Red Hat build of MicroShift 4.16

Networking

Configuring and managing cluster networking

Last Updated: 2024-06-28

Red Hat build of MicroShift 4.16 Networking

Configuring and managing cluster networking

Legal Notice

Copyright © 2024 Red Hat, Inc.

The text of and illustrations in this document are licensed by Red Hat under a Creative Commons Attribution–Share Alike 3.0 Unported license ("CC-BY-SA"). An explanation of CC-BY-SA is available at
http://creativecommons.org/licenses/by-sa/3.0/
. In accordance with CC-BY-SA, if you distribute this document or an adaptation of it, you must provide the URL for the original version.

Red Hat, as the licensor of this document, waives the right to enforce, and agrees not to assert, Section 4d of CC-BY-SA to the fullest extent permitted by applicable law.

Red Hat, Red Hat Enterprise Linux, the Shadowman logo, the Red Hat logo, JBoss, OpenShift, Fedora, the Infinity logo, and RHCE are trademarks of Red Hat, Inc., registered in the United States and other countries.

Linux ® is the registered trademark of Linus Torvalds in the United States and other countries.

Java ® is a registered trademark of Oracle and/or its affiliates.

XFS ® is a trademark of Silicon Graphics International Corp. or its subsidiaries in the United States and/or other countries.

MySQL ® is a registered trademark of MySQL AB in the United States, the European Union and other countries.

Node.js ® is an official trademark of Joyent. Red Hat is not formally related to or endorsed by the official Joyent Node.js open source or commercial project.

The OpenStack ® Word Mark and OpenStack logo are either registered trademarks/service marks or trademarks/service marks of the OpenStack Foundation, in the United States and other countries and are used with the OpenStack Foundation's permission. We are not affiliated with, endorsed or sponsored by the OpenStack Foundation, or the OpenStack community.

All other trademarks are the property of their respective owners.

Abstract

This document provides instructions for configuring and managing your MicroShift cluster network, including DNS, ingress, and the Pod network.

Table of Contents

CHAPTER 1. ABOUT THE OVN-KUBERNETES NETWORK PLUGIN 5
1.1. MICROSHIFT NETWORKING CONFIGURATION MATRIX 5
1.1.1. Default settings 6
1.2. NETWORK FEATURES 9
1.3. IP FORWARD 9
1.4. NETWORK PERFORMANCE OPTIMIZATIONS 9
1.5. MICROSHIFT NETWORKING COMPONENTS AND SERVICES 9
1.6. BRIDGE MAPPINGS 10
1.7. NETWORK TOPOLOGY 10
1.7.1. Description of the OVN logical components of the virtualized network 11
1.7.2. Description of the connections in the network topology figure 12
1.8. ADDITIONAL RESOURCES 12

CHAPTER 2. UNDERSTANDING NETWORKING SETTINGS 13
2.1. CREATING AN OVN-KUBERNETES CONFIGURATION FILE 13
2.2. RESTARTING THE OVNKUBE-MASTER POD 14
2.3. DEPLOYING MICROSHIFT BEHIND AN HTTP OR HTTPS PROXY 14
2.4. USING THE RPM-OSTREE HTTP OR HTTPS PROXY 15
2.5. USING A PROXY IN THE CRI-O CONTAINER RUNTIME 15
2.6. GETTING A SNAPSHOT OF OVS INTERFACES FROM A RUNNING CLUSTER 16
2.7. THE MICROSHIFT LOADBALANCER SERVICE FOR WORKLOADS 17
2.8. DEPLOYING A LOAD BALANCER FOR AN APPLICATION 17
2.9. BLOCKING EXTERNAL ACCESS TO THE NODEPORT SERVICE ON A SPECIFIC HOST INTERFACE 21
2.10. THE MULTICAST DNS PROTOCOL 22
2.11. AUDITING EXPOSED NETWORK PORTS 22
2.11.1. hostNetwork 22
2.11.2. hostPort 23
2.11.3. NodePort and LoadBalancer services 23

CHAPTER 3. UNDERSTANDING AND CONFIGURING THE ROUTER 25
3.1. ABOUT CONFIGURING THE ROUTER 25
3.1.1. Router settings and valid values 25
3.2. DISABLING THE ROUTER 26
3.3. CONFIGURING ROUTER INGRESS 27
3.3.1. Configuring router ports 27
3.3.2. Configuring router IP addresses 28
3.4. ADDITIONAL RESOURCES 29
3.5. CONFIGURING THE ROUTE ADMISSION POLICY 29

CHAPTER 4. NETWORK POLICIES 31
4.1. ABOUT NETWORK POLICIES 31
4.1.1. How network policy works in MicroShift 31
4.1.2. Optimizations for network policy with OVN-Kubernetes network plugin 33
4.2. CREATING NETWORK POLICIES 35
4.2.1. Example NetworkPolicy object 35
4.2.2. Creating a network policy using the CLI 36
4.2.3. Creating a default deny all network policy 37
4.2.4. Creating a network policy to allow traffic from external clients 38
4.2.5. Creating a network policy allowing traffic to an application from all namespaces 39
4.2.6. Creating a network policy allowing traffic to an application from a namespace 41
4.3. EDITING A NETWORK POLICY 43
4.3.1. Editing a network policy 43

4.3.2. Example NetworkPolicy object 44
4.4. DELETING A NETWORK POLICY 45
4.4.1. Deleting a network policy using the CLI 45
4.5. VIEWING A NETWORK POLICY 45
4.5.1. Viewing network policies using the CLI 46

CHAPTER 5. MULTIPLE NETWORKS 47
5.1. ABOUT USING MULTIPLE NETWORKS 47
5.1.1. Additional networks in MicroShift 47
5.1.1.1. Supported additional networks for network isolation 47
5.1.1.2. Use case: Additional networks for network isolation 47
5.1.1.3. How additional networks are implemented 48
5.1.1.4. How to attached additional networks to pods 48
5.1.1.5. Configurations for additional network types 48
5.1.2. Installing the Multus CNI plugin on a running cluster 48
5.1.3. Configuration for a bridge additional network 49
5.1.3.1. Bridge CNI plugin configuration example 50
5.1.4. Configuration for an ipvlan additional network 51
5.1.4.1. IPVLAN CNI plugin configuration example 52
5.1.5. Configuration for a macvlan additional network 53
5.1.5.1. MACVLAN CNI plugin configuration example 53
5.1.6. Additional resources 54
5.2. CONFIGURING AND USING MULTIPLE NETWORKS 54
5.2.1. IP address management types and additional networks 54
5.2.1.1. bridge interface specifics 54
5.2.1.2. macvlan interface specifics 54
5.2.1.3. ipvlan interface specifics 54
5.2.2. Creating a NetworkAttachmentDefinition for an additional network 54
5.2.3. Adding a pod to an additional network 56
5.2.4. Configuring an additional network 59
5.2.5. Removing a pod from an additional network 62
5.2.6. Troubleshooting Multus networking 62
5.2.6.1. Pod networking cannot be configured 62
5.2.6.2. Missing configuration file 63
5.2.7. Additional resources 63

CHAPTER 6. CONFIGURING ROUTES 64
6.1. CREATING AN HTTP-BASED ROUTE 64
6.1.1. HTTP Strict Transport Security 65
6.1.2. Enabling HTTP Strict Transport Security per-route 65
6.1.3. Disabling HTTP Strict Transport Security per-route 67
6.1.4. Enforcing HTTP Strict Transport Security per-domain 68
6.2. THROUGHPUT ISSUE TROUBLESHOOTING METHODS 68
6.3. USING COOKIES TO KEEP ROUTE STATEFULNESS 69
6.3.1. Annotating a route with a cookie 69
6.4. PATH-BASED ROUTES 70
6.5. HTTP HEADER CONFIGURATION 71
6.5.1. Special case headers 72
6.6. SETTING OR DELETING HTTP REQUEST AND RESPONSE HEADERS IN A ROUTE 73
6.7. CREATING A ROUTE THROUGH AN INGRESS OBJECT 74
6.8. CREATING A ROUTE USING THE DEFAULT CERTIFICATE THROUGH AN INGRESS OBJECT 77
6.9. CREATING A ROUTE USING THE DESTINATION CA CERTIFICATE IN THE INGRESS ANNOTATION 78
6.10. SECURED ROUTES 79

CHAPTER 7. USING A FIREWALL 80
7.1. ABOUT NETWORK TRAFFIC THROUGH THE FIREWALL 80
7.2. INSTALLING THE FIREWALLD SERVICE 80
7.3. REQUIRED FIREWALL SETTINGS 81
7.4. USING OPTIONAL PORT SETTINGS 81
7.5. ADDING SERVICES TO OPEN PORTS 82
7.6. ALLOWING NETWORK TRAFFIC THROUGH THE FIREWALL 83
7.6.1. Applying firewall settings 83
7.7. VERIFYING FIREWALL SETTINGS 83
7.8. OVERVIEW OF FIREWALL PORTS WHEN A SERVICE IS EXPOSED 84
7.9. ADDITIONAL RESOURCES 84
7.10. KNOWN FIREWALL ISSUE 84

CHAPTER 8. CONFIGURING NETWORK SETTINGS FOR FULLY DISCONNECTED HOSTS 85
8.1. PREPARING NETWORKING FOR FULLY DISCONNECTED HOSTS 85
8.1.1. Procedure summary 85
8.2. RESTORING MICROSHIFT NETWORKING SETTINGS TO DEFAULT 86
8.3. CONFIGURING THE NETWORKING SETTINGS FOR FULLY DISCONNECTED HOSTS 86

CHAPTER 1. ABOUT THE OVN-KUBERNETES NETWORK PLUGIN

The OVN-Kubernetes Container Network Interface (CNI) plugin is the default networking solution for MicroShift clusters. OVN-Kubernetes is a virtualized network for pods and services that is based on Open Virtual Network (OVN).

- Default network configuration and connections are applied automatically in MicroShift with the **microshift-networking** RPM during installation.
- A cluster that uses the OVN-Kubernetes network plugin also runs Open vSwitch (OVS) on the node.
- OVN-K configures OVS on the node to implement the declared network configuration.
- Host physical interfaces are not bound by default to the OVN-K gateway bridge, **br-ex**. You can use standard tools on the host for managing the default gateway, such as the Network Manager CLI (**nmcli**).
- Changing the CNI is not supported on MicroShift.

Using configuration files or custom scripts, you can configure the following networking settings:

- You can use subnet CIDR ranges to allocate IP addresses to pods.
- You can change the maximum transmission unit (MTU) value.
- You can configure firewall ingress and egress.
- You can define network policies in the MicroShift cluster, including ingress and egress rules.
- You can use the MicroShift Multus plug-in to chain other CNI plugins.
- You can configure or remove the ingress router.

1.1. MICROSHIFT NETWORKING CONFIGURATION MATRIX

The following table summarizes the status of networking features and capabilities that are either present as defaults, supported for configuration, or not available with the MicroShift service:

Table 1.1. MicroShift networking features and capabilities overview

Network capability	Availability	Configuration supported
Advertise address	Yes	Yes [1]
Kubernetes network policy	Yes	Yes
Kubernetes network policy logs	Not available	N/A
Load balancing	Yes	Yes
Multicast DNS	Yes	Yes [2]

Network capability	Availability	Configuration supported
Network proxies	Yes [3]	CRI-O
Network performance	Yes	MTU configuration
Egress IPs	Not available	N/A
Egress firewall	Not available	N/A
Egress router	Not available	N/A
Firewall	No [4]	Yes
Hardware offloading	Not available	N/A
Hybrid networking	Not available	N/A
IPsec encryption for intra-cluster communication	Not available	N/A
IPv6	Not available [5]	N/A
Ingress router	Yes	Yes [6]
Multiple networks plug-in	Yes	Yes

1. If unset, the default value is set to the next immediate subnet after the service network. For example, when the service network is **10.43.0.0/16**, the **advertiseAddress** is set to **10.44.0.0/32**.
2. You can use the multicast DNS protocol (mDNS) to allow name resolution and service discovery within a Local Area Network (LAN) using multicast exposed on the **5353/UDP** port.
3. There is no built-in transparent proxying of egress traffic in MicroShift. Egress must be manually configured.
4. Setting up the firewalld service is supported by RHEL for Edge.
5. IPv6 is not supported. IPv6 can only be used by connecting to other networks with the MicroShift Multus CNI plugin.
6. Configure by using the MicroShift **config.yaml** file.

1.1.1. Default settings

If you do not create a **config.yaml** file, default values are used. The following example shows the default configuration settings.

- To see the default values, run the following command:

```
$ microshift show-config
```

Default values example output in YAML form

```
apiServer:
  advertiseAddress: 10.44.0.0/32 (1)
  auditLog:
    maxFileAge: 0 (2)
    maxFileSize: 200 (3)
    maxFiles: 10 (4)
    profile: Default (5)
  namedCertificates:
    - certPath: ""
      keyPath: ""
      names:
        - ""
  subjectAltNames: [] (6)
debugging:
  logLevel: "Normal" (7)
dns:
  baseDomain: microshift.example.com (8)
etcd:
  memoryLimitMB: 0 (9)
ingress:
  listenAddress:
    - "" (10)
  ports: (11)
    http: 80
    https: 443
  routeAdmissionPolicy:
    namespaceOwnership: InterNamespaceAllowed (12)
  status: Managed (13)
manifests: (14)
  kustomizePaths:
    - /usr/lib/microshift/manifests
    - /usr/lib/microshift/manifests.d/*
    - /etc/microshift/manifests
    - /etc/microshift/manifests.d/*
network:
  clusterNetwork:
    - 10.42.0.0/16 (15)
  serviceNetwork:
    - 10.43.0.0/16 (16)
  serviceNodePortRange: 30000-32767 (17)
node:
  hostnameOverride: "" (18)
  nodeIP: "" (19)
```

(1) A string that specifies the IP address from which the API server is advertised to members of the cluster. The default value is calculated based on the address of the service network.

2. How long log files are kept before automatic deletion. The default value of **0** in the **maxFileAge** parameter means a log file is never deleted based on age. This value can be

3. By default, when the **audit.log** file reaches the **maxFileSize** limit, the **audit.log** file is rotated and MicroShift begins writing to a new **audit.log** file. This value can be configured.

4. The total number of log files kept. By default, MicroShift retains 10 log files. The oldest is deleted when an excess file is created. This value can be configured.

5. Logs only metadata for read and write requests; does not log request bodies except for OAuth access token requests. If you do not specify this field, the **Default** profile is used.

6. Subject Alternative Names for API server certificates.

7. Log verbosity. Valid values for this field are **Normal**, **Debug**, **Trace**, or **TraceAll**.

8. By default, **etcd** uses as much memory as needed to handle the load on the system. However, in memory constrained systems, it might be preferred or necessary to limit the amount of memory **etcd** can to use at a given time.

9. Base domain of the cluster. All managed DNS records are subdomains of this base.

10. The **ingress.listenAddress** value defaults to the entire network of the host. The valid configurable value is a list that can be either a single IP address or NIC name or multiple IP addresses and NIC names.

11. Default ports shown. Configurable. Valid values for both port entries are a single, unique port in the 1-65535 range. The values of the **ports.http** and **ports.https** fields cannot be the same.

12. Describes how hostname claims across namespaces are handled. By default, allows routes to claim different paths of the same hostname across namespaces. Valid values are **Strict** and **InterNamespaceAllowed**. Specifying **Strict** prevents routes in different namespaces from claiming the same hostname. If the value is deleted in a customized MicroShift **config.yaml**, the **InterNamespaceAllowed** value is automatically set.

13. Default router status, can be **Managed** or **Removed**.

14. The locations on the file system to scan for **kustomization** files to use to load manifests. Set to a list of paths to scan only those paths. Set to an empty list to disable loading manifests. The entries in the list can be glob patterns to match multiple subdirectories.

15. A block of IP addresses from which pod IP addresses are allocated. This field is immutable after installation.

16. A block of virtual IP addresses for Kubernetes services. IP address pool for services. A single entry is supported. This field is immutable after installation.

17. The port range allowed for Kubernetes services of type **NodePort**. If not specified, the default range of 30000-32767 is used. Services without a **NodePort** specified are automatically allocated one from this range. This parameter can be updated after the cluster is installed.

18. The name of the node. The default value is the hostname. If non-empty, this string is used to identify the node instead of the hostname.

19. The IP address of the node. The default value is the IP address of the default route.

1.2. NETWORK FEATURES

Networking features available with MicroShift 4.16 include:

- Kubernetes network policy
- Dynamic node IP
- Custom gateway interface
- Second gateway interface
- Cluster network on specified host interface
- Blocking external access to NodePort service on specific host interfaces

Networking features not available with MicroShift 4.16:

- Egress IP/firewall/QoS: disabled
- Hybrid networking: not supported
- IPsec: not supported
- Hardware offload: not supported

1.3. IP FORWARD

The host network **sysctl net.ipv4.ip_forward** kernel parameter is automatically enabled by the **ovnkube-master** container when started. This is required to forward incoming traffic to the CNI. For example, accessing the NodePort service from outside of a cluster fails if **ip_forward** is disabled.

1.4. NETWORK PERFORMANCE OPTIMIZATIONS

By default, three performance optimizations are applied to OVS services to minimize resource consumption:

- CPU affinity to **ovs-vswitchd.service** and **ovsdb-server.service**
- **no-mlockall** to **openvswitch.service**
- Limit handler and **revalidator** threads to **ovs-vswitchd.service**

1.5. MICROSHIFT NETWORKING COMPONENTS AND SERVICES

This brief overview describes networking components and their operation in MicroShift. The **microshift-networking** RPM is a package that automatically pulls in any networking-related dependencies and systemd services to initialize networking, for example, the **microshift-ovs-init** systemd service.

NetworkManager

NetworkManager is required to set up the initial gateway bridge on the MicroShift node. The NetworkManager and **NetworkManager-ovs** RPM packages are installed as dependencies to the **microshift-networking** RPM package, which contains the necessary configuration files. NetworkManager in MicroShift uses the **keyfile** plugin and is restarted after installation of the **microshift-networking** RPM package.

microshift-ovs-init

The **microshift-ovs-init.service** is installed by the **microshift-networking** RPM package as a dependent systemd service to **microshift.service**. It is responsible for setting up the OVS gateway bridge.

