# OpenShift Dedicated 4

## Networking

Configuring and managing networking in OpenShift Dedicated 4

Last Updated: 2020-06-18

# OpenShift Dedicated 4 Networking

Configuring and managing networking in OpenShift Dedicated 4

## Legal Notice



## Abstract

This document provides instructions for networking in your cluster.

# Table of Contents

# CHAPTER 1. UNDERSTANDING NETWORKING

Kubernetes ensures that Pods are able to network with each other, and allocates each Pod an IP address from an internal network. This ensures all containers within the Pod behave as if they were on the same host. Giving each Pod its own IP address means that Pods can be treated like physical hosts or virtual machines in terms of port allocation, networking, naming, service discovery, load balancing, application configuration, and migration.

NOTE

Some cloud platforms offer metadata APIs that listen on the 169.254.169.254 IP address, a link-local IP address in the IPv4 **169.254.0.0/16** CIDR block.

This CIDR block is not reachable from the pod network. Pods that need access to these IP addresses must be given host network access by setting the **spec.hostNetwork** field in the Pod spec to **true**.

If you allow a Pod host network access, you grant the Pod privileged access to the underlying network infrastructure.

## 1.1. OPENSHIFT DEDICATED DNS

If you are running multiple services, such as front-end and back-end services for use with multiple Pods, environment variables are created for user names, service IPs, and more so the front-end Pods can communicate with the back-end services. If the service is deleted and recreated, a new IP address can be assigned to the service, and requires the front-end Pods to be recreated to pick up the updated values for the service IP environment variable. Additionally, the back-end service must be created before any of the front-end Pods to ensure that the service IP is generated properly, and that it can be provided to the front-end Pods as an environment variable.

For this reason, OpenShift Dedicated has a built-in DNS so that the services can be reached by the service DNS as well as the service IP/port.

# CHAPTER 2. ACCESSING HOSTS

Learn how to create a bastion host to access OpenShift Dedicated instances and access the master nodes with secure shell (SSH) access.

## 2.1. ACCESSING HOSTS ON AMAZON WEB SERVICES IN AN INSTALLER-PROVISIONED INFRASTRUCTURE CLUSTER

The OpenShift Dedicated installer does not create any public IP addresses for any of the Amazon Elastic Compute Cloud (Amazon EC2) instances that it provisions for your OpenShift Dedicated cluster. In order to be able to SSH to your OpenShift Dedicated hosts, you must follow this procedure.

**Procedure**

1. Create a security group that allows SSH access into the virtual private cloud (VPC) created by the **openshift-install** command.

2. Create an Amazon EC2 instance on one of the public subnets the installer created.

3. Associate a public IP address with the Amazon EC2 instance that you created.
   Unlike with the OpenShift Dedicated installation, you should associate the Amazon EC2 instance you created with an SSH keypair. It does not matter what operating system you choose for this instance, as it will simply serve as an SSH bastion to bridge the internet into your OpenShift Dedicated cluster's VPC. The Amazon Machine Image (AMI) you use does matter. With Red Hat Enterprise Linux CoreOS, for example, you can provide keys via Ignition, like the installer does.

4. Once you provisioned your Amazon EC2 instance and can SSH into it, you must add the SSH key that you associated with your OpenShift Dedicated installation. This key can be different from the key for the bastion instance, but does not have to be.

   NOTE

   Direct SSH access is only recommended for disaster recovery. When the Kubernetes API is responsive, run privileged pods instead.

5. Run **oc get nodes**, inspect the output, and choose one of the nodes that is a master. The host name looks similar to **ip-10-0-1-163.ec2.internal**.

6. From the bastion SSH host you manually deployed into Amazon EC2, SSH into that master host. Ensure that you use the same SSH key you specified during the installation:

   ```
   $ ssh -i <ssh-key-path> core@<master-hostname>
   ```

# CHAPTER 3. DNS OPERATOR IN OPENSHIFT DEDICATED

The DNS Operator deploys and manages CoreDNS to provide a name resolution service to pods, enabling DNS-based Kubernetes Service discovery in OpenShift.

## 3.1. DNS OPERATOR

The DNS Operator implements the **dns** API from the **operator.openshift.io** API group. The operator deploys CoreDNS using a DaemonSet, creates a Service for the DaemonSet, and configures the kubelet to instruct pods to use the CoreDNS Service IP for name resolution.

## 3.2. VIEW THE DEFAULT DNS

Every new OpenShift Dedicated installation has a **dns.operator** named **default**.

**Procedure**

1. Use the **oc describe** command to view the default **dns**:

```
$ oc describe dns.operator/default
Name:         default
Namespace:
Labels:       <none>
Annotations:  <none>
API Version:  operator.openshift.io/v1
Kind:         DNS
...
Status:
  Cluster Domain:  cluster.local 1
  Cluster IP:      172.30.0.10 2
...
```

1. The Cluster Domain field is the base DNS domain used to construct fully qualified Pod and Service domain names.

2. The Cluster IP is the address pods query for name resolution. The IP is defined as the 10th address in the Service CIDR range.

