com/cloudera/cfm-tini

if your Kubernetes cluster has no internet connection or you want to use a self-hosted repository, replace these with the relevant paths.

Configuring persistence

Learn about configuring storage for NiFi Registry.

Cloudera Flow Management Operator for Kubernetes can configure persistent volumes for the following directories:

- flow_storage
- data
- extension_bundles

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

```
spec:
  persistence:
    size: 1Gi
    storageClass: default
    flowStorage:
      size: 3Gi
    data: {}
    extensionBundles:
      storageClass: SOME-STORAGE-CLASS
```

Configuring node certificate generation

Learn about certificate generation options.

Note:

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 connections to NiFi Registry

Learn about configuring connections for your NiFi Registry cluster.

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

Learn about configuring a connection to the NiFi Registry web UI.

You can configure a connection to the NiFi Registry 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 Registry does not support it well, but it is possible. For more information, see NiFi Registry documentation on proxy configuration. An example using ingress-nginx is included in this section.

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 Registry Web UI. It uses the TLS files generated by a Cloudera Flow Management Operator for Kubernetes created Certificate as defined in spec.securit y.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 Registry node in the cluster. The backend-protocol setting is needed for when NiFi Registry 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 acces the NiFi Registry web UI.

This YAML snippet configures a Route resource for accessing the NiFi Registry 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 Registry 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 Registry does not handle proxy paths well, it does not understand that https://nifi-registry.localhost/some/path/to/nifi-registry coming through the defined Ingress is intended to call the /nifi-registry API to load the UI. The rewrite-target annotation addresses this by capturing the /nifi-registry and anything that comes after and sends that as the path to the NiFi Registry pod. It translates /some/path/to/nifi-registry/ to /nifi-registry/. Similarly, the NiFi Registry web UI does not correctly form API calls going to the backend, attempting to call /nifi-registry/ instead of /some/path/to/nifi-registry/. 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 Registry 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-registry-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 Registry 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 Registry beyond the hostname provided in hostName. To add additional proxy hosts, add the following to your NiFi Registry YAML:

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

Configuring authentication and authorization for NiFi Registry

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

Note:

NiFi Registry 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:

Related Concepts

Configuring node certificate generation

Configuring the initial admin user

When you set up a secured NiFi Registry 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 Registry 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.

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 usiing the cubectl CLI utility:

```
kubectl create secret generic my-ldap-creds \
       --from-literal=managerPassword=myMan@gerPassw0rd \
 --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
```

Configuring OIDC authentication

NiFi Registry 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-configurat
ion
```

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

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

The Discovery URL and client credentials secret are provided to NiFi Registry 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 Registry attempts to validate HMAC protected tokens using the specified client secret. If this value is none, NiFi Registry 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 Registry to trust the certificate presented by the OIDC server, you must add a valid CA for your OIDC server to NiFi Registry. For informatioon on adding a CA to NiFi Registry, see Additional CA Bundles.

Related Information

OpenID Connect | Apache NiFi System Administrator's Guide

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.CryptoComplyFipsProv
ider
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 "putProviderProp
erty.CCJ";
    //CCJ Key Export and Translation
    permission com.safelogic.cryptocomply.crypto.CryptoServicesP
ermission "exportKeys";
    //CCJ SSL
```

```
    permission com.safelogic.cryptocomply.crypto.CryptoService
sPermission "tlsAlgorithmsEnabled";
    //CCJ Setting of Default SecureRandom
    permission com.safelogic.cryptocomply.crypto.CryptoService
sPermission "defaultRandomConfig";
    //CCJ Setting CryptoServicesRegistrar Properties
    permission com.safelogic.cryptocomply.crypto.CryptoServicesP
ermission "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-provide
rs.txt"
# Configure the keystore type
ENV KEYSTORE_TYPE=BCFKS

# Specify the security provider classe
```

```
ENV KEYSTORE_PROVIDER_CLASS=com.safelogic.cryptocomply.jcajce.provider.Cr
yptoComplyFipsProvider
```

3. Build the new image.

```
docker build -t <your-registry>/cloudera/cfm-nifi-k8s:2.9.0-b96-nifi_1.2
7.0.2.3.14.0-14-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/ni
fi/fips-providers/bctls.jar"
 # Class providing the keystore implementation
      providerClass: com.safelogic.cryptocomply.jcajce.provider.CryptoCo
mplyFipsProvider
 # 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

This custom resource example configures a basic NiFi Registry instance with a single replica, no security, and a Route to connect to the UI.

```
apiVersion: cfm.cloudera.com/v1alpha1
kind: NifiRegistry
metadata:
```

```
    name: mynifiregistry
spec:
    image:
        repository: container.repository.cloudera.com/cloudera/cfm-nifi-k8s
        tag: [***NIFI REGISTRY TAG***]
    tiniImage:
        repository: container.repository.cloudera.com/cloudera/cfm-tini
        tag: [***CFM TINI TAG***]
    hostName: mynifiregistry.[***OPENSHIFT ROUTER DOMAIN***]
    uiConnection:
        type: Route
```