OVN containers

Two OVN-Kubernetes daemon sets are rendered and applied by MicroShift.

- **ovnkube-master** Includes the **northd**, **nbdb**, **sbdb** and **ovnkube-master** containers.
- **ovnkube-node** The ovnkube-node includes the OVN-Controller container.
  After MicroShift starts, the OVN-Kubernetes daemon sets are deployed in the **openshift-ovn-kubernetes** namespace.

Packaging

OVN-Kubernetes manifests and startup logic are built into MicroShift. The systemd services and configurations included in the **microshift-networking** RPM are:

- **/etc/NetworkManager/conf.d/microshift-nm.conf** for **NetworkManager.service**
- **/etc/systemd/system/ovs-vswitchd.service.d/microshift-cpuaffinity.conf** for **ovs-vswitchd.service**
- **/etc/systemd/system/ovsdb-server.service.d/microshift-cpuaffinity.conf** for **ovs-server.service**
- **/usr/bin/configure-ovs-microshift.sh** for **microshift-ovs-init.service**
- **/usr/bin/configure-ovs.sh** for **microshift-ovs-init.service**
- **/etc/crio/crio.conf.d/microshift-ovn.conf** for the CRI-O service

1.6. BRIDGE MAPPINGS

Bridge mappings allow provider network traffic to reach the physical network. Traffic leaves the provider network and arrives at the **br-int** bridge. A patch port between **br-int** and **br-ex** then allows the traffic to traverse to and from the provider network and the edge network. Kubernetes pods are connected to the **br-int** bridge through virtual ethernet pair: one end of the virtual ethernet pair is attached to the pod namespace, and the other end is attached to the **br-int** bridge.

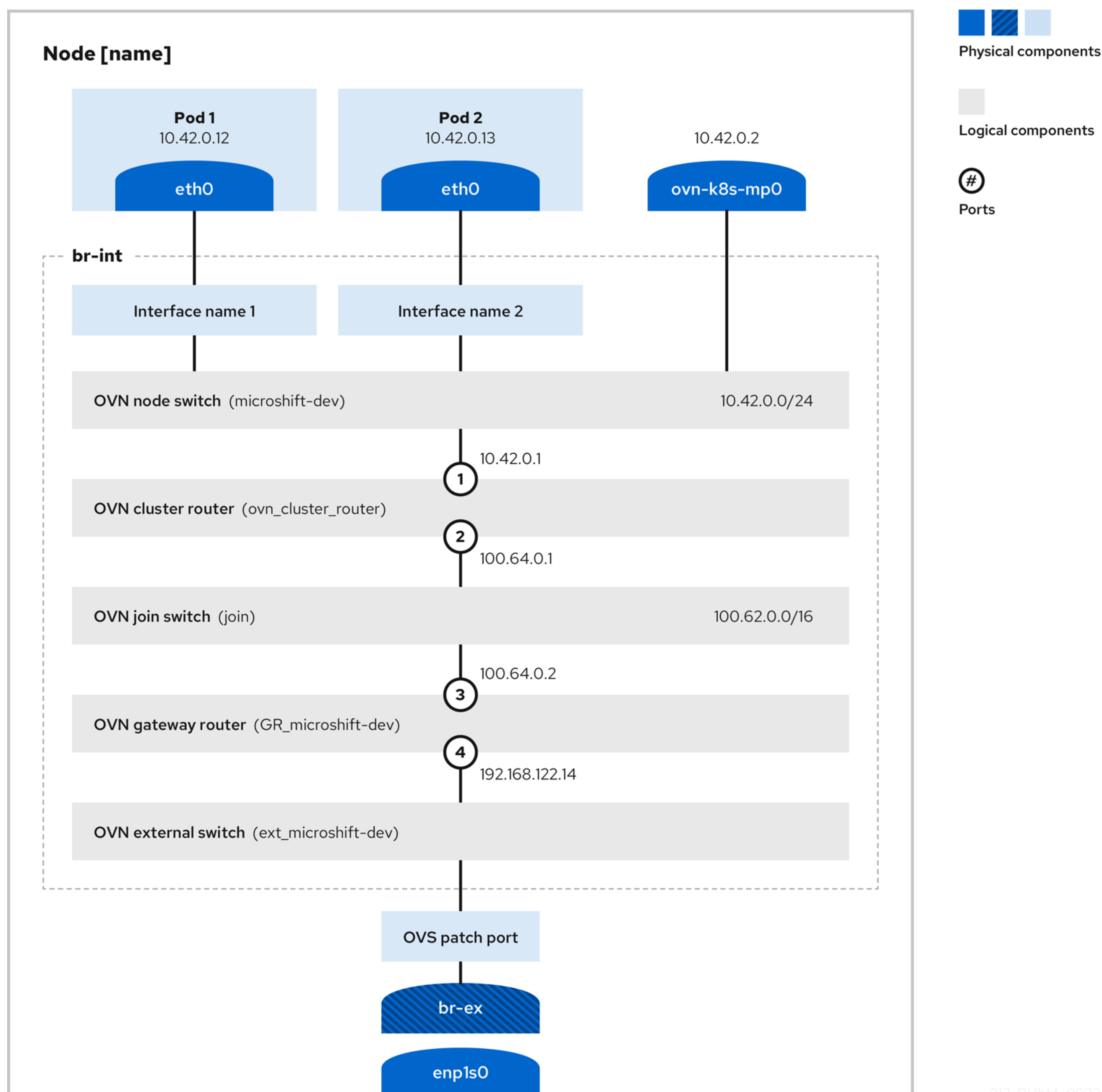
1.7. NETWORK TOPOLOGY

OVN-Kubernetes provides an overlay-based networking implementation. This overlay includes an OVS-based implementation of Service and NetworkPolicy. The overlay network uses the Geneve (Generic Network Virtualization Encapsulation) tunnel protocol. The pod maximum transmission unit (MTU) for the Geneve tunnel is set to the default route MTU if it is not configured.

To configure the MTU, you must set an equal-to or less-than value than the MTU of the physical interface on the host. A less-than value for the MTU makes room for the required information that is added to the tunnel header before it is transmitted.

OVS runs as a systemd service on the MicroShift node. The OVS RPM package is installed as a dependency to the **microshift-networking** RPM package. OVS is started immediately when the **microshift-networking** RPM is installed.

Red Hat build of MicroShift network topology



1.7.1. Description of the OVN logical components of the virtualized network

OVN node switch

A virtual switch named **<node-name>**. The OVN node switch is named according to the hostname of the node.

- In this example, the **node-name** is **microshift-dev**.

OVN cluster router

A virtual router named **ovn_cluster_router**, also known as the distributed router.

- In this example, the cluster network is **10.42.0.0/16**.

OVN join switch

A virtual switch named **join**.

OVN gateway router

A virtual router named **GR_<node-name>**, also known as the external gateway router.

OVN external switch

A virtual switch named **ext_<node-name>.**

1.7.2. Description of the connections in the network topology figure

- The north-south traffic between the network service and the OVN external switch **ext_microshift-dev** is provided through the host kernel by the gateway bridge **br-ex**.
- The OVN gateway router **GR_microshift-dev** is connected to the external network switch **ext_microshift-dev** through the logical router port 4. Port 4 is attached with the node IP address 192.168.122.14.
- The join switch **join** connects the OVN gateway router **GR_microshift-dev** to the OVN cluster router **ovn_cluster_router**. The IP address range is 100.62.0.0/16.
  - The OVN gateway router **GR_microshift-dev** connects to the OVN join switch **join** through the logical router port 3. Port 3 attaches with the internal IP address 100.64.0.2.
  - The OVN cluster router **ovn_cluster_router** connects to the join switch **join** through the logical router port 2. Port 2 attaches with the internal IP address 100.64.0.1.
- The OVN cluster router **ovn_cluster_router** connects to the node switch **microshift-dev** through the logical router port 1. Port 1 is attached with the OVN cluster network IP address 10.42.0.1.
- The east-west traffic between the pods and the network service is provided by the OVN cluster router **ovn_cluster_router** and the node switch **microshift-dev**. The IP address range is 10.42.0.0/24.
- The east-west traffic between pods is provided by the node switch **microshift-dev** without network address translation (NAT).
- The north-south traffic between the pods and the external network is provided by the OVN cluster router **ovn_cluster_router** and the host network. This router is connected through the **ovn-kubernetes** management port **ovn-k8s-mp0**, with the IP address 10.42.0.2.
- All the pods are connected to the OVN node switch through their interfaces.
  - In this example, Pod 1 and Pod 2 are connected to the node switch through **Interface 1** and **Interface 2**.

1.8. ADDITIONAL RESOURCES

- Using a YAML configuration file
- Understanding networking settings
- About using multiple networks
- About network policies

CHAPTER 2. UNDERSTANDING NETWORKING SETTINGS

Learn how to apply networking customization and default settings to MicroShift deployments. Each node is contained to a single machine and single MicroShift, so each deployment requires individual configuration, pods, and settings.

Cluster Administrators have several options for exposing applications that run inside a cluster to external traffic and securing network connections:

- A service such as NodePort
- API resources, such as **Ingress** and **Route**

By default, Kubernetes allocates each pod an internal IP address for applications running within the pod. Pods and their containers can have traffic between them, but clients outside the cluster do not have direct network access to pods except when exposed with a service such as NodePort.

2.1. CREATING AN OVN-KUBERNETES CONFIGURATION FILE

MicroShift uses built-in default OVN-Kubernetes values if an OVN-Kubernetes configuration file is not created. You can write an OVN-Kubernetes configuration file to **/etc/microshift/ovn.yaml**. An example file is provided for your configuration.

Procedure

1. To create your **ovn.yaml** file, run the following command:

```
$ sudo cp /etc/microshift/ovn.yaml.default /etc/microshift/ovn.yaml
```

2. To list the contents of the configuration file you created, run the following command:

```
$ cat /etc/microshift/ovn.yaml
```

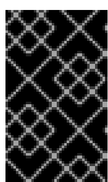
Example YAML file with default maximum transmission unit (MTU) value

```
mtu: 1400
```

3. To customize your configuration, you can change the MTU value. The table that follows provides details:

Table 2.1. Supported optional OVN-Kubernetes configurations for MicroShift

Field	Type	Default	Description	Example
mtu	uint32	auto	MTU value used for the pods	1300



IMPORTANT

If you change the **mtu** configuration value in the **ovn.yaml** file, you must restart the host that Red Hat build of MicroShift is running on to apply the updated setting.

Example custom ovn.yaml configuration file

```
mtu: 1300
```

2.2. RESTARTING THE OVNKUBE-MASTER POD

The following procedure restarts the **ovnkube-master** pod.

Prerequisites

- The OpenShift CLI (**oc**) is installed.
- Access to the cluster as a user with the **cluster-admin** role.
- A cluster installed on infrastructure configured with the OVN-Kubernetes network plugin.
- The KUBECONFIG environment variable is set.

Procedure

Use the following steps to restart the **ovnkube-master** pod.

1. Access the remote cluster by running the following command:

```
$ export KUBECONFIG=$PWD/kubeconfig
```

2. Find the name of the **ovnkube-master** pod that you want to restart by running the following command:

```
$ pod=$(oc get pods -n openshift-ovn-kubernetes | awk -F " " '/ovnkube-master/{print $1}')
```

3. Delete the **ovnkube-master** pod by running the following command:

```
$ oc -n openshift-ovn-kubernetes delete pod $pod
```

4. Confirm that a new **ovnkube-master** pod is running by using the following command:

```
$ oc get pods -n openshift-ovn-kubernetes
```

The listing of the running pods shows a new **ovnkube-master** pod name and age.

2.3. DEPLOYING MICROSHIFT BEHIND AN HTTP OR HTTPS PROXY

Deploy a MicroShift cluster behind an HTTP or HTTPS proxy when you want to add basic anonymity and security measures to your pods.

You must configure the host operating system to use the proxy service with all components initiating HTTP or HTTPS requests when deploying MicroShift behind a proxy.

All the user-specific workloads or pods with egress traffic, such as accessing cloud services, must be configured to use the proxy. There is no built-in transparent proxying of egress traffic in MicroShift.

2.4. USING THE RPM-OSTREE HTTP OR HTTPS PROXY

To use the HTTP or HTTPS proxy in RPM-OStree, you must add a **Service** section to the configuration file and set the **http_proxy environment** variable for the **rpm-ostreed** service.

Procedure

1. Add this setting to the **/etc/systemd/system/rpm-ostreed.service.d/00-proxy.conf** file:

```
[Service]
Environment="http_proxy=http://$PROXY_USER:$PROXY_PASSWORD@$PROXY_SERVE
R:$PROXY_PORT/"
```

2. Next, reload the configuration settings and restart the service to apply your changes.

   a. Reload the configuration settings by running the following command:

   ```
   $ sudo systemctl daemon-reload
   ```

   b. Restart the **rpm-ostreed** service by running the following command:

   ```
   $ sudo systemctl restart rpm-ostreed.service
   ```

2.5. USING A PROXY IN THE CRI-O CONTAINER RUNTIME

To use an HTTP or HTTPS proxy in **CRI-O**, you must add a **Service** section to the configuration file and set the **HTTP_PROXY** and **HTTPS_PROXY** environment variables. You can also set the **NO_PROXY** variable to exclude a list of hosts from being proxied.

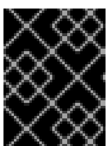
Procedure

1. Create the directory for the configuration file if it does not exist:

```
$ sudo mkdir /etc/systemd/system/crio.service.d/
```

2. Add the following settings to the **/etc/systemd/system/crio.service.d/00-proxy.conf** file:

```
[Service]
Environment=NO_PROXY="localhost,127.0.0.1"
Environment=HTTP_PROXY="http://$PROXY_USER:$PROXY_PASSWORD@$PROXY_SE
RVER:$PROXY_PORT/"
Environment=HTTPS_PROXY="http://$PROXY_USER:$PROXY_PASSWORD@$PROXY_S
ERVER:$PROXY_PORT/"
```



IMPORTANT

You must define the **Service** section of the configuration file for the environment variables or the proxy settings fail to apply.

3. Reload the configuration settings:

```
$ sudo systemctl daemon-reload
```

4. Restart the CRI-O service:

```
$ sudo systemctl restart crio
```

5. Restart the MicroShift service to apply the settings:

```
$ sudo systemctl restart microshift
```

Verification

1. Verify that pods are started by running the following command and examining the output:

```
$ oc get all -A
```

2. Verify that MicroShift is able to pull container images by running the following command and examining the output:

```
$ sudo crictl images
```

2.6. GETTING A SNAPSHOT OF OVS INTERFACES FROM A RUNNING CLUSTER

A snapshot represents the state and data of OVS interfaces at a specific point in time.

Procedure

- To see a snapshot of OVS interfaces from a running MicroShift cluster, use the following command:

```
$ sudo ovs-vsctl show
```

Example OVS interfaces in a running cluster

```
9d9f5ea2-9d9d-4e34-bbd2-dbac154fdc93
    Bridge br-ex
        Port br-ex
            Interface br-ex
                type: internal
        Port patch-br-ex_localhost.localdomain-to-br-int 1
            Interface patch-br-ex_localhost.localdomain-to-br-int
                type: patch
                options: {peer=patch-br-int-to-br-ex_localhost.localdomain} 2
    Bridge br-int
        fail_mode: secure
        datapath_type: system
        Port patch-br-int-to-br-ex_localhost.localdomain
            Interface patch-br-int-to-br-ex_localhost.localdomain
                type: patch
                options: {peer=patch-br-ex_localhost.localdomain-to-br-int}
        Port eebee1ce5568761
            Interface eebee1ce5568761 3
        Port b47b1995ada84f4
```

```
        Interface b47b1995ada84f4 4
      Port "3031f43d67c167f"
        Interface "3031f43d67c167f" 5
      Port br-int
        Interface br-int
          type: internal
      Port ovn-k8s-mp0 6
        Interface ovn-k8s-mp0
          type: internal
  ovs_version: "2.17.3"
```

1. The **patch-br-ex_localhost.localdomain-to-br-int** and **patch-br-int-to-br-ex_localhost.localdomain** are OVS patch ports that connect **br-ex** and **br-int**.
2. The **patch-br-ex_localhost.localdomain-to-br-int** and **patch-br-int-to-br-ex_localhost.localdomain** are OVS patch ports that connect **br-ex** and **br-int**.
3. The pod interface **eebee1ce5568761** is named with the first 15 bits of the pod sandbox ID and is plugged into the **br-int** bridge.
4. The pod interface **b47b1995ada84f4** is named with the first 15 bits of the pod sandbox ID and is plugged into the **br-int** bridge.
5. The pod interface **3031f43d67c167f** is named with the first 15 bits of the pod sandbox ID and is plugged into the **br-int** bridge.
6. The OVS internal port for hairpin traffic,**ovn-k8s-mp0** is created by the **ovnkube-master** container.

2.7. THE MICROSHIFT LOADBALANCER SERVICE FOR WORKLOADS

MicroShift has a built-in implementation of network load balancers that you can use for your workloads and applications within the cluster. You can create a **LoadBalancer** service by configuring a pod to interpret ingress rules and serve as an ingress controller. The following procedure gives an example of a deployment-based **LoadBalancer** service.

2.8. DEPLOYING A LOAD BALANCER FOR AN APPLICATION

The following example procedure uses the node IP address as the external IP address for the **LoadBalancer** service configuration file. Use this example as guidance for how to deploy load balancers.

Prerequisites

- The OpenShift CLI (**oc**) is installed.
- You installed a cluster on an infrastructure configured with the OVN-Kubernetes network plugin.
- The **KUBECONFIG** environment variable is set.