## 3.3. USING DNS FORWARDING

You can use DNS forwarding to override the forwarding configuration identified in **etc/resolv.conf** on a per-zone basis by specifying which name server should be used for a given zone.

**Procedure**

1. Modify the DNS Operator object named **default**:

```
$ oc edit dns.operator/default
```

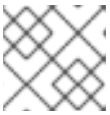
This allows the Operator to create and update the ConfigMap named **dns-default** with additional server configuration blocks based on **Server**. If none of the servers has a zone that matches the query, then name resolution falls back to the name servers that are specified in

**/etc/resolv.conf**.

### Sample DNS

```
apiVersion: operator.openshift.io/v1
kind: DNS
metadata:
  name: default
spec:
  servers:
  - name: foo-server 1
    zones: 2
      - foo.com
    forwardPlugin:
      upstreams: 3
        - 1.1.1.1
        - 2.2.2.2:5353
  - name: bar-server
    zones:
      - bar.com
      - example.com
    forwardPlugin:
      upstreams:
        - 3.3.3.3
        - 4.4.4.4:5454
```

1. **name** must comply with the **rfc6335** service name syntax.
2. **zones** must conform to the definition of a **subdomain** in **rfc1123**. The cluster domain, **cluster.local**, is an invalid **subdomain** for **zones**.
3. A maximum of 15 **upstreams** is allowed per **forwardPlugin**.



**NOTE**

If **servers** is undefined or invalid, the ConfigMap only contains the default server.

2. View the ConfigMap:

```
$ oc get configmap/dns-default -n openshift-dns -o yaml
```

### Sample DNS ConfigMap based on previous sample DNS

```
apiVersion: v1
data:
  Corefile: |
    foo.com:5353 {
        forward . 1.1.1.1 2.2.2.2:5353
    }
    bar.com:5353 example.com:5353 {
        forward . 3.3.3.3 4.4.4.4:5454 1
    }
    .:5353 {
```

```
        errors
        health
        kubernetes cluster.local in-addr.arpa ip6.arpa {
            pods insecure
            upstream
            fallthrough in-addr.arpa ip6.arpa
        }
        prometheus :9153
        forward . /etc/resolv.conf {
            policy sequential
        }
        cache 30
        reload
    }
kind: ConfigMap
metadata:
  labels:
    dns.operator.openshift.io/owning-dns: default
  name: dns-default
  namespace: openshift-dns
```

1 Changes to the **forwardPlugin** triggers a rolling update of the CoreDNS DaemonSet.

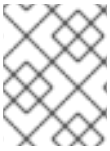
### Additional resources

- For more information on DNS forwarding, see the CoreDNS forward documentation.

# CHAPTER 4. CONFIGURING NETWORK POLICY WITH OPENSHIFT SDN

## 4.1. ABOUT NETWORK POLICY

In a cluster using a Kubernetes Container Network Interface (CNI) plug-in that supports Kubernetes network policy, network isolation is controlled entirely by NetworkPolicy Custom Resource (CR) objects. In OpenShift Dedicated 4, OpenShift SDN supports using NetworkPolicy in its default network isolation mode.



**NOTE**

The Kubernetes **v1** NetworkPolicy features are available in OpenShift Dedicated except for egress policy types and IPBlock.

**WARNING**

Network policy does not apply to the host network namespace. Pods with host networking enabled are unaffected by NetworkPolicy object rules.

By default, all Pods in a project are accessible from other Pods and network endpoints. To isolate one or more Pods in a project, you can create NetworkPolicy objects in that project to indicate the allowed incoming connections. Project administrators can create and delete NetworkPolicy objects within their own project.

If a Pod is matched by selectors in one or more NetworkPolicy objects, then the Pod will accept 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.

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: []
```

- Only allow connections from the OpenShift Dedicated Ingress Controller:
  To make a project allow only connections from the OpenShift Dedicated Ingress Controller, add the following NetworkPolicy object:

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

```
kind: NetworkPolicy
metadata:
  name: allow-from-openshift-ingress
spec:
  ingress:
  - from:
    - namespaceSelector:
        matchLabels:
          network.openshift.io/policy-group: ingress
  podSelector: {}
  policyTypes:
  - Ingress
```

If the Ingress Controller is configured with **endpointPublishingStrategy: HostNetwork**, then the Ingress Controller Pod runs on the host network. When running on the host network, the traffic from the Ingress Controller is assigned the **netid:0** Virtual Network ID (VNID). The **netid** for the namespace that is associated with the Ingress Operator is different, so the **matchLabel** in the **allow-from-openshift-ingress** network policy does not match traffic from the **default** Ingress Controller. Because the **default** namespace is assigned the **netid:0** VNID, you can allow traffic from the **default** Ingress Controller by labeling your **default** namespace with **network.openshift.io/policy-group: ingress**.

- Only accept connections from Pods within a project:
  To make Pods accept connections from other Pods in the same project, but reject all other connections from Pods in other projects, 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 samples, you can define both **allow-same-namespace** and **allow-http-and-https** policies within the same project. Thus allowing 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.

## 4.2. EXAMPLE NETWORKPOLICY OBJECT

The following annotates an example NetworkPolicy object:

```
kind: NetworkPolicy
apiVersion: extensions/v1beta1
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 describing the Pods the policy applies to. The policy object can only select Pods in the project that the NetworkPolicy object is defined.