Procedure

1. Verify that your pods are running by entering the following command:

```
$ oc get pods -A
```

Example output

```
NAMESPACE                          NAME                                                    READY   STATUS
RESTARTS  AGE
default                         i-06166fbb376f14a8bus-west-2computeinternal-debug-qtwcr  1/1
Running   0     46m
kube-system                        csi-snapshot-controller-5c6586d546-lprv4               1/1
Running   0     51m
kube-system                        csi-snapshot-webhook-6bf8ddc7f5-kz6k9                   1/1
Running   0     51m
openshift-dns                      dns-default-45jl7                                       2/2     Running   0
50m
openshift-dns                      node-resolver-7wmzf                                     1/1     Running   0
51m
openshift-ingress                  router-default-78b86fbf9d-qvj9s                         1/1     Running
0     51m
openshift-multus                   dhcp-daemon-j7qnf                                       1/1     Running   0
51m
openshift-multus                   multus-r758z                                            1/1     Running   0
51m
openshift-operator-lifecycle-manager catalog-operator-85fb86fcb9-t6zm7                     1/1
Running   0     51m
openshift-operator-lifecycle-manager olm-operator-87656d995-fvz84                          1/1
Running   0     51m
openshift-ovn-kubernetes           ovnkube-master-5rfhh                                    4/4     Running
0     51m
openshift-ovn-kubernetes           ovnkube-node-gcnt6                                      1/1     Running
0     51m
openshift-service-ca               service-ca-bf5b7c9f8-pn6rk                              1/1     Running
0     51m
openshift-storage                  topolvm-controller-549f7fbdd5-7vrmv                     5/5
Running   0     51m
openshift-storage                  topolvm-node-rht2m                                      3/3     Running   0
50m
```

2. Create a namespace by running the following commands:

```
$ NAMESPACE=<nginx-lb-test> (1)
```

(1) Replace _<nginx-lb-test> with the application namespace that you want to create.

```
$ oc create ns $NAMESPACE
```

Example namespace

The following example deploys three replicas of the test **nginx** application in the created namespace:

```
oc apply -n $NAMESPACE -f - <<EOF
apiVersion: v1
kind: ConfigMap
```

```
metadata:
  name: nginx
data:
  headers.conf: |
    add_header X-Server-IP  \$server_addr always;
---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx
spec:
  replicas: 3
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - image: quay.io/packit/nginx-unprivileged
        imagePullPolicy: Always
        name: nginx
        ports:
        - containerPort: 8080
        volumeMounts:
        - name: nginx-configs
          subPath: headers.conf
          mountPath: /etc/nginx/conf.d/headers.conf
        securityContext:
          allowPrivilegeEscalation: false
          seccompProfile:
            type: RuntimeDefault
          capabilities:
            drop: ["ALL"]
          runAsNonRoot: true
      volumes:
        - name: nginx-configs
          configMap:
            name: nginx
            items:
              - key: headers.conf
                path: headers.conf
EOF
```

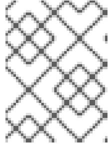
3. You can verify that the three sample replicas started successfully by running the following command:

```
$ oc get pods -n $NAMESPACE
```

4. Create a **LoadBalancer** service for the **nginx** test application by running the following command:

```
oc create -n $NAMESPACE -f - <<EOF
```

```
apiVersion: v1
kind: Service
metadata:
  name: nginx
spec:
  ports:
  - port: 81
    targetPort: 8080
  selector:
    app: nginx
  type: LoadBalancer
EOF
```



NOTE

You must ensure that the **port** parameter is a host port that is not occupied by other **LoadBalancer** services or MicroShift components.

5. Verify that the service file exists, that the external IP address is properly assigned, and that the external IP is identical to the node IP by running the following command:

```
$ oc get svc -n $NAMESPACE
```

Example output

```
NAME    TYPE           CLUSTER-IP      EXTERNAL-IP     PORT(S)        AGE
nginx   LoadBalancer   10.43.183.104   192.168.1.241   81:32434/TCP   2m
```

Verification

The following command forms five connections to the example **nginx** application using the external IP address of the **LoadBalancer** service configuration. The result of the command is a list of those server IP addresses.

- Verify that the load balancer sends requests to all the running applications by running the following command:

```
EXTERNAL_IP=192.168.1.241
seq 5 | xargs -Iz curl -s -I http://$EXTERNAL_IP:81 | grep X-Server-IP
```

The output of the previous command contains different IP addresses if the **LoadBalancer** service is successfully distributing the traffic to the applications, for example:

Example output

```
X-Server-IP: 10.42.0.41
X-Server-IP: 10.42.0.41
X-Server-IP: 10.42.0.43
X-Server-IP: 10.42.0.41
X-Server-IP: 10.42.0.43
```

2.9. BLOCKING EXTERNAL ACCESS TO THE NODEPORT SERVICE ON A SPECIFIC HOST INTERFACE

OVN-Kubernetes does not restrict the host interface where a NodePort service can be accessed from outside a Red Hat build of MicroShift node. The following procedure explains how to block the NodePort service on a specific host interface and restrict external access.

Prerequisites

- You must have an account with root privileges.

Procedure

1. Change the **NODEPORT** variable to the host port number assigned to your Kubernetes NodePort service by running the following command:

```
# export NODEPORT=30700
```

2. Change the **INTERFACE_IP** value to the IP address from the host interface that you want to block. For example:

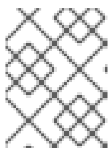
```
# export INTERFACE_IP=192.168.150.33
```

3. Insert a new rule in the **nat** table PREROUTING chain to drop all packets that match the destination port and IP address. For example:

```
$ sudo nft -a insert rule ip nat PREROUTING tcp dport $NODEPORT ip daddr
$INTERFACE_IP drop
```

4. List the new rule by running the following command:

```
$ sudo nft -a list chain ip nat PREROUTING
table ip nat {
 chain PREROUTING { # handle 1
  type nat hook prerouting priority dstnat; policy accept;
  tcp dport 30700 ip daddr 192.168.150.33 drop # handle 134
  counter packets 108 bytes 18074 jump OVN-KUBE-ETP # handle 116
  counter packets 108 bytes 18074 jump OVN-KUBE-EXTERNALIP # handle 114
  counter packets 108 bytes 18074 jump OVN-KUBE-NODEPORT # handle 112
 }
}
```



NOTE

Note the **handle** number of the newly added rule. You need to remove the **handle** number in the following step.

5. Remove the custom rule with the following sample command:

```
$ sudo nft -a delete rule ip nat PREROUTING handle 134
```

2.10. THE MULTICAST DNS PROTOCOL

You can use the multicast DNS protocol (mDNS) to allow name resolution and service discovery within a Local Area Network (LAN) using multicast exposed on the **5353/UDP** port.

MicroShift includes an embedded mDNS server for deployment scenarios in which the authoritative DNS server cannot be reconfigured to point clients to services on MicroShift. The embedded DNS server allows **.local** domains exposed by MicroShift to be discovered by other elements on the LAN.

2.11. AUDITING EXPOSED NETWORK PORTS

On MicroShift, the host port can be opened by a workload in the following cases. You can check logs to view the network services.

2.11.1. hostNetwork

When a pod is configured with the **hostNetwork:true** setting, the pod is running in the host network namespace. This configuration can independently open host ports. MicroShift component logs cannot be used to track this case, the ports are subject to firewalld rules. If the port opens in firewalld, you can view the port opening in the firewalld debug log.

Prerequisites

- You have root user access to your build host.

Procedure

1. Optional: You can check that the **hostNetwork:true** parameter is set in your ovnkube-node pod by using the following example command:

```
$ sudo oc get pod -n openshift-ovn-kubernetes <ovnkube-node-pod-name> -o json | jq -r
'.spec.hostNetwork' true
```

2. Enable debug in the firewalld log by running the following command:

```
$ sudo vi /etc/sysconfig/firewalld
FIREWALLD_ARGS=--debug=10
```

3. Restart the firewalld service:

```
$ sudo systemctl restart firewalld.service
```

4. To verify that the debug option was added properly, run the following command:

```
$ sudo systemd-cgls -u firewalld.service
```

The firewalld debug log is stored in the **/var/log/firewalld** path.

Example logs for when the port open rule is added:

```
2023-06-28 10:46:37 DEBUG1: config.getZoneByName('public')
2023-06-28 10:46:37 DEBUG1: config.zone.7.addPort('8080', 'tcp')
2023-06-28 10:46:37 DEBUG1: config.zone.7.getSettings()
```

```
2023-06-28 10:46:37 DEBUG1: config.zone.7.update('...')
2023-06-28 10:46:37 DEBUG1: config.zone.7.Updated('public')
```

Example logs for when the port open rule is removed:

```
2023-06-28 10:47:57 DEBUG1: config.getZoneByName('public')
2023-06-28 10:47:57 DEBUG2: config.zone.7.Introspect()
2023-06-28 10:47:57 DEBUG1: config.zone.7.removePort('8080', 'tcp')
2023-06-28 10:47:57 DEBUG1: config.zone.7.getSettings()
2023-06-28 10:47:57 DEBUG1: config.zone.7.update('...')
2023-06-28 10:47:57 DEBUG1: config.zone.7.Updated('public')
```

2.11.2. hostPort

You can access the hostPort setting logs in MicroShift. The following logs are examples for the hostPort setting:

Procedure

- You can access the logs by running the following command:

```
$ journalctl -u crio | grep "local port"
```

Example CRI-O logs when the host port is opened:

```
$ Jun 25 16:27:37 rhel92 crio[77216]: time="2023-06-25 16:27:37.033003098+08:00"
level=info msg="Opened local port tcp:443"
```

Example CRI-O logs when the host port is closed:

```
$ Jun 25 16:24:11 rhel92 crio[77216]: time="2023-06-25 16:24:11.342088450+08:00"
level=info msg="Closing host port tcp:443"
```

2.11.3. NodePort and LoadBalancer services

OVN-Kubernetes opens host ports for **NodePort** and **LoadBalancer** service types. These services add iptables rules that take the ingress traffic from the host port and forwards it to the clusterIP. Logs for the **NodePort** and **LoadBalancer** services are presented in the following examples:

Procedure

1. To access the name of your **ovnkube-master** pods, run the following command:

```
$ oc get pods -n openshift-ovn-kubernetes | awk '/ovnkube-master/{print $1}'
```

Example ovnkube-master pod name

```
ovnkube-master-n2shv
```

2. You can access the **NodePort** and **LoadBalancer** services logs using your **ovnkube-master** pod and running the following example command:

```
$ oc logs -n openshift-ovn-kubernetes <ovnkube-master-pod-name> ovnkube-master | grep -
E "OVN-KUBE-NODEPORT|OVN-KUBE-EXTERNALIP"
```

NodePort service:

Example logs in the ovnkube-master container of the ovnkube-master pod when a host port is open:

```
$ I0625 09:07:00.992980 2118395 iptables.go:27] Adding rule in table: nat, chain: OVN-
KUBE-NODEPORT with args: "-p TCP -m addrtype --dst-type LOCAL --dport 32718 -j DNAT
--to-destination 10.96.178.142:8081" for protocol: 0
```

Example logs in the ovnkube-master container of the ovnkube-master pod when a host port is closed:

```
$ Deleting rule in table: nat, chain: OVN-KUBE-NODEPORT with args: "-p TCP -m addrtype -
-dst-type LOCAL --dport 32718 -j DNAT --to-destination 10.96.178.142:8081" for protocol: 0
```

LoadBalancer service:

Example logs in the ovnkube-master container of the ovnkube-master pod when a host port is open:

```
$ I0625 09:34:10.406067  128902 iptables.go:27] Adding rule in table: nat, chain: OVN-
KUBE-EXTERNALIP with args: "-p TCP -d 172.16.47.129 --dport 8081 -j DNAT --to-
destination 10.43.114.94:8081" for protocol: 0
```

Example logs in the ovnkube-master container of the ovnkube-master pod when a host port is closed:

```
$ I0625 09:37:00.976953  128902 iptables.go:63] Deleting rule in table: nat, chain: OVN-
KUBE-EXTERNALIP with args: "-p TCP -d 172.16.47.129 --dport 8081 -j DNAT --to-
destination 10.43.114.94:8081" for protocol: 0
```

CHAPTER 3. UNDERSTANDING AND CONFIGURING THE ROUTER

Learn about default and custom settings for configuring the router and route admission policy with MicroShift.

3.1. ABOUT CONFIGURING THE ROUTER

To make ingress optional, you can configure MicroShift ingress router settings to manage which ports, if any, are exposed to network traffic. Specified routing is an example of ingress load balancing.

- The default ingress router is always on, running on all IP addresses on the **http: 80** and **https: 443** ports.
- Default router settings allow access to any namespace.

Some applications running on top of MicroShift might not require the default router and instead create their own. You can configure the router to control both ingress and namespace access.

TIP

You can check for the presence of the default router in your MicroShift installation before you begin configurations by using the **oc get deployment -n openshift-ingress** command, which returns the following output:

```
NAME           READY   UP-TO-DATE   AVAILABLE   AGE
router-default  1/1     1            1           2d23h
```

3.1.1. Router settings and valid values

The ingress router settings consist of the following parameters and valid values:

Example config.yaml router settings

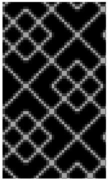
```
# ...
ingress:
  listenAddress:
    - "" (1)
  ports: (2)
    http: 80
    https: 443
  routeAdmissionPolicy:
    namespaceOwnership: InterNamespaceAllowed (3)
  status: Managed (4)
# ...
```

1. The **ingress.listenAddress** value defaults to the entire network of the host. Valid customizable values can be a single IP address or host name or a list of IP addresses or host names.

2. Valid values for both port entries are a single, unique port in the 1-65535 range. The values of the **ports.http** and **ports.https** fields cannot be the same.

3. Default value. Allows routes to claim different paths of the same host name across namespaces.

4. Default value. **Managed** is required for the ingress ports to remain open.



IMPORTANT

The firewalld service is bypassed by the default MicroShift router and by configurations that enable the router. Ingress and egress must be controlled by setting network policies when the router is active.

3.2. DISABLING THE ROUTER

In use cases such as industrial IoT spaces where MicroShift pods only need to connect to southbound operational systems and northbound cloud-data systems, inbound services are not needed. Use this procedure to disable the router in such egress-only use cases.

Prerequisites

- You installed MicroShift.
- You created a MicroShift **config.yaml** file.
- The OpenShift CLI (**oc**) is installed.

TIP

If you complete all the configurations that you need to make in the MicroShift **config.yaml** file at the same time, you can minimize system restarts.

Procedure

1. Update the value of **ingress.status** field to **Removed** in the MicroShift **config.yaml** file as shown in the following example:

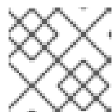
   Example config.yaml ingress stanza

```
# ...
ingress:
  ports:
    http: 80
    https: 443
  routeAdmissionPolicy:
    namespaceOwnership: InterNamespaceAllowed
  status: Removed (1)
# ...
```

   1. When the value is set to **Removed**, the ports listed in **ingress.ports** are automatically closed. Any other settings in the **ingress** stanza are ignored, for example, any values in the **routeAdmissionPolicy.namespaceOwnership** field.

2. Restart the MicroShift service by running the following command:

```
$ sudo systemctl restart microshift
```



NOTE

The MicroShift service outputs current configurations during restarts.

Verification

- After the system restarts, verify that the router has been removed and that ingress is stopped by running the following command:

  ```
  $ oc -n openshift-ingress get svc
  ```

  Expected output

  ```
  No resources found in openshift-ingress namespace.
  ```

3.3. CONFIGURING ROUTER INGRESS

If your MicroShift applications need to listen only for data traffic, you can configure the **listenAddress** setting to isolate your devices. You can also configure specific ports and IP addresses for network connections. Use the combination required to customize the endpoint configuration for your use case.

3.3.1. Configuring router ports

You can control which ports your devices use by configuring the router ingress fields.

Prerequisites

- You installed MicroShift.
- You created a MicroShift **config.yaml** file.
- The OpenShift CLI (**oc**) is installed.

TIP

If you complete all the configurations that you need to make in the MicroShift **config.yaml** file at the same time, you can minimize system restarts.

Procedure

1. Update the MicroShift **config.yaml** port values in the **ingress.ports.http** and **ingress.ports.https** fields to the ports you want to use:

   Example config.yaml router settings

   ```
   # ...
   ingress:
     ports: (1)
       http: 80
       https: 443
   ```

```
  routeAdmissionPolicy:
    namespaceOwnership: InterNamespaceAllowed
  status: Managed (2)
# ...
```

(1) Default ports shown. Customizable. Valid values for both port entries are a single, unique port in the 1-65535 range. The values of the **ports.http** and **ports.https** fields cannot be the same.

(2) The default value. **Managed** is required for the ingress ports to remain open.

2. Restart the MicroShift service by running the following command:

```
$ sudo systemctl restart microshift
```

3.3.2. Configuring router IP addresses

You can restrict the network traffic to the router by configuring specific IP addresses. For example:

- Use cases where the router is reachable only on internal networks, but not on northbound public networks
- Use cases where the router is reachable only by northbound public networks, but not on internal networks
- Use cases where the router is reachable by both internal networks and northbound public networks, but on separate IP addresses

Prerequisites

- You installed MicroShift.
- You created a MicroShift **config.yaml** file.
- The OpenShift CLI (**oc**) is installed.

TIP

If you complete all the configurations that you need to make in the MicroShift **config.yaml** file at the same time, you can minimize system restarts.

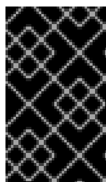
Procedure

1. Update the list in the **ingress.listenAddress** field in the MicroShift **config.yaml** according to your requirements and as shown in the following examples:

Default router IP address list

```
# ...
ingress:
  listenAddress:
    - "<host_network>" (1)
# ...
```

1. The **ingress.listenAddress** value defaults to the entire network of the host. To continue to use the default list, remove the **listen.Address** field from the MicroShift **config.yaml** file. To customize this parameter, use a list. The list can contain either a single IP address or NIC name or multiple IP addresses and NIC names.



IMPORTANT

You must either remove the **listenAddress** parameter or add values to it in the form of a list when using the **config.yaml** file. Do not leave the field empty or MicroShift crashes on restart.

Example router setting with a single host IP address

```
# ...
ingress:
  listenAddress:
    - 10.2.1.100
# ...
```

Example router setting with a combination of IP addresses and NIC names

```
# ...
ingress:
  listenAddress:
    - 10.2.1.100
    - 10.2.2.10
    - ens3
# ...
```

2. Restart the MicroShift service by running the following command:

```
$ sudo systemctl restart microshift
```

Verification

- To verify that your settings are applied, make sure that the **ingress.listenAddress** IP addresses are reachable, then you can **curl** the route with the destination to one of these load balancer IP address.

3.4. ADDITIONAL RESOURCES

- Default settings (MicroShift)
- About network policies

3.5. CONFIGURING THE ROUTE ADMISSION POLICY

By default, MicroShift allows routes in multiple namespaces to use the same hostname. You can prevent routes from claiming the same hostname in different namespaces by configuring the route admission policy.

Prerequisites

Prerequisites

- You installed MicroShift.
- You created a MicroShift **config.yaml** file.
- You installed the OpenShift CLI (**oc**).

TIP

If you complete all the configurations that you need to make in the MicroShift **config.yaml** file at the same time, you can minimize system restarts.

Procedure

1. To prevent routes in different namespaces from claiming the same hostname, update the **namespaceOwnership** field value to **Strict** in the MicroShift **config.yaml** file. See the following example:

   Example config.yaml route admission policy

   ```
   # ...
   ingress:
     routeAdmissionPolicy:
       namespaceOwnership: Strict (1)
   # ...
   ```

   (1) Prevents routes in different namespaces from claiming the same host. Valid values are **Strict** and **InterNamespaceAllowed**. If you delete the value in a customized **config.yaml**, the **InterNamespaceAllowed** value is set automatically.

2. To apply the configuration, restart the MicroShift service by running the following command:

   ```
   $ sudo systemctl restart microshift
   ```

CHAPTER 4. NETWORK POLICIES

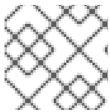
4.1. ABOUT NETWORK POLICIES

Learn how network policies work for MicroShift to restrict or allow network traffic to pods in your cluster.

4.1.1. How network policy works in MicroShift

In a cluster using the default OVN-Kubernetes Container Network Interface (CNI) plugin for MicroShift, network isolation is controlled by both firewalld, which is configured on the host, and by **NetworkPolicy** objects created within MicroShift. Simultaneous use of firewalld and **NetworkPolicy** is supported.

- Network policies work only within boundaries of OVN-Kubernetes-controlled traffic, so they can apply to every situation except for **hostPort/hostNetwork** enabled pods.
- Firewalld settings also do not apply to **hostPort/hostNetwork** enabled pods.



NOTE

Firewalld rules run before any **NetworkPolicy** is enforced.

WARNING

Network policy does not apply to the host network namespace. Pods with host networking enabled are unaffected by network policy rules. However, pods connecting to the host-networked pods might be affected by the network policy rules.

Network policies cannot block traffic from localhost.

By default, all pods in a MicroShift node are accessible from other pods and network endpoints. To isolate one or more pods in a cluster, you can create **NetworkPolicy** objects to indicate allowed incoming connections. You can create and delete **NetworkPolicy** objects.

If a pod is matched by selectors in one or more **NetworkPolicy** objects, then the pod accepts only connections that are allowed by at least one of those **NetworkPolicy** objects. A pod that is not selected by any **NetworkPolicy** objects is fully accessible.

A network policy applies to only the TCP, UDP, ICMP, and SCTP protocols. Other protocols are not affected.

The following example **NetworkPolicy** objects demonstrate supporting different scenarios:

- Deny all traffic:
  To make a project deny by default, add a **NetworkPolicy** object that matches all pods but accepts no traffic:

```
kind: NetworkPolicy
apiVersion: networking.k8s.io/v1
```

```
metadata:
  name: deny-by-default
spec:
  podSelector: {}
  ingress: []
```

- Allow connections from the default router, which is the ingress in MicroShift:
  To allow connections from the MicroShift default router, add the following **NetworkPolicy** object:

```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: allow-from-openshift-ingress
spec:
  ingress:
  - from:
    - namespaceSelector:
        matchLabels:
          ingresscontroller.operator.openshift.io/deployment-ingresscontroller: default
  podSelector: {}
  policyTypes:
  - Ingress
```

- Only accept connections from pods within the same namespace:
  To make pods accept connections from other pods in the same namespace, but reject all other connections from pods in other namespaces, add the following **NetworkPolicy** object:

```
kind: NetworkPolicy
apiVersion: networking.k8s.io/v1
metadata:
  name: allow-same-namespace
spec:
  podSelector: {}
  ingress:
  - from:
    - podSelector: {}
```

- Only allow HTTP and HTTPS traffic based on pod labels:
  To enable only HTTP and HTTPS access to the pods with a specific label (**role=frontend** in following example), add a **NetworkPolicy** object similar to the following:

```
kind: NetworkPolicy
apiVersion: networking.k8s.io/v1
metadata:
  name: allow-http-and-https
spec:
  podSelector:
    matchLabels:
      role: frontend
  ingress:
  - ports:
    - protocol: TCP
```

```
      port: 80
    - protocol: TCP
      port: 443
```

- Accept connections by using both namespace and pod selectors:
  To match network traffic by combining namespace and pod selectors, you can use a **NetworkPolicy** object similar to the following:

```
kind: NetworkPolicy
apiVersion: networking.k8s.io/v1
metadata:
  name: allow-pod-and-namespace-both
spec:
  podSelector:
    matchLabels:
      name: test-pods
  ingress:
    - from:
      - namespaceSelector:
          matchLabels:
            project: project_name
        podSelector:
          matchLabels:
            name: test-pods
```

NetworkPolicy objects are additive, which means you can combine multiple **NetworkPolicy** objects together to satisfy complex network requirements.

For example, for the **NetworkPolicy** objects defined in previous examples, you can define both **allow-same-namespace** and **allow-http-and-https** policies. That configuration allows the pods with the label **role=frontend** to accept any connection allowed by each policy. That is, connections on any port from pods in the same namespace, and connections on ports **80** and **443** from pods in any namespace.

4.1.2. Optimizations for network policy with OVN-Kubernetes network plugin

When designing your network policy, refer to the following guidelines:

- For network policies with the same **spec.podSelector** spec, it is more efficient to use one network policy with multiple **ingress** or **egress** rules, than multiple network policies with subsets of **ingress** or **egress** rules.

- Every **ingress** or **egress** rule based on the **podSelector** or **namespaceSelector** spec generates the number of OVS flows proportional to **number of pods selected by network policy + number of pods selected by ingress or egress rule**. Therefore, it is preferable to use the **podSelector** or **namespaceSelector** spec that can select as many pods as you need in one rule, instead of creating individual rules for every pod.
  For example, the following policy contains two rules:

```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: test-network-policy
spec:
  podSelector: {}
  ingress:
```

```
  - from:
    - podSelector:
        matchLabels:
          role: frontend
  - from:
    - podSelector:
        matchLabels:
          role: backend
```

The following policy expresses those same two rules as one:

```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: test-network-policy
spec:
  podSelector: {}
  ingress:
  - from:
    - podSelector:
        matchExpressions:
        - {key: role, operator: In, values: [frontend, backend]}
```

The same guideline applies to the **spec.podSelector** spec. If you have the same **ingress** or **egress** rules for different network policies, it might be more efficient to create one network policy with a common **spec.podSelector** spec. For example, the following two policies have different rules:

```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: policy1
spec:
  podSelector:
    matchLabels:
      role: db
  ingress:
  - from:
    - podSelector:
        matchLabels:
          role: frontend
---
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: policy2
spec:
  podSelector:
    matchLabels:
      role: client
  ingress:
  - from:
    - podSelector:
        matchLabels:
          role: frontend
```

The following network policy expresses those same two rules as one:

```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: policy3
spec:
  podSelector:
    matchExpressions:
    - {key: role, operator: In, values: [db, client]}
  ingress:
  - from:
    - podSelector:
        matchLabels:
          role: frontend
```

You can apply this optimization when only multiple selectors are expressed as one. In cases where selectors are based on different labels, it may not be possible to apply this optimization. In those cases, consider applying some new labels for network policy optimization specifically.

4.2. CREATING NETWORK POLICIES

You can create a network policy for a namespace.

4.2.1. Example NetworkPolicy object

The following annotates an example NetworkPolicy object:

```
kind: NetworkPolicy
apiVersion: networking.k8s.io/v1
metadata:
  name: allow-27107 (1)
spec:
  podSelector: (2)
    matchLabels:
      app: mongodb
  ingress:
  - from:
    - podSelector: (3)
        matchLabels:
          app: app
    ports: (4)
    - protocol: TCP
      port: 27017
```

1. The name of the NetworkPolicy object.
2. A selector that describes the pods to which the policy applies.
3. A selector that matches the pods from which the policy object allows ingress traffic. The selector matches pods in the same namespace as the NetworkPolicy.
4. A list of one or more destination ports on which to accept traffic.

4.2.2. Creating a network policy using the CLI

To define granular rules describing ingress or egress network traffic allowed for namespaces in your cluster, you can create a network policy.

Prerequisites

- You installed the OpenShift CLI (**oc**).
- You are working in the namespace that the network policy applies to.

Procedure

1. Create a policy rule:
   a. Create a **<policy_name>.yaml** file:

```
$ touch <policy_name>.yaml
```

where:

<policy_name>
Specifies the network policy file name.

   b. Define a network policy in the file that you just created, such as in the following examples:

Deny ingress from all pods in all namespaces

This is a fundamental policy, blocking all cross-pod networking other than cross-pod traffic allowed by the configuration of other Network Policies.

```
kind: NetworkPolicy
apiVersion: networking.k8s.io/v1
metadata:
  name: deny-by-default
spec:
  podSelector: {}
  policyTypes:
  - Ingress
  ingress: []
```

Allow ingress from all pods in the same namespace

```
kind: NetworkPolicy
apiVersion: networking.k8s.io/v1
metadata:
  name: allow-same-namespace
spec:
  podSelector:
  ingress:
  - from:
    - podSelector: {}
```

Allow ingress traffic to one pod from a particular namespace

This policy allows traffic to pods labelled **pod-a** from pods running in **namespace-y**.

```
kind: NetworkPolicy
apiVersion: networking.k8s.io/v1
metadata:
  name: allow-traffic-pod
spec:
  podSelector:
   matchLabels:
      pod: pod-a
  policyTypes:
  - Ingress
  ingress:
  - from:
    - namespaceSelector:
        matchLabels:
           kubernetes.io/metadata.name: namespace-y
```

2. To create the network policy object, enter the following command:

```
$ oc apply -f <policy_name>.yaml -n <namespace>
```

where:

<policy_name>
: Specifies the network policy file name.

<namespace>
: Optional: Specifies the namespace if the object is defined in a different namespace than the current namespace.

Example output

```
networkpolicy.networking.k8s.io/deny-by-default created
```

4.2.3. Creating a default deny all network policy

This is a fundamental policy, blocking all cross-pod networking other than network traffic allowed by the configuration of other deployed network policies. This procedure enforces a default **deny-by-default** policy.

Prerequisites

- You installed the OpenShift CLI (**oc**).
- You are working in the namespace that the network policy applies to.

Procedure

1. Create the following YAML that defines a **deny-by-default** policy to deny ingress from all pods in all namespaces. Save the YAML in the **deny-by-default.yaml** file:

```
kind: NetworkPolicy
apiVersion: networking.k8s.io/v1
metadata:
  name: deny-by-default
  namespace: default 1
spec:
  podSelector: {} 2
  ingress: [] 3
```

1. **namespace: default** deploys this policy to the **default** namespace.
2. **podSelector:** is empty, this means it matches all the pods. Therefore, the policy applies to all pods in the default namespace.
3. There are no **ingress** rules specified. This causes incoming traffic to be dropped to all pods.

2. Apply the policy by entering the following command:

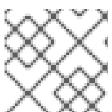
```
$ oc apply -f deny-by-default.yaml
```

Example output

```
networkpolicy.networking.k8s.io/deny-by-default created
```

4.2.4. Creating a network policy to allow traffic from external clients

With the **deny-by-default** policy in place you can proceed to configure a policy that allows traffic from external clients to a pod with the label **app=web**.



NOTE

Firewalld rules run before any **NetworkPolicy** is enforced.

Follow this procedure to configure a policy that allows external service from the public Internet directly or by using a Load Balancer to access the pod. Traffic is only allowed to a pod with the label **app=web**.

Prerequisites

- You installed the OpenShift CLI (**oc**).
- You are working in the namespace that the network policy applies to.

Procedure

1. Create a policy that allows traffic from the public Internet directly or by using a load balancer to access the pod. Save the YAML in the **web-allow-external.yaml** file:

```
kind: NetworkPolicy
apiVersion: networking.k8s.io/v1
metadata:
```

```
  name: web-allow-external
  namespace: default
spec:
  policyTypes:
  - Ingress
  podSelector:
    matchLabels:
      app: web
  ingress:
    - {}
```

2. Apply the policy by entering the following command:

```
$ oc apply -f web-allow-external.yaml
```

Example output

```
networkpolicy.networking.k8s.io/web-allow-external created
```

4.2.5. Creating a network policy allowing traffic to an application from all namespaces

Follow this procedure to configure a policy that allows traffic from all pods in all namespaces to a particular application.

Prerequisites

- You installed the OpenShift CLI (**oc**).
- You are working in the namespace that the network policy applies to.

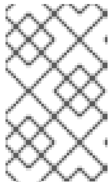
Procedure

1. Create a policy that allows traffic from all pods in all namespaces to a particular application. Save the YAML in the **web-allow-all-namespaces.yaml** file:

```
kind: NetworkPolicy
apiVersion: networking.k8s.io/v1
metadata:
  name: web-allow-all-namespaces
  namespace: default
spec:
  podSelector:
    matchLabels:
      app: web (1)
  policyTypes:
  - Ingress
  ingress:
  - from:
    - namespaceSelector: {} (2)
```

(1) Applies the policy only to **app:web** pods in default namespace.

2 Selects all pods in all namespaces.



NOTE

By default, if you omit specifying a **namespaceSelector** it does not select any namespaces, which means the policy allows traffic only from the namespace the network policy is deployed to.

2. Apply the policy by entering the following command:

```
$ oc apply -f web-allow-all-namespaces.yaml
```

Example output

```
networkpolicy.networking.k8s.io/web-allow-all-namespaces created
```

Verification

1. Start a web service in the **default** namespace by entering the following command:

```
$ oc run web --namespace=default --image=nginx --labels="app=web" --expose --port=80
```

2. Run the following command to deploy an **alpine** image in the **secondary** namespace and to start a shell:

```
$ oc run test-$RANDOM --namespace=secondary --rm -i -t --image=alpine -- sh
```

3. Run the following command in the shell and observe that the request is allowed:

```
# wget -qO- --timeout=2 http://web.default
```

Expected output

```
<!DOCTYPE html>
<html>
<head>
<title>Welcome to nginx!</title>
<style>
html { color-scheme: light dark; }
body { width: 35em; margin: 0 auto;
font-family: Tahoma, Verdana, Arial, sans-serif; }
</style>
</head>
<body>
<h1>Welcome to nginx!</h1>
<p>If you see this page, the nginx web server is successfully installed and
working. Further configuration is required.</p>

<p>For online documentation and support please refer to
<a href="http://nginx.org/">nginx.org</a>.<br/>
Commercial support is available at
```

```
<a href="http://nginx.com/">nginx.com</a>.</p>

<p><em>Thank you for using nginx.</em></p>
</body>
</html>
```

4.2.6. Creating a network policy allowing traffic to an application from a namespace

Follow this procedure to configure a policy that allows traffic to a pod with the label **app=web** from a particular namespace. You might want to do this to:

- Restrict traffic to a production database only to namespaces where production workloads are deployed.
- Enable monitoring tools deployed to a particular namespace to scrape metrics from the current namespace.

Prerequisites

- You installed the OpenShift CLI (**oc**).
- You are working in the namespace that the network policy applies to.

Procedure

1. Create a policy that allows traffic from all pods in a particular namespaces with a label **purpose=production**. Save the YAML in the **web-allow-prod.yaml** file:

```
kind: NetworkPolicy
apiVersion: networking.k8s.io/v1
metadata:
  name: web-allow-prod
  namespace: default
spec:
  podSelector:
    matchLabels:
      app: web (1)
  policyTypes:
  - Ingress
  ingress:
  - from:
    - namespaceSelector:
        matchLabels:
          purpose: production (2)
```

(1) Applies the policy only to **app:web** pods in the default namespace.

(2) Restricts traffic to only pods in namespaces that have the label **purpose=production**.

2. Apply the policy by entering the following command:

```
$ oc apply -f web-allow-prod.yaml
```

Example output

```
networkpolicy.networking.k8s.io/web-allow-prod created
```

Verification

1. Start a web service in the **default** namespace by entering the following command:

```
$ oc run web --namespace=default --image=nginx --labels="app=web" --expose --port=80
```

2. Run the following command to create the **prod** namespace:

```
$ oc create namespace prod
```

3. Run the following command to label the **prod** namespace:

```
$ oc label namespace/prod purpose=production
```

4. Run the following command to create the **dev** namespace:

```
$ oc create namespace dev
```

5. Run the following command to label the **dev** namespace:

```
$ oc label namespace/dev purpose=testing
```

6. Run the following command to deploy an **alpine** image in the **dev** namespace and to start a shell:

```
$ oc run test-$RANDOM --namespace=dev --rm -i -t --image=alpine -- sh
```

7. Run the following command in the shell and observe that the request is blocked:

```
# wget -qO- --timeout=2 http://web.default
```

Expected output

```
wget: download timed out
```

8. Run the following command to deploy an **alpine** image in the **prod** namespace and start a shell:

```
$ oc run test-$RANDOM --namespace=prod --rm -i -t --image=alpine -- sh
```

9. Run the following command in the shell and observe that the request is allowed:

```
# wget -qO- --timeout=2 http://web.default
```

Expected output

```
<!DOCTYPE html>
```

```
<html>
<head>
<title>Welcome to nginx!</title>
<style>
html { color-scheme: light dark; }
body { width: 35em; margin: 0 auto;
font-family: Tahoma, Verdana, Arial, sans-serif; }
</style>
</head>
<body>
<h1>Welcome to nginx!</h1>
<p>If you see this page, the nginx web server is successfully installed and
working. Further configuration is required.</p>

<p>For online documentation and support please refer to
<a href="http://nginx.org/">nginx.org</a>.<br/>
Commercial support is available at
<a href="http://nginx.com/">nginx.com</a>.</p>

<p><em>Thank you for using nginx.</em></p>
</body>
</html>
```

4.3. EDITING A NETWORK POLICY

You can edit an existing network policy for a namespace. Typical edits might include changes to the pods to which the policy applies, allowed ingress traffic, and the destination ports on which to accept traffic. The **apiVersion**, **kind**, and **name** fields must not be changed when editing **NetworkPolicy** objects, as these define the resource itself.