3 A selector matching the Pods that the policy object allows ingress traffic from. The selector will match Pods in any project.

4 A list of one or more destination ports to accept traffic on.

## 4.3. CREATING A NETWORKPOLICY OBJECT

To define granular rules describing Ingress network traffic allowed for projects in your cluster, you can create NetworkPolicy objects.

### Prerequisites

- A cluster using the OpenShift SDN network plug-in with **mode: NetworkPolicy** set. This mode is the default for OpenShift SDN.
- Install the OpenShift Command-line Interface (CLI), commonly known as **oc**.
- You must log in to the cluster.

### Procedure

1. Create a policy rule:
   a. Create a **<policy-name>.yaml** file where **<policy-name>** describes the policy rule.
   b. In the file you just created define a policy object, such as in the following example:

```
kind: NetworkPolicy
apiVersion: networking.k8s.io/v1
metadata:
  name: <policy-name> 1
spec:
  podSelector:
  ingress: []
```

1 Specify a name for the policy object.

2. Run the following command to create the policy object:

```
$ oc create -f <policy-name>.yaml -n <project>
```

In the following example, a new NetworkPolicy object is created in a project named **project1**:

```
$ oc create -f default-deny.yaml -n project1
networkpolicy "default-deny" created
```

## 4.4. DELETING A NETWORKPOLICY OBJECT

You can delete a NetworkPolicy object.

**Prerequisites**

- A cluster using the OpenShift SDN network plug-in with **mode: NetworkPolicy** set. This mode is the default for OpenShift SDN.
- Install the OpenShift Command-line Interface (CLI), commonly known as **oc**.
- You must log in to the cluster.

**Procedure**

- To delete a NetworkPolicy object, run the following command:

```
$ oc delete networkpolicy -l name=<policy-name> 1
```

1 Specify the name of the NetworkPolicy object to delete.

## 4.5. VIEWING NETWORKPOLICY OBJECTS

You can list the NetworkPolicy objects in your cluster.

**Prerequisites**

- A cluster using the OpenShift SDN network plug-in with **mode: NetworkPolicy** set. This mode is the default for OpenShift SDN.
- Install the OpenShift Command-line Interface (CLI), commonly known as **oc**.
- You must log in to the cluster.

**Procedure**

- To view NetworkPolicy objects defined in your cluster, run the following command:

```
$ oc get networkpolicy
```

## 4.6. CONFIGURING MULTITENANT ISOLATION USING NETWORKPOLICY

You can configure your project to isolate it from Pods and Services in other projects.

**Prerequisites**

- A cluster using the OpenShift SDN network plug-in with **mode: NetworkPolicy** set. This mode is the default for OpenShift SDN.
- Install the OpenShift Command-line Interface (CLI), commonly known as **oc**.
- You must log in to the cluster.

**Procedure**

1. Create the following files containing NetworkPolicy object definitions:

   a. A file named **allow-from-openshift-ingress.yaml** containing the following:

```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: allow-from-openshift-ingress
spec:
  ingress:
  - from:
    - namespaceSelector:
        matchLabels:
          network.openshift.io/policy-group: ingress
  podSelector: {}
  policyTypes:
  - Ingress
```

   b. A file named **allow-from-openshift-monitoring.yaml** containing the following:

```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: allow-from-openshift-monitoring
spec:
  ingress:
  - from:
    - namespaceSelector:
        matchLabels:
          network.openshift.io/policy-group: monitoring
  podSelector: {}
  policyTypes:
  - Ingress
```

2. For each policy file, run the following command to create the NetworkPolicy object:

```
$ oc apply -f <policy-name>.yaml \ 1
  -n <project> 2
```

   1 Replace **<policy-name>** with the filename of the file containing the policy.

   2 Replace **<project>** with the name of the project to apply the NetworkPolicy object to.

3. If the **default** Ingress Controller configuration has the **spec.endpointPublishingStrategy: HostNetwork** value set, you must apply a label to the **default** OpenShift Dedicated namespace to allow network traffic between the Ingress Controller and the project:

   a. Determine if your **default** Ingress Controller uses the **HostNetwork** endpoint publishing strategy:

```
$ oc get --namespace openshift-ingress-operator ingresscontrollers/default \
  --output jsonpath='{.status.endpointPublishingStrategy.type}'
```

b. If the previous command reports the endpoint publishing strategy as **HostNetwork**, set a label on the **default** namespace:

```
$ oc label namespace default 'network.openshift.io/policy-group=ingress'
```

4. Optional: Confirm that the NetworkPolicy object exists in your current project by running the following command:

```
$ oc get networkpolicy <policy-name> -o yaml
```

In the following example, the **allow-from-openshift-ingress** NetworkPolicy object is displayed:

```
$ oc get networkpolicy allow-from-openshift-ingress -o yaml

apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: allow-from-openshift-ingress
  namespace: project1
spec:
  ingress:
  - from:
    - namespaceSelector:
        matchLabels:
          network.openshift.io/policy-group: ingress
  podSelector: {}
  policyTypes:
  - Ingress
```

# CHAPTER 5. OPENSHIFT SDN NETWORK PROVIDER

## 5.1. ABOUT OPENSHIFT SDN

OpenShift Dedicated uses a software-defined networking (SDN) approach to provide a unified cluster network that enables communication between Pods across the OpenShift Dedicated cluster. This Pod network is established and maintained by the OpenShift SDN, which configures an overlay network using Open vSwitch (OVS).

OpenShift SDN supports only the *network policy* mode, which allows project administrators to configure their own isolation policies by using NetworkPolicy objects.

## 5.2. CONFIGURING AN EGRESS FIREWALL TO CONTROL ACCESS TO EXTERNAL IP ADDRESSES

As a cluster administrator, you can create an egress firewall for a project that will restrict egress traffic leaving your OpenShift Dedicated cluster.

### 5.2.1. How an egress firewall works in a project

As a cluster administrator, you can use an *egress firewall* to limit the external hosts that some or all Pods can access from within the cluster. An egress firewall supports the following scenarios:

- A Pod can only connect to internal hosts and cannot initiate connections to the public Internet.
- A Pod can only connect to the public Internet and cannot initiate connections to internal hosts that are outside the OpenShift Dedicated cluster.
- A Pod cannot reach specified internal subnets or hosts outside the OpenShift Dedicated cluster.
- A Pod can connect to only specific external hosts.

You configure an egress firewall policy by creating an EgressNetworkPolicy Custom Resource (CR) object and specifying an IP address range in CIDR format or by specifying a DNS name. For example, you can allow one project access to a specified IP range but deny the same access to a different project. Or you can restrict application developers from updating from Python pip mirrors, and force updates to come only from approved sources.

**IMPORTANT**

You must have OpenShift SDN configured to use either the network policy or multitenant modes to configure egress firewall policy.

If you use network policy mode, egress policy is compatible with only one policy per namespace and will not work with projects that share a network, such as global projects.

**CAUTION**

Egress firewall rules do not apply to traffic that goes through routers. Any user with permission to create a Route CR object can bypass egress network policy rules by creating a route that points to a forbidden destination.

#### 5.2.1.1. Limitations of an egress firewall

An egress firewall has the following limitations:

- No project can have more than one EgressNetworkPolicy object.
- The **default** project cannot use egress network policy.
- When using the OpenShift SDN network provider in multitenant mode, the following limitations apply:
  - Global projects cannot use an egress firewall. You can make a project global by using the **oc adm pod-network make-projects-global** command.
  - Projects merged by using the **oc adm pod-network join-projects** command cannot use an egress firewall in any of the joined projects.

Violating any of these restrictions results in broken egress network policy for the project, and may cause all external network traffic to be dropped.

#### 5.2.1.2. Matching order for egress network policy rules

The egress network policy rules are evaluated in the order that they are defined, from first to last. The first rule that matches an egress connection from a Pod applies. Any subsequent rules are ignored for that connection.

#### 5.2.1.3. How Domain Name Server (DNS) resolution works

If you use DNS names in any of your egress firewall policy rules, proper resolution of the domain names is subject to the following restrictions:

- Domain name updates are polled based on the TTL (time to live) value of the domain returned by the local non-authoritative servers.
- The Pod must resolve the domain from the same local name servers when necessary. Otherwise the IP addresses for the domain known by the egress firewall controller and the Pod can be different. If the IP addresses for a host name differ, the egress firewall might not be enforced consistently.
- Because the egress firewall controller and Pods asynchronously poll the same local name server, the Pod might obtain the updated IP address before the egress controller does, which causes a race condition. Due to this current limitation, domain name usage in EgressNetworkPolicy objects is only recommended for domains with infrequent IP address changes.

NOTE

The egress firewall always allows Pods access to the external interface of the node that the Pod is on for DNS resolution.

If you use domain names in your egress firewall policy and your DNS resolution is not handled by a DNS server on the local node, then you must add egress firewall rules that allow access to your DNS server's IP addresses. if you are using domain names in your Pods.