4.3.1. Editing a network policy

You can edit a network policy in a namespace.

Prerequisites

- You installed the OpenShift CLI (**oc**).
- You are working in the namespace where the network policy exists.

Procedure

1. Optional: To list the network policy objects in a namespace, enter the following command:

   ```
   $ oc get networkpolicy
   ```

   where:

   <namespace>

   Optional: Specifies the namespace if the object is defined in a different namespace than the current namespace.

2. Edit the network policy object.

- If you saved the network policy definition in a file, edit the file and make any necessary changes, and then enter the following command.

  ```
  $ oc apply -n <namespace> -f <policy_file>.yaml
  ```

  where:

  <namespace>
  Optional: Specifies the namespace if the object is defined in a different namespace than the current namespace.

  <policy_file>
  Specifies the name of the file containing the network policy.

- If you need to update the network policy object directly, enter the following command:

  ```
  $ oc edit networkpolicy <policy_name> -n <namespace>
  ```

  where:

  <policy_name>
  Specifies the name of the network policy.

  <namespace>
  Optional: Specifies the namespace if the object is defined in a different namespace than the current namespace.

3. Confirm that the network policy object is updated.

   ```
   $ oc describe networkpolicy <policy_name> -n <namespace>
   ```

   where:

   <policy_name>
   Specifies the name of the network policy.

   <namespace>
   Optional: Specifies the namespace if the object is defined in a different namespace than the current namespace.

4.3.2. Example NetworkPolicy object

The following annotates an example NetworkPolicy object:

```
kind: NetworkPolicy
apiVersion: networking.k8s.io/v1
metadata:
  name: allow-27107 (1)
spec:
  podSelector: (2)
    matchLabels:
      app: mongodb
  ingress:
  - from:
    - podSelector: (3)
```

```
    matchLabels:
      app: app
  ports: (4)
  - protocol: TCP
    port: 27017
```

1. The name of the NetworkPolicy object.
2. A selector that describes the pods to which the policy applies.
3. A selector that matches the pods from which the policy object allows ingress traffic. The selector matches pods in the same namespace as the NetworkPolicy.
4. A list of one or more destination ports on which to accept traffic.

4.4. DELETING A NETWORK POLICY

You can delete a network policy from a namespace.

4.4.1. Deleting a network policy using the CLI

You can delete a network policy in a namespace.

Prerequisites

- You installed the OpenShift CLI (**oc**).
- You are working in the namespace where the network policy exists.

Procedure

- To delete a network policy object, enter the following command:

  ```
  $ oc delete networkpolicy <policy_name> -n <namespace>
  ```

  where:

  <policy_name>
  : Specifies the name of the network policy.

  <namespace>
  : Optional: Specifies the namespace if the object is defined in a different namespace than the current namespace.

  Example output

  ```
  networkpolicy.networking.k8s.io/default-deny deleted
  ```

4.5. VIEWING A NETWORK POLICY

Use the following procedure to view a network policy for a namespace.

4.5.1. Viewing network policies using the CLI

You can examine the network policies in a namespace.

Prerequisites

- You installed the OpenShift CLI (**oc**).
- You are working in the namespace where the network policy exists.

Procedure

- List network policies in a namespace:
  - To view network policy objects defined in a namespace, enter the following command:

    ```
    $ oc get networkpolicy
    ```

  - Optional: To examine a specific network policy, enter the following command:

    ```
    $ oc describe networkpolicy <policy_name> -n <namespace>
    ```

    where:

    <policy_name>
    Specifies the name of the network policy to inspect.

    <namespace>
    Optional: Specifies the namespace if the object is defined in a different namespace than the current namespace.

    For example:

    ```
    $ oc describe networkpolicy allow-same-namespace
    ```

    Output for oc describe command

    ```
    Name:         allow-same-namespace
    Namespace:    ns1
    Created on:   2021-05-24 22:28:56 -0400 EDT
    Labels:       <none>
    Annotations:  <none>
    Spec:
      PodSelector:     <none> (Allowing the specific traffic to all pods in this namespace)
      Allowing ingress traffic:
        To Port: <any> (traffic allowed to all ports)
        From:
          PodSelector: <none>
      Not affecting egress traffic
      Policy Types: Ingress
    ```

CHAPTER 5. MULTIPLE NETWORKS

5.1. ABOUT USING MULTIPLE NETWORKS

In addition to the default OVN-Kubernetes Container Network Interface (CNI) plugin, the MicroShift Multus CNI is available to chain other CNI plugins. Installing and using MicroShift Multus is optional.

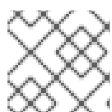
5.1.1. Additional networks in MicroShift

During cluster installation, the *default* pod network is configured with default values unless you customize the configuration. The default network handles all ordinary network traffic for the cluster. Using the MicroShift Multus CNI plugin, you can add additional interfaces to pods from other networks. This gives you flexibility when you configure pods that deliver network functionality, such as switching or routing.

5.1.1.1. Supported additional networks for network isolation

The following additional networks are supported in MicroShift 4.16:

- Bridge: Allows pods on the same host to communicate with each other and the host.
- IPVLAN: Allows pods on a host to communicate with other hosts.
  - This is similar to a MACVLAN-based additional network.
  - Each pod shares the same MAC address as the parent physical network interface, unlike a MACVLAN-based additional network.
- MACVLAN: Allows pods on a host to communicate with other hosts and the pods on those other hosts by using a physical network interface. Each pod that is attached to a MACVLAN-based additional network is provided with a unique MAC address.



NOTE

Setting network policies for additional networks is not supported.

5.1.1.2. Use case: Additional networks for network isolation

You can use an additional network in situations where network isolation is needed, including control plane and data plane separation. For example, you can configure an additional interface if you want pods to access a network on the host and also communicate with devices deployed to the edge. These edge devices might be on an isolated operator network or are periodically disconnected.

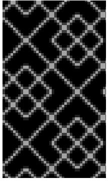
Isolating network traffic is useful for the following performance and security reasons:

Performance

You can send traffic on two different planes to manage the amount of traffic on each plane.

Security

You can send sensitive traffic onto a network plane that is managed specifically for security considerations, and you can separate private data that must not be shared between tenants or customers.



IMPORTANT

The Multus CNI plugin is deployed when the MicroShift service starts up. Therefore, a host restart is required if the **microshift-multus** RPM package is added after MicroShift has started. Restarting ensures that all containers are re-created with Multus annotations.

5.1.1.3. How additional networks are implemented

All of the pods in the cluster still use the cluster-wide default network to maintain connectivity across the cluster. Every pod has an **eth0** interface that is attached to the cluster-wide pod network.

- You can view the interfaces for a pod by using the **oc get pod <pod_name> -o=jsonpath='{ .metadata.annotations.k8s\.v1\.cni\.cncf\.io/network-status }'** command.
- If you add additional network interfaces that use the MicroShift Multus CNI, they are named **net1**, **net2**, ..., **netN**.
- The CNI configuration is created when the MicroShift Multus DaemonSet starts. This configuration is autogenerated and includes the primary CNI that is the default delegate. For MicroShift, the default CNI is OVN-Kubernetes.

5.1.1.4. How to attached additional networks to pods

To attach additional network interfaces to a pod, you must create and apply configurations that define how the interfaces are attached.

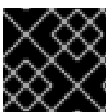
- You must configure any additional networks you want to use. Because of individual differences in networks, no default configuration is provided.
- You must apply YAML manifest to specify each interface by using a **NetworkAttachmentDefinition** custom resource (CR). A configuration inside each of these CRs defines how that interface is created.
- CRI-O must be configured to use Multus. A default configuration is included in the **microshift-multus** RPM.
  - If the Multus CNI is installed on an existing MicroShift instance, the host must be restarted.
  - If the Multus CNI is installed alongside MicroShift, you can add CRs and pods and then start the MicroShift service. Restarting the host in this scenario is not needed.

5.1.1.5. Configurations for additional network types

The specific configuration fields for additional networks is described in the following sections.

5.1.2. Installing the Multus CNI plugin on a running cluster

If you want to attach additional networks to a pod for high-performance network configurations, you can install the MicroShift Multus RPM package. After installation, a host restart is required to recreate all the pods with the Multus annotation.



IMPORTANT

Uninstalling the Multus CNI plugin is not supported.

Prerequisites

1. You have root access to the host.

Procedure

1. Install the Multus RPM package by running the following command:

```
$ sudo dnf install microshift-multus
```

TIP

If you create your custom resources (CRs) for additional networks now, you can complete your installation and apply configurations with one restart.

2. To apply the package manifest to an active cluster, restart the host by running the following command:

```
$ sudo systemctl restart
```

Verification

1. After restarting, ensure that the Multus CNI plugin components are created by running the following command:

```
$ oc get pod -A | grep multus
```

Example output

```
openshift-multus    dhcp-daemon-ktzqf    1/1   Running   0     45h
openshift-multus    multus-4frf4         1/1   Running   0     45h
```

Next steps

1. If you have not done so, configure and apply the additional networks you want to use.

2. Deploy your applications that use the created CRs.

5.1.3. Configuration for a bridge additional network

The following object describes the configuration parameters for the Bridge CNI plugin:

Table 5.1. Bridge CNI plugin JSON configuration object

Field	Type	Description
cniVersion	**string**	The CNI specification version. The **0.4.0** value is required.
type	**string**	The name of the CNI plugin to configure: **bridge**.

Field	Type	Description
ipam	**object**	The configuration object for the IPAM CNI plugin. The plugin manages IP address assignment for the attachment definition.
bridge	**string**	Optional: Specify the name of the virtual bridge to use. If the bridge interface does not exist on the host, it is created. The default value is **cni0**.
ipMasq	**boolean**	Optional: Set to **true** to enable IP masquerading for traffic that leaves the virtual network. The source IP address for all traffic is rewritten to the bridge's IP address. If the bridge does not have an IP address, this setting has no effect. The default value is **false**.
isGateway	**boolean**	Optional: Set to **true** to assign an IP address to the bridge. The default value is **false**.
isDefaultGateway	**boolean**	Optional: Set to **true** to configure the bridge as the default gateway for the virtual network. The default value is **false**. If **isDefaultGateway** is set to **true**, then **isGateway** is also set to **true** automatically.
forceAddress	**boolean**	Optional: Set to **true** to allow assignment of a previously assigned IP address to the virtual bridge. When set to **false**, if an IPv4 address or an IPv6 address from overlapping subsets is assigned to the virtual bridge, an error occurs. The default value is **false**.
hairpinMode	**boolean**	Optional: Set to **true** to allow the virtual bridge to send an Ethernet frame back through the virtual port it was received on. This mode is also known as *reflective relay*. The default value is **false**.
promiscMode	**boolean**	Optional: Set to **true** to enable promiscuous mode on the bridge. The default value is **false**.
mtu	**string**	Optional: Set the maximum transmission unit (MTU) to the specified value. The default value is automatically set by the kernel.
enabledad	**boolean**	Optional: Enables duplicate address detection for the container side **veth**. The default value is **false**.
macspoofchk	**boolean**	Optional: Enables mac spoof check, limiting the traffic originating from the container to the mac address of the interface. The default value is **false**.

5.1.3.1. Bridge CNI plugin configuration example

The following example configures an additional network named **bridge-conf** for use with the MicroShift Multus CNI:

```
apiVersion: "k8s.cni.cncf.io/v1"
kind: NetworkAttachmentDefinition
metadata:
  name: bridge-conf
spec:
  config: '{
      "cniVersion": "0.4.0",
      "type": "bridge",
      "bridge": "test-bridge",
      "mode": "bridge",
      "ipam": {
        "type": "host-local",
        "ranges": [
          [
            {
              "subnet": "10.10.0.0/16",
              "rangeStart": "10.10.1.20",
              "rangeEnd": "10.10.3.50",
              "gateway": "10.10.0.254"
            }
          ]
        ],
        "dataDir": "/var/lib/cni/test-bridge"
      }
    }'
```

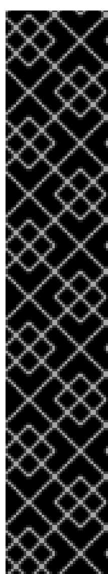
5.1.4. Configuration for an ipvlan additional network

The following object describes the configuration parameters for the IPVLAN CNI plugin:

Table 5.2. IPVLAN CNI plugin JSON configuration object

Field	Type	Description
cniVersion	**string**	The CNI specification version. The **0.3.1** value is required.
name	**string**	The value for the **name** parameter you provided previously for the CNO configuration.
type	**string**	The name of the CNI plugin to configure: **ipvlan**.
ipam	**object**	The configuration object for the IPAM CNI plugin. The plugin manages IP address assignment for the attachment definition. This is required unless the plugin is chained.
mode	**string**	Optional: The operating mode for the virtual network. The value must be **l2**, **l3**, or **l3s**. The default value is **l2**.

Field	Type	Description
master	**string**	Optional: The Ethernet interface to associate with the network attachment. If a **master** is not specified, the interface for the default network route is used.
mtu	**integer**	Optional: Set the maximum transmission unit (MTU) to the specified value. The default value is automatically set by the kernel.
linkInContainer	**boolean**	Optional: Specifies whether the master interface is in the container network namespace or the main network namespace. Set the value to **true** to request the use of a container namespace master interface.



IMPORTANT

- The **ipvlan** object does not allow virtual interfaces to communicate with the **master** interface. Therefore the container is not able to reach the host by using the **ipvlan** interface. Be sure that the container joins a network that provides connectivity to the host, such as a network supporting the Precision Time Protocol (**PTP**).
- A single **master** interface cannot simultaneously be configured to use both **macvlan** and **ipvlan**.
- For IP allocation schemes that cannot be interface agnostic, the **ipvlan** plugin can be chained with an earlier plugin that handles this logic. If the **master** is omitted, then the previous result must contain a single interface name for the **ipvlan** plugin to enslave. If **ipam** is omitted, then the previous result is used to configure the **ipvlan** interface.

5.1.4.1. IPVLAN CNI plugin configuration example

The following example configures an additional network named **ipvlan-net**:

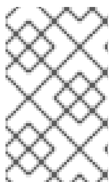
```
{
  "cniVersion": "0.3.1",
  "name": "ipvlan-net",
  "type": "ipvlan",
  "master": "eth1",
  "linkInContainer": false,
  "mode": "l3",
  "ipam": {
    "type": "static",
    "addresses": [
      {
        "address": "192.168.10.10/24"
      }
    ]
  }
}
```

5.1.5. Configuration for a macvlan additional network

The following object describes the configuration parameters for the MACVLAN CNI plugin:

Table 5.3. MACVLAN CNI plugin JSON configuration object

Field	Type	Description
cniVersion	**string**	The CNI specification version. The **0.3.1** value is required.
name	**string**	The value for the **name** parameter you provided previously for the CNO configuration.
type	**string**	The name of the CNI plugin to configure: **macvlan**.
ipam	**object**	The configuration object for the IPAM CNI plugin. The plugin manages IP address assignment for the attachment definition.
mode	**string**	Optional: Configures traffic visibility on the virtual network. Must be either **bridge**, **passthru**, **private**, or **vepa**. If a value is not provided, the default value is **bridge**.
master	**string**	Optional: The host network interface to associate with the newly created macvlan interface. If a value is not specified, then the default route interface is used.
mtu	**string**	Optional: The maximum transmission unit (MTU) to the specified value. The default value is automatically set by the kernel.
linkInContainer	**boolean**	Optional: Specifies whether the master interface is in the container network namespace or the main network namespace. Set the value to **true** to request the use of a container namespace master interface.



NOTE

If you specify the **master** key for the plugin configuration, use a different physical network interface than the one that is associated with your primary network plugin to avoid possible conflicts.

5.1.5.1. MACVLAN CNI plugin configuration example

The following example configures an additional network named **macvlan-net**:

```
{
  "cniVersion": "0.3.1",
  "name": "macvlan-net",
  "type": "macvlan",
  "master": "eth1",
  "linkInContainer": false,
  "mode": "bridge",
  "ipam": {
```

```
    "type": "dhcp"
    }
}
```

5.1.6. Additional resources

- Configuring and using multiple networks

5.2. CONFIGURING AND USING MULTIPLE NETWORKS

After you have installed the MicroShift Multus Container Network Interface (CNI), you can use other networking plugins by using configurations.

5.2.1. IP address management types and additional networks

IP addresses are provisioned for an additional network through an IP Address Management (IPAM) CNI plugin that you configure. Supported IP address provisioning types in MicroShift are **host-local**, **static**, and **dhcp**.

5.2.1.1. bridge interface specifics

When using the **bridge** type interface and the **dhcp** IPAM, a DHCP server listening on the bridged network is required. If you are using a firewall, configuring the **firewalld** service by running the **firewall-cmd --remove-service=dhcp** command to allow DHCP traffic on the network zone is also required.

5.2.1.2. macvlan interface specifics

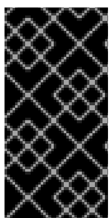
The **macvlan** type interface accesses the network that the host is connected to. This means that the interface can receive an IP address from the DHCP server on the host network if the **dhcp** IPAM plugin is used.