### 5.2.2. EgressNetworkPolicy custom resource (CR) object

The following YAML describes an EgressNetworkPolicy CR object:

```
kind: EgressNetworkPolicy
apiVersion: v1
metadata:
  name: <name> 1
spec:
  egress: 2
    ...
```

1 Specify a **name** for your egress firewall policy.

2 Specify a collection of one or more egress network policy rules as described in the following section.

#### 5.2.2.1. EgressNetworkPolicy rules

The following YAML describes an egress firewall rule object. The **egress** key expects an array of one or more objects.

```
egress:
- type: <type> 1
  to: 2
    cidrSelector: <cidr> 3
    dnsName: <dns-name> 4
```

1 Specify the type of rule. The value must be either **Allow** or **Deny**.

2 Specify a value for either the **cidrSelector** key or the **dnsName** key for the rule. You cannot use both keys in a rule.

3 Specify an IP address range in CIDR format.

4 Specify a domain name.

#### 5.2.2.2. Example EgressNetworkPolicy CR object

The following example defines several egress firewall policy rules:

```
kind: EgressNetworkPolicy
apiVersion: v1
metadata:
  name: default-rules 1
spec:
  egress: 2
  - type: Allow
    to:
      cidrSelector: 1.2.3.0/24
  - type: Allow
    to:
      dnsName: www.example.com
```

```
- type: Deny
  to:
    cidrSelector: 0.0.0.0/0
```

1. The name for the policy object.
2. A collection of egress firewall policy rule objects.

### 5.2.3. Creating an egress firewall policy object

As a cluster administrator, you can create an egress firewall policy object for a project.

**IMPORTANT**

If the project already has an EgressNetworkPolicy object defined, you must edit the existing policy to make changes to the egress firewall rules.

**Prerequisites**

- A cluster that uses the OpenShift SDN network provider plug-in.
- Install the OpenShift Command-line Interface (CLI), commonly known as **oc**.
- You must log in to the cluster as a cluster administrator.

**Procedure**

1. Create a policy rule:
   a. Create a **<policy-name>.yaml** file where **<policy-name>** describes the egress policy rules.
   b. In the file you created, define an egress policy object.
2. Enter the following command to create the policy object:

   ```
   $ oc create -f <policy-name>.yaml -n <project>
   ```

   In the following example, a new EgressNetworkPolicy object is created in a project named **project1**:

   ```
   $ oc create -f default-rules.yaml -n project1
   egressnetworkpolicy.network.openshift.io/default-rules created
   ```

3. Optional: Save the **<policy-name>.yaml** so that you can make changes later.

## 5.3. EDITING AN EGRESS FIREWALL FOR A PROJECT

As a cluster administrator, you can modify network traffic rules for an existing egress firewall.

### 5.3.1. Editing an EgressNetworkPolicy object

As a cluster administrator, you can update the egress firewall for a project.

**Prerequisites**

- A cluster using the OpenShift SDN network plug-in.
- Install the OpenShift Command-line Interface (CLI), commonly known as **oc**.
- You must log in to the cluster as a cluster administrator.

**Procedure**

To edit an existing egress network policy object for a project, complete the following steps:

1. Find the name of the EgressNetworkPolicy object for the project. Replace **<project>** with the name of the project.

```
$ oc get -n <project> egressnetworkpolicy
```

2. Optional: If you did not save a copy of the EgressNetworkPolicy object when you created the egress network firewall, enter the following command to create a copy.

```
$ oc get -n <project> \ 1
  egressnetworkpolicy <name> \ 2
  -o yaml > <filename>.yaml 3
```

1. Replace **<project>** with the name of the project
2. Replace **<name>** with the name of the object.
3. Replace **<filename>** with the name of the file to save the YAML.

3. Enter the following command to replace the EgressNetworkPolicy object. Replace **<filename>** with the name of the file containing the updated EgressNetworkPolicy object.

```
$ oc replace -f <filename>.yaml
```

### 5.3.2. EgressNetworkPolicy custom resource (CR) object

The following YAML describes an EgressNetworkPolicy CR object:

```
kind: EgressNetworkPolicy
apiVersion: v1
metadata:
  name: <name> 1
spec:
  egress: 2
    ...
```

1. Specify a **name** for your egress firewall policy.
2. Specify a collection of one or more egress network policy rules as described in the following section.

#### 5.3.2.1. EgressNetworkPolicy rules

The following YAML describes an egress firewall rule object. The **egress** key expects an array of one or more objects.

```
egress:
- type: <type> 1
  to: 2
    cidrSelector: <cidr> 3
    dnsName: <dns-name> 4
```

1. Specify the type of rule. The value must be either **Allow** or **Deny**.
2. Specify a value for either the **cidrSelector** key or the **dnsName** key for the rule. You cannot use both keys in a rule.
3. Specify an IP address range in CIDR format.
4. Specify a domain name.

#### 5.3.2.2. Example EgressNetworkPolicy CR object

The following example defines several egress firewall policy rules:

```
kind: EgressNetworkPolicy
apiVersion: v1
metadata:
  name: default-rules 1
spec:
  egress: 2
  - type: Allow
    to:
      cidrSelector: 1.2.3.0/24
  - type: Allow
    to:
      dnsName: www.example.com
  - type: Deny
    to:
      cidrSelector: 0.0.0.0/0
```

1. The name for the policy object.
2. A collection of egress firewall policy rule objects.

## 5.4. REMOVING AN EGRESS FIREWALL FROM A PROJECT

As a cluster administrator, you can remove an egress firewall from a project to remove all restrictions on network traffic from the project that leaves the OpenShift Dedicated cluster.

### 5.4.1. Removing an EgressNetworkPolicy object

As a cluster administrator, you can remove an egress firewall from a project.

### Prerequisites

- A cluster using the OpenShift SDN network plug-in.
- Install the OpenShift Command-line Interface (CLI), commonly known as **oc**.
- You must log in to the cluster as a cluster administrator.

### Procedure

To remove an egress network policy object for a project, complete the following steps:

1. Find the name of the EgressNetworkPolicy object for the project. Replace **<project>** with the name of the project.

```
$ oc get -n <project> egressnetworkpolicy
```

2. Enter the following command to delete the EgressNetworkPolicy object. Replace **<project>** with the name of the project and **<name>** with the name of the object.

```
$ oc delete -n <project> egressnetworkpolicy <name>
```

# CHAPTER 6. CONFIGURING ROUTES

## 6.1. ROUTE CONFIGURATION

### 6.1.1. Configuring route timeouts

You can configure the default timeouts for an existing route when you have services in need of a low timeout, which is required for Service Level Availability (SLA) purposes, or a high timeout, for cases with a slow back end.

#### Prerequisites

- You need a deployed Ingress Controller on a running cluster.

#### Procedure

1. Using the **oc annotate** command, add the timeout to the route:

```
$ oc annotate route <route_name> \
    --overwrite haproxy.router.openshift.io/timeout=<timeout><time_unit> 1
```

1 Supported time units are microseconds (us), milliseconds (ms), seconds (s), minutes (m), hours (h), or days (d).

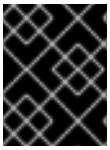
The following example sets a timeout of two seconds on a route named **myroute**:

```
$ oc annotate route myroute --overwrite haproxy.router.openshift.io/timeout=2s
```

### 6.1.2. Enabling HTTP strict transport security

HTTP Strict Transport Security (HSTS) policy is a security enhancement, which ensures that only HTTPS traffic is allowed on the host. Any HTTP requests are dropped by default. This is useful for ensuring secure interactions with websites, or to offer a secure application for the user's benefit.

When HSTS is enabled, HSTS adds a Strict Transport Security header to HTTPS responses from the site. You can use the **insecureEdgeTerminationPolicy** value in a route to redirect to send HTTP to HTTPS. However, when HSTS is enabled, the client changes all requests from the HTTP URL to HTTPS before the request is sent, eliminating the need for a redirect. This is not required to be supported by the client, and can be disabled by setting **max-age=0**.



IMPORTANT

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

#### Procedure

- To enable HSTS on a route, add the **haproxy.router.openshift.io/hsts_header** value to the edge terminated or re-encrypt route:

```
apiVersion: v1
```

```
kind: Route
metadata:
  annotations:
    haproxy.router.openshift.io/hsts_header: max-age=31536000;includeSubDomains;preload
```
(1) (2) (3)

(1) **max-age** is the only required parameter. It measures the length of time, in seconds, that the HSTS policy is in effect. The client updates **max-age** whenever a response with a HSTS header is received from the host. When **max-age** times out, the client discards the policy.

(2) **includeSubDomains** is optional. When included, it tells the client that all subdomains of the host are to be treated the same as the host.

(3) **preload** is optional. When **max-age** is greater than 0, then including **preload** in **haproxy.router.openshift.io/hsts_header** allows 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, before they have interacted with the site. Without **preload** set, browsers must have interacted with the site over HTTPS to get the header.

### 6.1.3. Troubleshooting throughput issues

Sometimes applications deployed through OpenShift Dedicated can cause network throughput issues such as unusually high latency between specific services.

Use the following methods to analyze performance issues if Pod logs do not reveal any cause of the problem:

- 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 OpenShift Dedicated 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.

  tcpdump generates a file at **/tmp/dump.pcap** containing all traffic between these two Pods. Ideally, 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. Run the tool from the Pods first, then from the nodes, to locate any bottlenecks.

### 6.1.4. Using cookies to keep route statefulness

OpenShift Dedicated 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.

OpenShift Dedicated 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.

#### 6.1.4.1. 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. By deleting the cookie it can force the next request to re-choose an endpoint. So, if a server was overloaded it tries to remove the requests from the client and redistribute them.

**Procedure**

1. Annotate the route with the desired cookie name:

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

   For example, to annotate the cookie name of **my_cookie** to the **my_route** with the annotation of **my_cookie_annotation**:

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

2. Save the cookie, and access the route:

```
$ curl $my_route -k -c /tmp/my_cookie
```

### 6.1.5. Route-specific annotations

The Ingress Controller can set the default options for all the routes it exposes. An individual route can override some of these defaults by providing specific configurations in its annotations.