5.2.1.3. ipvlan interface specifics

The **ipvlan** interface also has direct access to the host network, but shares a MAC address with the host interface. The **ipvlan** type interface cannot be used with the **dhcp** plugin because of the shared MAC address. The IPAM plugin does not support the DHCP protocol with **ClientID**.

5.2.2. Creating a NetworkAttachmentDefinition for an additional network

Use the following procedure to create a **NetworkAttachmentDefinition** configuration file for an additional network. In this example, a bridge-type interface is used. You can also use the example workflow here that uses **host-local** IP address management (IPAM) to configure other supported additional network types.



IMPORTANT

If you use **bridge** and the **dhcp** IPAM, a DHCP server listening on the bridged network is required. If you are also using a firewall, configuring the firewalld service to allow DHCP traffic on the network zone is also required. You can run the **firewall-cmd --remove-service=dhcp** command in this case.

Prerequisites

- The MicroShift Multus CNI is installed.
- The OpenShift CLI (**oc**) is installed.
- The cluster is running.

Procedure

1. Optional: Verify that the MicroShift cluster is running with the Multus CNI by running the following command:

```
$ oc get pods -n openshift-multus
```

Example output

```
NAME                READY   STATUS    RESTARTS   AGE
dhcp-daemon-dfbzw   1/1     Running   0          5h
multus-rz8xc        1/1     Running   0          5h
```

2. Create a **NetworkAttachmentDefinition** configuration file by running the following command and using the following example file for reference:

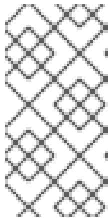
```
$ oc apply -f network-attachment-definition.yaml
```

Example **NetworkAttachmentDefinition** file

```
apiVersion: "k8s.cni.cncf.io/v1"
kind: NetworkAttachmentDefinition
metadata:
  name: bridge-conf
spec:
  config: '{
      "cniVersion": "0.4.0",
      "type": "bridge", (1)
      "bridge": "br-test", (2)
      "mode": "bridge",
      "ipam": {
        "type": "host-local", (3)
        "ranges": [
          [
            {
              "subnet": "10.10.0.0/24",
              "rangeStart": "10.10.0.20",
              "rangeEnd": "10.10.0.50",
              "gateway": "10.10.0.254"
            }
          ],
          [
            {
              "subnet": "fd00:IJKL:MNOP:10::0/64", (4)
              "rangeStart": "fd00:IJKL:MNOP:10::1",
              "rangeEnd": "fd00:IJKL:MNOP:10::9"
```

```
    "dataDir": "/var/lib/cni/br-test"
  }
}'
```

1. The **type** value specifies a name of the CNI plugin. This example uses the **bridge** type.
2. The **bridge** value is name of the bridge on the MicroShift host that is used. The additional interface of the pod is connected to that bridge. If the interface does not exist on the host, the Bridge CNI creates it. If the interface already exists, it is reused. In this example, the name of the interface is **br-test**.
3. The IPAM type.
4. IPv6 addresses can be added to the secondary interface.



NOTE

Using the name of the bridge is specific to the **bridge** type of plugin. Other plugins use different fields in their **NetworkAttachmentDefinitions**. For example, the **macvlan** and **ipvlan** configurations use **master** to specify the host interface to attach.

5.2.3. Adding a pod to an additional network

You can add a pod to an additional network. At the time a pod is created, additional networks are attached to it. The pod continues to send normal cluster-related network traffic over the default network.

If you want to attach additional networks to a pod that is already running, you must restart the pod.

Prerequisites

- The OpenShift CLI (**oc**) is installed.
- The cluster is running.
- A network defined by a **NetworkAttachmentDefinition** object that you want to attach the pod to exists.

Procedure

1. Add an annotation to a **Pod** YAML file. Only one of the following annotation formats can be used:
   a. To attach an additional network without any customization, add an annotation with the following format. Replace **<network>** with the name of the additional network to associate with the pod:

```
apiVersion: v1
kind: Pod
metadata:
  annotations:
    k8s.v1.cni.cncf.io/networks: <network>[,<network>,...] (1)
# ...
```

1. Replace **<network>** with the name of the additional network to associate with the pod. To specify more than one additional network, separate each network with a comma. Do not include whitespace between the comma. If you specify the same additional network multiple times, that pod will have multiple network interfaces attached to that network.

Example annotation for a bridge-type additional network

```
apiVersion: v1
kind: Pod
metadata:
  annotations:
    k8s.v1.cni.cncf.io/networks: bridge-conf
# ...
```

b. To attach an additional network with customizations, add an annotation with the following format:

```
apiVersion: v1
kind: Pod
metadata:
  annotations:
    k8s.v1.cni.cncf.io/networks: |-
      [
        {
          "name": "<network>", (1)
          "namespace": "<namespace>", (2)
          "default-route": ["<default-route>"] (3)
        }
      ]
# ...
```

1. Specify the name of the additional network defined by a **NetworkAttachmentDefinition** object.
2. Specify the namespace where the **NetworkAttachmentDefinition** object is defined.
3. Optional: Specify an override for the default route, such as **192.168.17.1**.

2. To create a **Pod** YAML file and add the **NetworkAttachmentDefinition** annotation for an additional network, run the following command and use the example YAML:

```
$ oc apply -f ./<test-bridge>.yaml (1)
```

1. Replace *<test-bridge>* with the pod name that you want to use.

Example output

```
pod/test-bridge created
```

Example test-bridge pod YAML

```
apiVersion: v1
kind: Pod
metadata:
  name: test-bridge
  annotations:
    k8s.v1.cni.cncf.io/networks: bridge-conf
  labels:
    app: test-bridge
spec:
  terminationGracePeriodSeconds: 0
  containers:
  - name: hello-microshift
    image: quay.io/microshift/busybox:1.36
    command: ["/bin/sh"]
    args: ["-c", "while true; do echo -ne \"HTTP/1.0 200 OK\r\nContent-Length: 16\r\n\r\nHello MicroShift\" | nc -l -p 8080 ; done"]
    ports:
    - containerPort: 8080
      protocol: TCP
    securityContext:
      allowPrivilegeEscalation: false
      capabilities:
        drop:
        - ALL
      runAsNonRoot: true
      runAsUser: 1001
      runAsGroup: 1001
      seccompProfile:
        type: RuntimeDefault
```

3. Make sure that the **NetworkAttachmentDefinition** annotation is correct:

Example NetworkAttachmentDefinition annotation

```
apiVersion: v1
kind: Pod
metadata:
  annotations:
    k8s.v1.cni.cncf.io/networks: bridge-conf
# ...
```

4. Optional: To confirm that the **NetworkAttachmentDefinition** annotation exists in a **Pod** YAML, run the following command, replacing **<name>** with the name of the pod.

```
$ oc get pod <name> -o yaml (1)
```

(1) Replace *<name>* with the pod name you want to use. In the following example, **test-bridge** is used.

In the following example, the **test-bridge** is attached to the **net1** additional network:

```
$ oc get pod test-bridge -o yaml
```

Example output

```
apiVersion: v1
kind: Pod
metadata:
  annotations:
    k8s.v1.cni.cncf.io/networks: bridge-conf
    k8s.v1.cni.cncf.io/network-status: |- 1
      [{
          "name": "ovn-kubernetes",
          "interface": "eth0",
          "ips": [
              "10.42.0.18"
          ],
          "default": true,
          "dns": {}
      },{
          "name": "bridge-conf",
          "interface": "net1",
          "ips": [
              "20.2.2.100"
          ],
          "mac": "22:2f:60:a5:f8:00",
          "dns": {}
      }]
  name: pod
  namespace: default
spec:
# ...
status:
# ...
```

1. The **k8s.v1.cni.cncf.io/network-status** parameter is a JSON array of objects. Each object describes the status of an additional network attached to the pod. The annotation value is stored as a plain text value.

5. Verify that the pod is running by running the following command:

```
$ oc get pod
```

Example output

```
NAME        READY   STATUS    RESTARTS   AGE
test-bridge  1/1     Running   0          81s
```

5.2.4. Configuring an additional network

After you have created the NetworkAttachmentDefinition object and applied it, use the following example procedure to configure an additional network. In this example, the **bridge** type additional network is used. You can also use this workflow for other additional network types.

Prerequisite

1. You created and applied the **NetworkAttachmentDefinition** object configuration.

Procedure

1. Verify that the bridge was created on the host by running the following command:

```
$ ip a show br-test
```

Example output

```
22: br-test: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue state UP
group default qlen 1000
    link/ether 96:bf:ca:be:1d:15 brd ff:ff:ff:ff:ff:ff
    inet6 fe80::34e2:bbff:fed2:31f2/64 scope link
      valid_lft forever preferred_lft forever
```

2. Configure an IP address for the bridge by running the following command:

```
$ sudo ip addr add 10.10.0.10/24 dev br-test
```

3. Verify that the IP address configuration is added to the bridge by running the following command:

```
$ ip a show br-test
```

Example output

```
22: br-test: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue state UP
group default qlen 1000
    link/ether 96:bf:ca:be:1d:15 brd ff:ff:ff:ff:ff:ff
    inet 10.10.0.10/24 scope global br-test (1)
      valid_lft forever preferred_lft forever
    inet6 fe80::34e2:bbff:fed2:31f2/64 scope link
      valid_lft forever preferred_lft forever
```

(1) The IP address is configured as expected.

4. Verify the IP address of the pod by running the following command:

```
$ oc get pod test-bridge --
output=jsonpath='{.metadata.annotations.k8s\.v1\.cni\.cncf\.io/network-status}'
```

Example output

```
[{
    "name": "ovn-kubernetes",
    "interface": "eth0",
    "ips": [
        "10.42.0.17"
    ],
    "mac": "0a:58:0a:2a:00:11",
    "default": true,
```

```
    "dns": {}
},{
    "name": "default/bridge-conf", 1
    "interface": "net1",
    "ips": [
        "10.10.0.20"
    ],
    "mac": "82:01:98:e5:0c:b7",
    "dns": {}
```

1. The bridge additional network is attached as expected.

5. Optional: You can use **oc exec** to access the pod and confirm its interfaces by using the **ip** command:

```
$ oc exec -ti test-bridge -- ip a
```

Example output

```
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue qlen 1000
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
       valid_lft forever preferred_lft forever
    inet6 ::1/128 scope host
       valid_lft forever preferred_lft forever
2: eth0@if21: <BROADCAST,MULTICAST,UP,LOWER_UP,M-DOWN> mtu 1500 qdisc
noqueue
    link/ether 0a:58:0a:2a:00:11 brd ff:ff:ff:ff:ff:ff
    inet 10.42.0.17/24 brd 10.42.0.255 scope global eth0
       valid_lft forever preferred_lft forever
    inet6 fe80::858:aff:fe2a:11/64 scope link
       valid_lft forever preferred_lft forever
3: net1@if23: <BROADCAST,MULTICAST,UP,LOWER_UP,M-DOWN> mtu 1500 qdisc
noqueue
    link/ether 82:01:98:e5:0c:b7 brd ff:ff:ff:ff:ff:ff
    inet 10.10.0.20/24 brd 10.10.0.255 scope global net1 1
       valid_lft forever preferred_lft forever
    inet6 fe80::8001:98ff:fee5:cb7/64 scope link
       valid_lft forever preferred_lft forever
```

1. Pod is attached to the 10.10.0.20 IP address on the **net1 interface** as expected.

6. Confirm that the connection is working as expected by accessing the HTTP server in the pod from the MicroShift host. Use the following command:

```
$ curl 10.10.0.20:8080
```

Example output

```
Hello MicroShift
```

5.2.5. Removing a pod from an additional network

You can remove a pod from an additional network only by deleting the pod.

Prerequisites

- An additional network is attached to the pod.
- Install the OpenShift CLI (**oc**).
- Log in to the cluster.

Procedure

- To delete the pod, enter the following command:

```
$ oc delete pod <name> -n <namespace>
```

  - **<name>** is the name of the pod.
  - **<namespace>** is the namespace that contains the pod.

5.2.6. Troubleshooting Multus networking

If the settings for multiple networks are not configured properly, pods can fail to start. The following steps can help you solve for a couple common scenarios.

5.2.6.1. Pod networking cannot be configured

If the Multus CNI plugin cannot apply networking annotations to a pod, the pod does not start. Pods can also fail to start if any of the additional network CNIs fail.

Example error

```
Warning  NoNetworkFound    0s    multus   cannot find a network-attachment-definitio (asdasd) in
namespace (default): network-attachment-definitions.k8s.cni.cncf.io "bad-ref-doesnt-exist" not found
```

In this case, you can take the following steps to trouble CNI failures:

- Verify the values in both the **NetworkAttachmentDefinitions** and the annotations.
- Remove the annotation to verify whether the pod is created successfully with just the default network. If not, this might indicate a networking problem other than the Multus configuration.
- If you are a device administrator, you can inspect the **crio.service** or **microshift.service** logs, paying special attention to those that are generated by the **kubelet**.
  For example, the following error from the **kubelet** shows that the primary CNI is not running. This situation can be caused by pods not starting or because of a CRI-O misconfiguration such as an incorrect **cni_default_network** setting.

  Example kubelet-generated error

```
Feb 06 13:47:31 dev microshift[1494]: kubelet E0206 13:47:31.163290    1494
pod_workers.go:1298] "Error syncing pod, skipping" err="network is not ready: container
```

```
runtime network not ready: NetworkReady=false reason:NetworkPluginNotReady
message:Network plugin returns error: No CNI configuration file in /etc/cni/net.d/. Has your
network provider started?" pod="default/samplepod" podUID="fe0f7f7a-8c47-4488-952b-
8abc0d8e2602"
```

5.2.6.2. Missing configuration file

Sometimes a pod cannot be created because the annotations reference a **NetworkAttachmentDefinition** configuration YAML that does not exist. In this case an error such as the following is usually produced:

Example log

```
cannot find a network-attachment-definition (bad-conf) in namespace (default): network-attachment-
definitions.k8s.cni.cncf.io "bad-conf" not found" pod="default/samplepod"`
```

Example error output

```
"CreatePodSandbox for pod failed" err="rpc error: code = Unknown desc = failed to create pod
network sandbox k8s_samplepod_default_5fa13105-1bfb-4c6b-aee7-
3437cfb50e25_0(7517818bd8e85f07b551f749c7529be88b4e7daef0dd572d049aa636950c76c6):
error adding pod default_samplepod to CNI network \"multus-cni-network\": plugin type=\"multus\"
name=\"multus-cni-network\" failed (add): Multus: [default/samplepod/5fa13105-1bfb-4c6b-aee7-
3437cfb50e25]: error loading k8s delegates k8s args: TryLoadPodDelegates: error in getting k8s
network for pod: GetNetworkDelegates: failed getting the delegate: getKubernetesDelegate: cannot
find a network-attachment-definition (bad-conf) in namespace (default): network-attachment-
definitions.k8s.cni.cncf.io \"bad-conf\" not found" pod="default/samplepod"
```

To fix this error, create and apply the **NetworkAttachmentDefinitions** YAML.

5.2.7. Additional resources

- About using multiple networks
- Configuration of IP address assignment for an additional network

CHAPTER 6. CONFIGURING ROUTES

You can configure routes for MicroShift for clusters.

6.1. CREATING AN HTTP-BASED ROUTE

A route allows you to host your application at a public URL. It can either be secure or unsecured, depending on the network security configuration of your application. An HTTP-based route is an unsecured route that uses the basic HTTP routing protocol and exposes a service on an unsecured application port.

The following procedure describes how to create a simple HTTP-based route to a web application, using the **hello-openshift** application as an example.

Prerequisites

- You installed the OpenShift CLI (**oc**).
- You are logged in as an administrator.
- You have a web application that exposes a port and a TCP endpoint listening for traffic on the port.

Procedure

1. Create a project called **hello-openshift** by running the following command:

```
$ oc new-project hello-openshift
```

2. Create a pod in the project by running the following command:

```
$ oc create -f https://raw.githubusercontent.com/openshift/origin/master/examples/hello-openshift/hello-pod.json
```

3. Create a service called **hello-openshift** by running the following command:

```
$ oc expose pod/hello-openshift
```

4. Create an unsecured route to the **hello-openshift** application by running the following command:

```
$ oc expose svc hello-openshift
```

Verification

- To verify that the **route** resource that you created, run the following command:

```
$ oc get routes -o yaml <name of resource> (1)
```

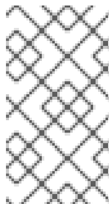
(1) In this example, the route is named **hello-openshift**.

Sample YAML definition of the created unsecured route:

```
apiVersion: route.openshift.io/v1
kind: Route
metadata:
  name: hello-openshift
spec:
  host: hello-openshift-hello-openshift.<Ingress_Domain> 1
  port:
    targetPort: 8080 2
  to:
    kind: Service
    name: hello-openshift
```

1. **<Ingress_Domain>** is the default ingress domain name. The **ingresses.config/cluster** object is created during the installation and cannot be changed. If you want to specify a different domain, you can specify an alternative cluster domain using the **appsDomain** option.

2. **targetPort** is the target port on pods that is selected by the service that this route points to.



NOTE

To display your default ingress domain, run the following command:

```
$ oc get ingresses.config/cluster -o jsonpath={.spec.domain}
```

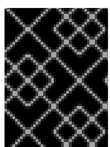
6.1.1. HTTP Strict Transport Security

HTTP Strict Transport Security (HSTS) policy is a security enhancement, which signals to the browser client that only HTTPS traffic is allowed on the route host. HSTS also optimizes web traffic by signaling HTTPS transport is required, without using HTTP redirects. HSTS is useful for speeding up interactions with websites.

When HSTS policy is enforced, HSTS adds a Strict Transport Security header to HTTP and HTTPS responses from the site. You can use the **insecureEdgeTerminationPolicy** value in a route to redirect HTTP to HTTPS. When HSTS is enforced, the client changes all requests from the HTTP URL to HTTPS before the request is sent, eliminating the need for a redirect.

Cluster administrators can configure HSTS to do the following:

- Enable HSTS per-route
- Disable HSTS per-route
- Enforce HSTS per-domain, for a set of domains, or use namespace labels in combination with domains



IMPORTANT

HSTS works only with secure routes, either edge-terminated or re-encrypt. The configuration is ineffective on HTTP or passthrough routes.