**Table 6.1. Route annotations**

| Variable | Description | Environment variable used as default |
|---|---|---|
| **haproxy.router.openshift.io/balance** | Sets the load-balancing algorithm. Available options are **source**, **roundrobin**, and **leastconn**. | **ROUTER_TCP_BALANCE_SCHEME** for passthrough routes. Otherwise, use **ROUTER_LOAD_BALANCE_ALGORITHM**. |

| Variable | Description | Environment variable used as default |
|---|---|---|
| **haproxy.router.openshift.io/disable_cookies** | Disables the use of cookies to track related connections. If set to **true** or **TRUE**, the balance algorithm is used to choose which back-end serves connections for each incoming HTTP request. | |
| **router.openshift.io/cookie_name** | Specifies an optional cookie to use for this route. The name must consist of any combination of upper and lower case letters, digits, "_", and "-". The default is the hashed internal key name for the route. | |
| **haproxy.router.openshift.io/pod-concurrent-connections** | Sets the maximum number of connections that are allowed to a backing pod from a router. Note: if there are multiple pods, each can have this many connections. But if you have multiple routers, there is no coordination among them, each may connect this many times. If not set, or set to 0, there is no limit. | |
| **haproxy.router.openshift.io/rate-limit-connections** | Setting **true** or **TRUE** to enables rate limiting functionality. | |
| **haproxy.router.openshift.io/rate-limit-connections.concurrent-tcp** | Limits the number of concurrent TCP connections shared by an IP address. | |
| **haproxy.router.openshift.io/rate-limit-connections.rate-http** | Limits the rate at which an IP address can make HTTP requests. | |
| **haproxy.router.openshift.io/rate-limit-connections.rate-tcp** | Limits the rate at which an IP address can make TCP connections. | |
| **haproxy.router.openshift.io/timeout** | Sets a server-side timeout for the route. (TimeUnits) | **ROUTER_DEFAULT_SERVER_TIMEOUT** |
| **router.openshift.io/haproxy.health.check.interval** | Sets the interval for the back-end health checks. (TimeUnits) | **ROUTER_BACKEND_CHECK_INTERVAL** |

| Variable | Description | Environment variable used as default |
|---|---|---|
| **haproxy.router.openshift.io/ip_whitelist** | Sets a whitelist for the route. | |
| **haproxy.router.openshift.io/hsts_header** | Sets a Strict-Transport-Security header for the edge terminated or re-encrypt route. | |

NOTE

Environment variables can not be edited.

### A route setting custom timeout

```
apiVersion: v1
kind: Route
metadata:
  annotations:
    haproxy.router.openshift.io/timeout: 5500ms (1)
...
```

(1) Specifies the new timeout with HAProxy supported units (**us**, **ms**, **s**, **m**, **h**, **d**). If the unit is not provided, **ms** is the default.

NOTE

Setting a server-side timeout value for passthrough routes too low can cause WebSocket connections to timeout frequently on that route.

## 6.2. SECURED ROUTES

The following sections describe how to create re-encrypt and edge routes with custom certificates.

IMPORTANT

If you create routes in Microsoft Azure through public endpoints, the resource names are subject to restriction. You cannot create resources that use certain terms. For a list of terms that Azure restricts, see Resolve reserved resource name errors in the Azure documentation.

### 6.2.1. Creating a re-encrypt route with a custom certificate

You can configure a secure route using reencrypt TLS termination with a custom certificate by using the **oc create route** command.

#### Prerequisites

- You must 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** resource that you want to expose.

NOTE

Password protected key files are not supported. To remove a passphrase from a key file, use the following command:

```
$ openssl rsa -in password_protected_tls.key -out tls.key
```

### Procedure

This procedure creates a **Route** resource with a custom certificate and reencrypt TLS termination. The following assumes that the certificate/key pair are in the **tls.crt** and **tls.key** files in the current working directory. You must also specify a destination CA certificate to enable the Ingress Controller to trust the service's certificate. You may also specify a CA certificate if needed to complete the certificate chain. Substitute the actual path names for **tls.crt**, **tls.key**, **cacert.crt**, and (optionally) **ca.crt**. Substitute the name of the **Service** resource that you want to expose for **frontend**. Substitute the appropriate host name for **www.example.com**.

- Create a secure **Route** resource using reencrypt TLS termination and a custom certificate:

```
$ oc create route reencrypt --service=frontend --cert=tls.crt --key=tls.key --dest-ca-
cert=destca.crt --ca-cert=ca.crt --hostname=www.example.com
```

If you examine the resulting **Route** resource, it should look similar to the following:

#### YAML Definition of the Secure Route

```
apiVersion: v1
kind: Route
metadata:
  name: frontend
spec:
  host: www.example.com
  to:
    kind: Service
    name: frontend
  tls:
    termination: reencrypt
    key: |-
      -----BEGIN PRIVATE KEY-----
      [...]
      -----END PRIVATE KEY-----
    certificate: |-
      -----BEGIN CERTIFICATE-----
      [...]
      -----END CERTIFICATE-----
```

```
caCertificate: |-
  -----BEGIN CERTIFICATE-----
  [...]
  -----END CERTIFICATE-----
destinationCACertificate: |-
  -----BEGIN CERTIFICATE-----
  [...]
  -----END CERTIFICATE-----
```

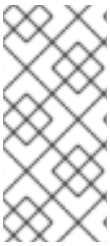
See **oc create route reencrypt --help** for more options.

### 6.2.2. Creating an edge route with a custom certificate

You can configure a secure route using edge TLS termination with a custom certificate by using the **oc create route** command. With an edge route, the Ingress Controller terminates TLS encryption before forwarding traffic to the destination Pod. The route specifies the TLS certificate and key that the Ingress Controller uses for the route.

#### Prerequisites

- You must 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 **Service** resource that you want to expose.



NOTE

Password protected key files are not supported. To remove a passphrase from a key file, use the following command:

```
$ openssl rsa -in password_protected_tls.key -out tls.key
```

#### Procedure

This procedure creates a **Route** resource with a custom certificate and edge TLS termination. The following assumes that the certificate/key pair are in the **tls.crt** and **tls.key** files in the current working directory. You may also specify a CA certificate if needed to complete the certificate chain. Substitute the actual path names for **tls.crt**, **tls.key**, and (optionally) **ca.crt**. Substitute the name of the **Service** resource that you want to expose for **frontend**. Substitute the appropriate host name for **www.example.com**.

- Create a secure **Route** resource using edge TLS termination and a custom certificate.

```
$ oc create route edge --service=frontend --cert=tls.crt --key=tls.key --ca-cert=ca.crt --
hostname=www.example.com
```

If you examine the resulting **Route** resource, it should look similar to the following:

#### YAML Definition of the Secure Route

```
apiVersion: v1
```

```
kind: Route
metadata:
  name: frontend
spec:
  host: www.example.com
  to:
    kind: Service
    name: frontend
  tls:
    termination: edge
    key: |-
      -----BEGIN PRIVATE KEY-----
      [...]
      -----END PRIVATE KEY-----
    certificate: |-
      -----BEGIN CERTIFICATE-----
      [...]
      -----END CERTIFICATE-----
    caCertificate: |-
      -----BEGIN CERTIFICATE-----
      [...]
      -----END CERTIFICATE-----
```

See **oc create route edge --help** for more options.

# CHAPTER 7. CONFIGURING INGRESS CLUSTER TRAFFIC

## 7.1. CONFIGURING INGRESS CLUSTER TRAFFIC USING AN INGRESS CONTROLLER

OpenShift Dedicated provides methods for communicating from outside the cluster with services running in the cluster. This method uses an Ingress Controller.

### 7.1.1. Creating a project and service

If the project and service that you want to expose do not exist, first create the project, then the service.

If the project and service already exist, skip to the procedure on exposing the service to create a route.

#### Prerequisites

- Install the **oc** CLI and log in as a cluster administrator.

#### Procedure

1. Create a new project for your service:

   ```
   $ oc new-project <project_name>
   ```

   For example:

   ```
   $ oc new-project myproject
   ```

2. Use the **oc new-app** command to create a service. For example:

   ```
   $ oc new-app \
       -e MYSQL_USER=admin \
       -e MYSQL_PASSWORD=redhat \
       -e MYSQL_DATABASE=mysqldb \
       registry.redhat.io/rhscl/mysql-80-rhel7
   ```

3. Run the following command to see that the new service is created:

   ```
   $ oc get svc -n myproject
   NAME             TYPE        CLUSTER-IP     EXTERNAL-IP   PORT(S)    AGE
   mysql-80-rhel7   ClusterIP   172.30.63.31   <none>        3306/TCP   4m55s
   ```

   By default, the new service does not have an external IP address.

### 7.1.2. Exposing the service by creating a route

You can expose the service as a route by using the **oc expose** command.

#### Procedure

To expose the service:

1. Log in to OpenShift Dedicated.
2. Log in to the project where the service you want to expose is located:

```
$ oc project project1
```

3. Run the following command to expose the route:

```
$ oc expose service <service_name>
```

For example:

```
$ oc expose service mysql-80-rhel7
route "mysql-80-rhel7" exposed
```

4. Use a tool, such as cURL, to make sure you can reach the service using the cluster IP address for the service:

```
$ curl <pod_ip>:<port>
```

For example:

```
$ curl 172.30.131.89:3306
```

The examples in this section use a MySQL service, which requires a client application. If you get a string of characters with the **Got packets out of order** message, you are connected to the service.

If you have a MySQL client, log in with the standard CLI command:

```
$ mysql -h 172.30.131.89 -u admin -p
Enter password:
Welcome to the MariaDB monitor.  Commands end with ; or \g.

MySQL [(none)]>
```