6.1.2. Enabling HTTP Strict Transport Security per-route

HTTP strict transport security (HSTS) is implemented in the HAProxy template and applied to edge and re-encrypt routes that have the **haproxy.router.openshift.io/hsts_header** annotation.

Prerequisites

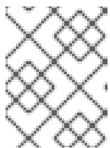
- You have root access to the cluster.
- You installed the OpenShift CLI (**oc**).

Procedure

- To enable HSTS on a route, add the **haproxy.router.openshift.io/hsts_header** value to the edge-terminated or re-encrypt route. You can use the **oc annotate** tool to do this by running the following command:

```
$ oc annotate route <route_name> -n <namespace> --overwrite=true
"haproxy.router.openshift.io/hsts_header"="max-age=31536000;\ 1
includeSubDomains;preload"
```

1 In this example, the maximum age is set to **31536000** ms, which is approximately 8.5 hours.



NOTE

In this example, the equal sign (**=**) is in quotes. This is required to properly execute the annotate command.

Example route configured with an annotation

```
apiVersion: route.openshift.io/v1
kind: Route
metadata:
  annotations:
    haproxy.router.openshift.io/hsts_header: max-age=31536000;includeSubDomains;preload
1 2 3
...
spec:
  host: def.abc.com
  tls:
    termination: "reencrypt"
    ...
  wildcardPolicy: "Subdomain"
```

1 Required. **max-age** measures the length of time, in seconds, that the HSTS policy is in effect. If set to **0**, it negates the policy.

2 Optional. When included, **includeSubDomains** tells the client that all subdomains of the host must have the same HSTS policy as the host.

3 Optional. When **max-age** is greater than 0, you can add **preload** in **haproxy.router.openshift.io/hsts_header** to allow external services to include this site in their HSTS preload lists. For example, sites such as Google can construct a list of sites that have **preload** set. Browsers can then use these lists to determine which sites they can communicate with over HTTPS, even before they have interacted with the site. Without

preload set, browsers must have interacted with the site over HTTPS, at least once, to get the header.

6.1.3. Disabling HTTP Strict Transport Security per-route

To disable HTTP strict transport security (HSTS) per-route, you can set the **max-age** value in the route annotation to **0**.

Prerequisites

- You have root access to the cluster.
- You installed the OpenShift CLI (**oc**).

Procedure

- To disable HSTS, set the **max-age** value in the route annotation to **0**, by entering the following command:

```
$ oc annotate route <route_name> -n <namespace> --overwrite=true
"haproxy.router.openshift.io/hsts_header"="max-age=0"
```

TIP

You can alternatively apply the following YAML to create the config map:

Example of disabling HSTS per-route

```
metadata:
  annotations:
    haproxy.router.openshift.io/hsts_header: max-age=0
```

- To disable HSTS for every route in a namespace, enter the following command:

```
$ oc annotate route --all -n <namespace> --overwrite=true
"haproxy.router.openshift.io/hsts_header"="max-age=0"
```

Verification

1. To query the annotation for all routes, enter the following command:

```
$ oc get route  --all-namespaces -o go-template='{{range .items}}{{if .metadata.annotations}}
{{$a := index .metadata.annotations "haproxy.router.openshift.io/hsts_header"}}{{$n :=
.metadata.name}}{{with $a}}Name: {{$n}} HSTS: {{$a}}{{"\n"}}{{else}}{{""}}{{end}}{{end}}
{{end}}'
```

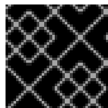
Example output

```
Name: routename HSTS: max-age=0
```

6.1.4. Enforcing HTTP Strict Transport Security per-domain

You can configure a route with a compliant HSTS policy annotation. To handle upgraded clusters with non-compliant HSTS routes, you can update the manifests at the source and apply the updates.

You cannot use **oc expose route** or **oc create route** commands to add a route in a domain that enforces HSTS because the API for these commands does not accept annotations.



IMPORTANT

HSTS cannot be applied to insecure, or non-TLS, routes.

Prerequisites

- You have root access to the cluster.
- You installed the OpenShift CLI (**oc**).

Procedure

- Apply HSTS to all routes in the cluster by running the following **oc annotate command**:

```
$ oc annotate route --all --all-namespaces --overwrite=true
"haproxy.router.openshift.io/hsts_header"="max-
age=31536000;preload;includeSubDomains"
```

- Apply HSTS to all routes in a particular namespace by running the following **oc annotate command**:

```
$ oc annotate route --all -n <my_namespace> --overwrite=true
"haproxy.router.openshift.io/hsts_header"="max-
age=31536000;preload;includeSubDomains" (1)
```

(1) Replace _<my_namespace> with the namespace you want to use.

Verification

- Review the HSTS annotations on all routes by running the following command:

```
$ oc get route  --all-namespaces -o go-template='{{range .items}}{{if .metadata.annotations}}
{{$a := index .metadata.annotations "haproxy.router.openshift.io/hsts_header"}}{{$n :=
.metadata.name}}{{with $a}}Name: {{$n}} HSTS: {{$a}}{{"\n"}}{{else}}{{""}}{{end}}{{end}}
{{end}}'
```

Example output

```
Name: <_routename_> HSTS: max-age=31536000;preload;includeSubDomains
```

6.2. THROUGHPUT ISSUE TROUBLESHOOTING METHODS

Sometimes applications deployed by using Red Hat build of MicroShift can cause network throughput issues, such as unusually high latency between specific services.

If pod logs do not reveal any cause of the problem, use the following methods to analyze performance issues:

- Use a packet analyzer, such as **ping** or **tcpdump** to analyze traffic between a pod and its node. For example, run the **tcpdump** tool on each pod while reproducing the behavior that led to the issue. Review the captures on both sides to compare send and receive timestamps to analyze the latency of traffic to and from a pod. Latency can occur in Red Hat build of MicroShift if a node interface is overloaded with traffic from other pods, storage devices, or the data plane.

```
$ tcpdump -s 0 -i any -w /tmp/dump.pcap host <podip 1> && host <podip 2> 1
```

1. **podip** is the IP address for the pod. Run the **oc get pod <pod_name> -o wide** command to get the IP address of a pod.

  The **tcpdump** command generates a file at **/tmp/dump.pcap** containing all traffic between these two pods. You can run the analyzer shortly before the issue is reproduced and stop the analyzer shortly after the issue is finished reproducing to minimize the size of the file. You can also run a packet analyzer between the nodes (eliminating the SDN from the equation) with:

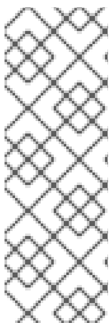
```
$ tcpdump -s 0 -i any -w /tmp/dump.pcap port 4789
```

- Use a bandwidth measuring tool, such as **iperf**, to measure streaming throughput and UDP throughput. Locate any bottlenecks by running the tool from the pods first, and then running it from the nodes.

6.3. USING COOKIES TO KEEP ROUTE STATEFULNESS

Red Hat build of MicroShift provides sticky sessions, which enables stateful application traffic by ensuring all traffic hits the same endpoint. However, if the endpoint pod terminates, whether through restart, scaling, or a change in configuration, this statefulness can disappear.

Red Hat build of MicroShift can use cookies to configure session persistence. The ingress controller selects an endpoint to handle any user requests, and creates a cookie for the session. The cookie is passed back in the response to the request and the user sends the cookie back with the next request in the session. The cookie tells the ingress controller which endpoint is handling the session, ensuring that client requests use the cookie so that they are routed to the same pod.



NOTE

Cookies cannot be set on passthrough routes, because the HTTP traffic cannot be seen. Instead, a number is calculated based on the source IP address, which determines the backend.

If backends change, the traffic can be directed to the wrong server, making it less sticky. If you are using a load balancer, which hides source IP, the same number is set for all connections and traffic is sent to the same pod.

6.3.1. Annotating a route with a cookie

You can set a cookie name to overwrite the default, auto-generated one for the route. This allows the application receiving route traffic to know the cookie name. Deleting the cookie can force the next request to re-choose an endpoint. The result is that if a server is overloaded, that server tries to remove the requests from the client and redistribute them.

Procedure

1. Annotate the route with the specified cookie name:

```
$ oc annotate route <route_name> router.openshift.io/cookie_name="<cookie_name>"
```

where:

<route_name>
Specifies the name of the route.

<cookie_name>
Specifies the name for the cookie.

For example, to annotate the route **my_route** with the cookie name **my_cookie**:

```
$ oc annotate route my_route router.openshift.io/cookie_name="my_cookie"
```

2. Capture the route hostname in a variable:

```
$ ROUTE_NAME=$(oc get route <route_name> -o jsonpath='{.spec.host}')
```

where:

<route_name>
Specifies the name of the route.

3. Save the cookie, and then access the route:

```
$ curl $ROUTE_NAME -k -c /tmp/cookie_jar
```

Use the cookie saved by the previous command when connecting to the route:

```
$ curl $ROUTE_NAME -k -b /tmp/cookie_jar
```

6.4. PATH-BASED ROUTES

Path-based routes specify a path component that can be compared against a URL, which requires that the traffic for the route be HTTP based. Thus, multiple routes can be served using the same hostname, each with a different path. Routers should match routes based on the most specific path to the least.

The following table shows example routes and their accessibility:

Table 6.1. Route availability

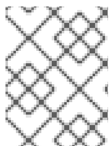
Route	When Compared to	Accessible
www.example.com/test	*www.example.com/test*	Yes
	www.example.com	No

Route	When Compared to	Accessible
www.example.com/test and *www.example.com*	*www.example.com/test*	Yes
	www.example.com	Yes
www.example.com	*www.example.com/text*	Yes (Matched by the host, not the route)
	www.example.com	Yes

An unsecured route with a path

```
apiVersion: route.openshift.io/v1
kind: Route
metadata:
  name: route-unsecured
spec:
  host: www.example.com
  path: "/test" 1
  to:
    kind: Service
    name: service-name
```

1 The path is the only added attribute for a path-based route.

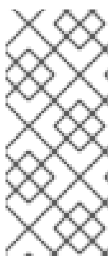


NOTE

Path-based routing is not available when using passthrough TLS, as the router does not terminate TLS in that case and cannot read the contents of the request.

6.5. HTTP HEADER CONFIGURATION

When setting or deleting headers, you can use an individual route to modify request and response headers. You can also set certain headers by using route annotations. The various ways of configuring headers can present challenges when working together.



NOTE

You can only set or delete headers within a **Route** CR. You cannot append headers. If an HTTP header is set with a value, that value must be complete and not require appending in the future. In situations where it makes sense to append a header, such as the X-Forwarded-For header, use the **spec.httpHeaders.forwardedHeaderPolicy** field, instead of **spec.httpHeaders.actions**.

Example Route spec

```
apiVersion: route.openshift.io/v1
kind: Route
```

```yaml
# ...
spec:
  httpHeaders:
    actions:
      response:
      - name: X-Frame-Options
        action:
          type: Set
          set:
            value: SAMEORIGIN
```

Any actions defined in a route override values set using route annotations.

6.5.1. Special case headers

The following headers are either prevented entirely from being set or deleted, or allowed under specific circumstances:

Header name	Configurable using **Route** spec	Reason for disallowment	Configurable using another method
proxy	No	The **proxy** HTTP request header can be used to exploit vulnerable CGI applications by injecting the header value into the **HTTP_PROXY** environment variable. The **proxy** HTTP request header is also non-standard and prone to error during configuration.	No
host	Yes	When the **host** HTTP request header is set using the **IngressController** CR, HAProxy can fail when looking up the correct route.	No
strict-transport-security	No	The **strict-transport-security** HTTP response header is already handled using route annotations and does not need a separate implementation.	Yes: the **haproxy.router.openshift.io/hsts_header** route annotation

Header name	Configurable using **Route** spec	Reason for disallowment	Configurable using another method
cookie and **set-cookie**	No	The cookies that HAProxy sets are used for session tracking to map client connections to particular back-end servers. Allowing these headers to be set could interfere with HAProxy's session affinity and restrict HAProxy's ownership of a cookie.	Yes: * the **haproxy.router.openshift.io/disable_cookie** route annotation * the **haproxy.router.openshift.io/cookie_name** route annotation

6.6. SETTING OR DELETING HTTP REQUEST AND RESPONSE HEADERS IN A ROUTE

You can set or delete certain HTTP request and response headers for compliance purposes or other reasons. You can set or delete these headers either for all routes served by an Ingress Controller or for specific routes.

For example, you might want to enable a web application to serve content in alternate locations for specific routes if that content is written in multiple languages, even if there is a default global location specified by the Ingress Controller serving the routes.

The following procedure creates a route that sets the Content-Location HTTP request header so that the URL associated with the application, **https://app.example.com**, directs to the location **https://app.example.com/lang/en-us**. Directing application traffic to this location means that anyone using that specific route is accessing web content written in American English.

Prerequisites

- You have installed the OpenShift CLI (**oc**).
- You are logged into an Red Hat build of MicroShift cluster as a project administrator.
- You have a web application that exposes a port and an HTTP or TLS endpoint listening for traffic on the port.

Procedure

1. Create a route definition and save it in a file called **app-example-route.yaml**:

   YAML definition of the created route with HTTP header directives

```
apiVersion: route.openshift.io/v1
kind: Route
# ...
spec:
  host: app.example.com
  tls:
    termination: edge
```

```
to:
  kind: Service
  name: app-example
httpHeaders:
  actions: (1)
    response: (2)
    - name: Content-Location (3)
      action:
        type: Set (4)
        set:
          value: /lang/en-us (5)
```

1. The list of actions you want to perform on the HTTP headers.
2. The type of header you want to change. In this case, a response header.
3. The name of the header you want to change. For a list of available headers you can set or delete, see *HTTP header configuration*.
4. The type of action being taken on the header. This field can have the value **Set** or **Delete**.
5. When setting HTTP headers, you must provide a **value**. The value can be a string from a list of available directives for that header, for example **DENY**, or it can be a dynamic value that will be interpreted using HAProxy's dynamic value syntax. In this case, the value is set to the relative location of the content.

2. Create a route to your existing web application using the newly created route definition:

```
$ oc -n app-example create -f app-example-route.yaml
```

For HTTP request headers, the actions specified in the route definitions are executed after any actions performed on HTTP request headers in the Ingress Controller. This means that any values set for those request headers in a route will take precedence over the ones set in the Ingress Controller. For more information on the processing order of HTTP headers, see *HTTP header configuration*.

6.7. CREATING A ROUTE THROUGH AN INGRESS OBJECT

Some ecosystem components have an integration with Ingress resources but not with route resources. To cover this case, Red Hat build of MicroShift automatically creates managed route objects when an Ingress object is created. These route objects are deleted when the corresponding Ingress objects are deleted.

Procedure

1. Define an Ingress object in the Red Hat build of MicroShift console or by entering the **oc create** command:

YAML Definition of an Ingress

```
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: frontend
```

```
  annotations:
    route.openshift.io/termination: "reencrypt" (1)
    route.openshift.io/destination-ca-certificate-secret: secret-ca-cert (2)
spec:
  rules:
  - host: www.example.com (3)
    http:
      paths:
      - backend:
          service:
            name: frontend
            port:
              number: 443
        path: /
        pathType: Prefix
  tls:
  - hosts:
    - www.example.com
    secretName: example-com-tls-certificate
```

1. The **route.openshift.io/termination** annotation can be used to configure the **spec.tls.termination** field of the **Route** as **Ingress** has no field for this. The accepted values are **edge**, **passthrough** and **reencrypt**. All other values are silently ignored. When the annotation value is unset, **edge** is the default route. The TLS certificate details must be defined in the template file to implement the default edge route.

3. When working with an **Ingress** object, you must specify an explicit hostname, unlike when working with routes. You can use the **<host_name>.<cluster_ingress_domain>** syntax, for example **apps.openshiftdemos.com**, to take advantage of the ***.<cluster_ingress_domain>** wildcard DNS record and serving certificate for the cluster. Otherwise, you must ensure that there is a DNS record for the chosen hostname.

   a. If you specify the **passthrough** value in the **route.openshift.io/termination** annotation, set **path** to **''** and **pathType** to **ImplementationSpecific** in the spec:

```
spec:
  rules:
  - host: www.example.com
    http:
      paths:
      - path: ''
        pathType: ImplementationSpecific
        backend:
          service:
            name: frontend
            port:
              number: 443
```

```
$ oc apply -f ingress.yaml
```

2. The **route.openshift.io/destination-ca-certificate-secret** can be used on an Ingress object to define a route with a custom destination certificate (CA). The annotation references a kubernetes secret, **secret-ca-cert** that will be inserted into the generated route.

a. To specify a route object with a destination CA from an ingress object, you must create a **kubernetes.io/tls** or **Opaque** type secret with a certificate in PEM-encoded format in the **data.tls.crt** specifier of the secret.

2. List your routes:

```
$ oc get routes
```

The result includes an autogenerated route whose name starts with **frontend-**:

```
NAME            HOST/PORT        PATH    SERVICES    PORT    TERMINATION
WILDCARD
frontend-gnztq   www.example.com         frontend    443     reencrypt/Redirect   None
```

If you inspect this route, it looks this:

YAML Definition of an autogenerated route

```
apiVersion: route.openshift.io/v1
kind: Route
metadata:
  name: frontend-gnztq
  ownerReferences:
  - apiVersion: networking.k8s.io/v1
    controller: true
    kind: Ingress
    name: frontend
    uid: 4e6c59cc-704d-4f44-b390-617d879033b6
spec:
  host: www.example.com
  path: /
  port:
    targetPort: https
  tls:
    certificate: |
      -----BEGIN CERTIFICATE-----
      [...]
      -----END CERTIFICATE-----
    insecureEdgeTerminationPolicy: Redirect
    key: |
      -----BEGIN RSA PRIVATE KEY-----
      [...]
      -----END RSA PRIVATE KEY-----
    termination: reencrypt
    destinationCACertificate: |
      -----BEGIN CERTIFICATE-----
      [...]
      -----END CERTIFICATE-----
  to:
    kind: Service
    name: frontend
```

6.8. CREATING A ROUTE USING THE DEFAULT CERTIFICATE THROUGH AN INGRESS OBJECT

If you create an Ingress object without specifying any TLS configuration, Red Hat build of MicroShift generates an insecure route. To create an Ingress object that generates a secure, edge-terminated route using the default ingress certificate, you can specify an empty TLS configuration as follows.

Prerequisites

- You have a service that you want to expose.
- You have access to the OpenShift CLI (**oc**).

Procedure

1. Create a YAML file for the Ingress object. In this example, the file is called **example-ingress.yaml**:

   YAML definition of an Ingress object

   ```
   apiVersion: networking.k8s.io/v1
   kind: Ingress
   metadata:
     name: frontend
     ...
   spec:
     rules:
       ...
     tls:
     - {} 1
   ```

   1 Use this exact syntax to specify TLS without specifying a custom certificate.

2. Create the Ingress object by running the following command:

   ```
   $ oc create -f example-ingress.yaml
   ```

Verification

- Verify that Red Hat build of MicroShift has created the expected route for the Ingress object by running the following command:

  ```
  $ oc get routes -o yaml
  ```

  Example output

  ```
  apiVersion: v1
  items:
  - apiVersion: route.openshift.io/v1
    kind: Route
    metadata:
      name: frontend-j9sdd 1
  ```

```
 ...
spec:
...
  tls: (2)
    insecureEdgeTerminationPolicy: Redirect
    termination: edge (3)
...
```

1. The name of the route includes the name of the Ingress object followed by a random suffix.
2. In order to use the default certificate, the route should not specify **spec.certificate**.
3. The route should specify the **edge** termination policy.

6.9. CREATING A ROUTE USING THE DESTINATION CA CERTIFICATE IN THE INGRESS ANNOTATION

The **route.openshift.io/destination-ca-certificate-secret** annotation can be used on an Ingress object to define a route with a custom destination CA certificate.

Prerequisites

- You may have a certificate/key pair in PEM-encoded files, where the certificate is valid for the route host.
- You may have a separate CA certificate in a PEM-encoded file that completes the certificate chain.
- You must have a separate destination CA certificate in a PEM-encoded file.
- You must have a service that you want to expose.

Procedure

1. Add the **route.openshift.io/destination-ca-certificate-secret** to the Ingress annotations:

```
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: frontend
  annotations:
    route.openshift.io/termination: "reencrypt"
    route.openshift.io/destination-ca-certificate-secret: secret-ca-cert (1)
...
```

(1) The annotation references a kubernetes secret.

2. The secret referenced in this annotation will be inserted into the generated route.

Example output

```
apiVersion: route.openshift.io/v1
```

```
kind: Route
metadata:
  name: frontend
  annotations:
    route.openshift.io/termination: reencrypt
    route.openshift.io/destination-ca-certificate-secret: secret-ca-cert
spec:
...
  tls:
    insecureEdgeTerminationPolicy: Redirect
    termination: reencrypt
    destinationCACertificate: |
      -----BEGIN CERTIFICATE-----
      [...]
      -----END CERTIFICATE-----
...
```

6.10. SECURED ROUTES

Secure routes provide the ability to use several types of TLS termination to serve certificates to the client. The following links to the OpenShift Container Platform documentation describe how to create re-encrypt, edge, and passthrough routes with custom certificates.

- Creating a re-encrypt route with a custom certificate
- Creating an edge route with a custom certificate
- Creating a passthrough route

CHAPTER 7. USING A FIREWALL

Firewalls are not required in MicroShift, but using a firewall can prevent undesired access to the MicroShift API.

7.1. ABOUT NETWORK TRAFFIC THROUGH THE FIREWALL

Firewalld is a networking service that runs in the background and responds to connection requests, creating a dynamic customizable host-based firewall. If you are using Red Hat Enterprise Linux for Edge (RHEL for Edge) with MicroShift, firewalld should already be installed and you just need to configure it. Details are provided in procedures that follow. Overall, you must explicitly allow the following OVN-Kubernetes traffic when the **firewalld** service is running:

CNI pod to CNI pod

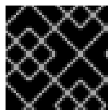
CNI pod to Host-Network pod Host-Network pod to Host-Network pod

CNI pod

The Kubernetes pod that uses the CNI network

Host-Network pod

The Kubernetes pod that uses host network You can configure the **firewalld** service by using the following procedures. In most cases, firewalld is part of RHEL for Edge installations. If you do not have firewalld, you can install it with the simple procedure in this section.



IMPORTANT

MicroShift pods must have access to the internal CoreDNS component and API servers.

Additional resources

- Required firewall settings
- Allowing network traffic through the firewall

7.2. INSTALLING THE FIREWALLD SERVICE

If you are using RHEL for Edge, firewalld should be installed. To use the service, you can simply configure it. The following procedure can be used if you do not have firewalld, but want to use it.

Install and run the **firewalld** service for MicroShift by using the following steps.

Procedure

1. Optional: Check for firewalld on your system by running the following command:

```
$ rpm -q firewalld
```

2. If the **firewalld** service is not installed, run the following command:

```
$ sudo dnf install -y firewalld
```

3. To start the firewall, run the following command:

```
$ sudo systemctl enable firewalld --now
```

7.3. REQUIRED FIREWALL SETTINGS

An IP address range for the cluster network must be enabled during firewall configuration. You can use the default values or customize the IP address range. If you choose to customize the cluster network IP address range from the default **10.42.0.0/16** setting, you must also use the same custom range in the firewall configuration.

Table 7.1. Firewall IP address settings

IP Range	Firewall rule required	Description
10.42.0.0/16	No	Host network pod access to other pods
169.254.169.1	Yes	Host network pod access to Red Hat build of MicroShift API server

The following are examples of commands for settings that are mandatory for firewall configuration:

Example commands

- Configure host network pod access to other pods:

```
$ sudo firewall-cmd --permanent --zone=trusted --add-source=10.42.0.0/16
```

- Configure host network pod access to services backed by Host endpoints, such as the Red Hat build of MicroShift API:

```
$ sudo firewall-cmd --permanent --zone=trusted --add-source=169.254.169.1
```

7.4. USING OPTIONAL PORT SETTINGS

The MicroShift firewall service allows optional port settings.

Procedure

- To add customized ports to your firewall configuration, use the following command syntax:

```
$ sudo firewall-cmd --permanent --zone=public --add-port=<port number>/<port protocol>
```

Table 7.2. Optional ports

Port(s)	Protocol(s)	Description
80	TCP	HTTP port used to serve applications through the OpenShift Container Platform router.

Port(s)	Protocol(s)	Description
443	TCP	HTTPS port used to serve applications through the OpenShift Container Platform router.
5353	UDP	mDNS service to respond for OpenShift Container Platform route mDNS hosts.
30000-32767	TCP	Port range reserved for NodePort services; can be used to expose applications on the LAN.
30000-32767	UDP	Port range reserved for NodePort services; can be used to expose applications on the LAN.
6443	TCP	HTTPS API port for the Red Hat build of MicroShift API.

The following are examples of commands used when requiring external access through the firewall to services running on MicroShift, such as port 6443 for the API server, for example, ports 80 and 443 for applications exposed through the router.

Example command

- Configuring a port for the MicroShift API server:

```
$ sudo firewall-cmd --permanent --zone=public --add-port=6443/tcp
```

To close unnecessary ports in your MicroShift instance, follow the procedure in "Closing unused or unnecessary ports to enhance network security".

Additional resources

- Closing unused or unnecessary ports to enhance network security

7.5. ADDING SERVICES TO OPEN PORTS

On a MicroShift instance, you can open services on ports by using the **firewall-cmd** command.

Procedure

1. Optional: You can view all predefined services in firewalld by running the following command

```
$ sudo firewall-cmd --get-services
```

2. To open a service that you want on a default port, run the following example command:

```
$ sudo firewall-cmd --add-service=mdns
```

7.6. ALLOWING NETWORK TRAFFIC THROUGH THE FIREWALL

You can allow network traffic through the firewall by configuring the IP address range and inserting the DNS server to allow internal traffic from pods through the network gateway.

Procedure

1. Use one of the following commands to set the IP address range:
   a. Configure the IP address range with default values by running the following command:

      ```
      $ sudo firewall-offline-cmd --permanent --zone=trusted --add-source=10.42.0.0/16
      ```

   b. Configure the IP address range with custom values by running the following command:

      ```
      $ sudo firewall-offline-cmd --permanent --zone=trusted --add-source=<custom IP range>
      ```

2. To allow internal traffic from pods through the network gateway, run the following command:

   ```
   $ sudo firewall-offline-cmd --permanent --zone=trusted --add-source=169.254.169.1
   ```

7.6.1. Applying firewall settings

To apply firewall settings, use the following one-step procedure:

Procedure

- After you have finished configuring network access through the firewall, run the following command to restart the firewall and apply the settings:

```
$ sudo firewall-cmd --reload
```

7.7. VERIFYING FIREWALL SETTINGS

After you have restarted the firewall, you can verify your settings by listing them.

Procedure

- To verify rules added in the default public zone, such as ports-related rules, run the following command:

  ```
  $ sudo firewall-cmd --list-all
  ```

- To verify rules added in the trusted zone, such as IP-range related rules, run the following command:

  ```
  $ sudo firewall-cmd --zone=trusted --list-all
  ```

7.8. OVERVIEW OF FIREWALL PORTS WHEN A SERVICE IS EXPOSED

Firewalld is often active when you run services on MicroShift. This can disrupt certain services on MicroShift because traffic to the ports might be blocked by the firewall. You must ensure that the necessary firewall ports are open if you want certain services to be accessible from outside the host. There are several options for opening your ports:

- Services of the **NodePort** and **LoadBalancer** type are automatically available with OVN-Kubernetes.
  In these cases, OVN-Kubernetes adds iptables rules so the traffic to the node IP address is delivered to the relevant ports. This is done using the PREROUTING rule chain and is then forwarded to the OVN-K to bypass the firewalld rules for local host ports and services. Iptables and firewalld are backed by nftables in RHEL 9. The nftables rules, which the iptables generates, always have priority over the rules that the firewalld generates.

- Pods with the **HostPort** parameter settings are automatically available. This also includes the **router-default** pod, which uses ports 80 and 443.
  For **HostPort** pods, the CRI-O config sets up iptables DNAT (Destination Network Address Translation) to the pod's IP address and port.

These methods function for clients whether they are on the same host or on a remote host. The iptables rules, which are added by OVN-Kubernetes and CRI-O, attach to the PREROUTING and OUTPUT chains. The local traffic goes through the OUTPUT chain with the interface set to the **lo** type. The DNAT runs before it hits filler rules in the INPUT chain.

Because the MicroShift API server does not run in CRI-O, it is subject to the firewall configurations. You can open port 6443 in the firewall to access the API server in your MicroShift cluster.

7.9. ADDITIONAL RESOURCES

- RHEL: Using and configuring firewalld

- RHEL: Viewing the current status of firewalld

7.10. KNOWN FIREWALL ISSUE

- To avoid breaking traffic flows with a firewall reload or restart, execute firewall commands before starting RHEL. The CNI driver in MicroShift makes use of iptable rules for some traffic flows, such as those using the NodePort service. The iptable rules are generated and inserted by the CNI driver, but are deleted when the firewall reloads or restarts. The absence of the iptable rules breaks traffic flows. If firewall commands have to be executed after MicroShift is running, manually restart **ovnkube-master** pod in the **openshift-ovn-kubernetes** namespace to reset the rules controlled by the CNI driver.

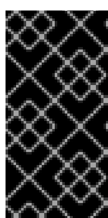
CHAPTER 8. CONFIGURING NETWORK SETTINGS FOR FULLY DISCONNECTED HOSTS

Learn how to apply networking customization and settings to run MicroShift on fully disconnected hosts. A disconnected host should be the Red Hat Enterprise Linux (RHEL) operating system, versions 9.0+, whether real or virtual, that runs without network connectivity.

8.1. PREPARING NETWORKING FOR FULLY DISCONNECTED HOSTS

Use the procedure that follows to start and run MicroShift clusters on devices running fully disconnected operating systems. A MicroShift host is considered fully disconnected if it has no external network connectivity.

Typically this means that the device does not have an attached network interface controller (NIC) to provide a subnet. These steps can also be completed on a host with a NIC that is removed after setup. You can also automate these steps on a host that does not have a NIC by using the **%post** phase of a Kickstart file.



IMPORTANT

Configuring networking settings for disconnected environments is necessary because MicroShift requires a network device to support cluster communication. To meet this requirement, you must configure MicroShift networking settings to use the "fake" IP address you assign to the system loopback device during setup.

8.1.1. Procedure summary

To run MicroShift on a disconnected host, the following steps are required:

Prepare the host

- Stop MicroShift if it is currently running and clean up changes the service has made to the network.
- Set a persistent hostname.
- Add a “fake” IP address on the loopback interface.
- Configure DNS to use the fake IP as local name server.
- Add an entry for the hostname to **/etc/hosts**.

Update the MicroShift configuration

- Define the **nodeIP** parameter as the new loopback IP address.
- Set the **.node.hostnameOverride** parameter to the persistent hostname.

For the changes to take effect

- Disable the default NIC if attached.
- Restart the host or device.

After starting, MicroShift runs using the loopback device for within-cluster communication.

8.2. RESTORING MICROSHIFT NETWORKING SETTINGS TO DEFAULT

You can remove networking customizations and return the network to default settings by stopping MicroShift and running a clean-up script.

Prerequisites

- RHEL 9 or newer.
- MicroShift 4.14 or newer.
- Access to the host CLI.

Procedure

1. Stop the MicroShift service by running the following command:

```
$ sudo systemctl stop microshift
```

2. Stop the **kubepods.slice** systemd unit by running the following command:

```
$ sudo systemctl stop kubepods.slice
```

3. MicroShift installs a helper script to undo network changes made by OVN-K. Run the cleanup script by entering the following command:

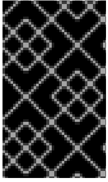
```
$ sudo /usr/bin/microshift-cleanup-data --ovn
```

8.3. CONFIGURING THE NETWORKING SETTINGS FOR FULLY DISCONNECTED HOSTS

To configure the networking settings for running MicroShift on a fully disconnected host, you must prepare the host, update the networking configuration, then restart to apply the new settings. All commands are executed from the host CLI.

Prerequisites

- RHEL 9 or newer.
- MicroShift 4.14 or newer.
- Access to the host CLI.
- A valid IP address chosen to avoid both internal and potential future external IP conflicts when running MicroShift.
- MicroShift networking settings are set to defaults.



IMPORTANT

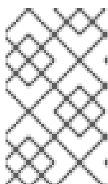
The following procedure is for use cases in which access to the MicroShift cluster is not required after devices are deployed in the field. There is no remote cluster access after the network connection is removed.

Procedure

1. Add a fake IP address to the loopback interface by running the following command:

```
$ IP="10.44.0.1" (1)
$ sudo nmcli con add type loopback con-name stable-microshift ifname lo ip4 ${IP}/32
```

(1) The fake IP address used in this example is “10.44.0.1”.



NOTE

Any valid IP works if it avoids both internal MicroShift and potential future external IP conflicts. This can be any subnet that does not collide with the MicroShift node subnet or is be accessed by other services on the device.

2. Configure the DNS interface to use the local name server by setting modifying the settings to ignore automatic DNS and reset it to the local name server:

   a. Bypass the automatic DNS by running the following command:

   ```
   $ sudo nmcli conn modify stable-microshift ipv4.ignore-auto-dns yes
   ```

   b. Point the DNS interface to use the local name server:

   ```
   $ sudo nmcli conn modify stable-microshift ipv4.dns "10.44.1.1"
   ```

3. Get the hostname of the device by running the following command:

```
$ NAME="$(hostnamectl hostname)"
```

4. Add an entry for the hostname of the node in the **/etc/hosts** file by running the following command:

```
$ echo "$IP $NAME" | sudo tee -a /etc/hosts >/dev/null
```

5. Update the MicroShift configuration file by adding the following YAML snippet to **/etc/microshift/config.yaml**:

```
sudo tee /etc/microshift/config.yaml > /dev/null <<EOF
node:
  hostnameOverride: hostnameOverride: $(echo $NAME)
  nodeIP: $(echo $IP)
EOF
```

6. MicroShift is now ready to use the loopback device for cluster communications. Finish preparing the device for offline use.

a. If the device currently has a NIC attached, disconnect the device from the network.

b. Shut down the device and disconnect the NIC.

c. Restart the device for the offline configuration to take effect.

7. Restart the MicroShift host to apply the configuration changes by running the following command:

```
$ sudo systemctl reboot (1)
```

(1) This step restarts the cluster. Wait for the greenboot health check to report the system healthy before implementing verification.

Verification

At this point, network access to the MicroShift host has been severed. If you have access to the host terminal, you can use the host CLI to verify that the cluster has started in a stable state.

1. Verify that the MicroShift cluster is running by entering the following command:

```
$ export KUBECONFIG=/var/lib/microshift/resources/kubeadmin/kubeconfig
$ sudo -E oc get pods -A
```

Example output

```
NAMESPACE                  NAME                                      READY   STATUS    RESTARTS
AGE
kube-system               csi-snapshot-controller-74d566564f-66n2f   1/1     Running   0
1m
kube-system               csi-snapshot-webhook-69bdff8879-xs6mb      1/1     Running   0
1m
openshift-dns             dns-default-dxglm                         2/2     Running   0          1m
openshift-dns             node-resolver-dbf5v                       1/1     Running   0          1m
openshift-ingress         router-default-8575d888d8-xmq9p           1/1     Running   0
1m
openshift-ovn-kubernetes   ovnkube-master-gcsx8                     4/4     Running   1          1m
openshift-ovn-kubernetes   ovnkube-node-757mf                       1/1     Running   1          1m
openshift-service-ca       service-ca-7d7c579f54-68jt4              1/1     Running   0          1m
openshift-storage          topolvm-controller-6d777f795b-bx22r      5/5     Running   0
1m
openshift-storage          topolvm-node-fcf8l                       4/4     Running   0          1m
```