OpenShift Container Platform 4.1

Networking

Configuring and managing cluster networking

Last Updated: 2020-04-02

OpenShift Container Platform 4.1 Networking

Configuring and managing cluster networking

Legal Notice

Abstract

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

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.

1.1. OPENSHIFT CONTAINER PLATFORM 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 Container Platform 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 Container Platform 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 Container Platform 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 Container Platform cluster. In order to be able to SSH to your OpenShift Container Platform 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 Container Platform 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 Container Platform 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 Container Platform 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. CLUSTER NETWORK OPERATOR IN OPENSHIFT CONTAINER PLATFORM

The Cluster Network Operator (CNO) deploys and manages the cluster network components on an OpenShift Container Platform cluster, including the Container Network Interface (CNI) Software Defined Networking (SDN) plug-in selected for the cluster during installation.

3.1. CLUSTER NETWORK OPERATOR

The Cluster Network Operator implements the **network** API from the **operator.openshift.io** API group. The Operator deploys the OpenShift SDN plug-in, or a different SDN plug-in if selected during cluster installation, using a DaemonSet.

Procedure

The Cluster Network Operator is deployed during installation as a Kubernetes **Deployment**.

1. Run the following command to view the Deployment status:

```
$ oc get -n openshift-network-operator deployment/network-operator

NAME               READY   UP-TO-DATE   AVAILABLE   AGE
network-operator   1/1     1            1           56m
```

2. Run the following command to view the state of the Cluster Network Operator:

```
$ oc get clusteroperator/network

NAME      VERSION   AVAILABLE   PROGRESSING   DEGRADED   SINCE
network   4.1.0     True        False         False      50m
```

The following fields provide information about the status of the operator: **AVAILABLE**, **PROGRESSING**, and **DEGRADED**. The **AVAILABLE** field is **True** when the Cluster Network Operator reports an available status condition.

3.2. VIEWING THE CLUSTER NETWORK CONFIGURATION

Every new OpenShift Container Platform installation has a **network.config** object named **cluster**.

Procedure

- Use the **oc describe** command to view the cluster network configuration:

```
$ oc describe network.config/cluster

Name:         cluster
Namespace:
Labels:       <none>
Annotations:  <none>
API Version:  config.openshift.io/v1
Kind:         Network
Metadata:
  Self Link:           /apis/config.openshift.io/v1/networks/cluster
```

```
Spec: ①
  Cluster Network:
    Cidr:         10.128.0.0/14
    Host Prefix:  23
  Network Type:   OpenShiftSDN
  Service Network:
    172.30.0.0/16
Status: ②
  Cluster Network:
    Cidr:               10.128.0.0/14
    Host Prefix:        23
  Cluster Network MTU:  8951
  Network Type:         OpenShiftSDN
  Service Network:
    172.30.0.0/16
Events:  <none>
```

① The **Spec** field displays the configured state of the cluster network.

② The **Status** field displays the current state of the cluster network configuration.

3.3. VIEWING CLUSTER NETWORK OPERATOR STATUS

You can inspect the status and view the details of the Cluster Network Operator using the **oc describe** command.

Procedure

- Run the following command to view the status of the Cluster Network Operator:

```
$ oc describe clusteroperators/network
```

3.4. VIEWING CLUSTER NETWORK OPERATOR LOGS

You can view Cluster Network Operator logs by using the **oc logs** command.

Procedure

- Run the following command to view the logs of the Cluster Network Operator:

```
$ oc logs --namespace=openshift-network-operator deployment/network-operator
```

3.5. CLUSTER NETWORK OPERATOR CUSTOM RESOURCE (CR)

The cluster network configuration in the **Network.operator.openshift.io** custom resource (CR) stores the configuration settings for the Cluster Network Operator (CNO).

The following CR displays the default configuration for the CNO and explains both the parameters you can configure and valid parameter values:

Cluster Network Operator CR

```
apiVersion: operator.openshift.io/v1
kind: Network
spec:
  clusterNetwork: ①
  - cidr: 10.128.0.0/14
    hostPrefix: 23
  serviceNetwork: ②
  - 172.30.0.0/16
  defaultNetwork:
    type: OpenShiftSDN ③
    openshiftSDNConfig: ④
      mode: NetworkPolicy ⑤
      mtu: 1450 ⑥
      vxlanPort: 4789 ⑦
  kubeProxyConfig: ⑧
    iptablesSyncPeriod: 30s ⑨
    proxyArguments:
      iptables-min-sync-period: ⑩
      - 30s
```

① A list specifying the blocks of IP addresses from which Pod IPs are allocated and the subnet prefix length assigned to each individual node.

② A block of IP addresses for services. The OpenShift SDN Container Network Interface (CNI) plug-in supports only a single IP address block for the service network.

③ The Software Defined Networking (SDN) plug-in being used. OpenShift SDN is the only plug-in supported in OpenShift Container Platform 4.1.

④ OpenShift SDN specific configuration parameters.

⑤ The isolation mode for the OpenShift SDN CNI plug-in.

⑥ MTU for the VXLAN overlay network. This value is normally configured automatically.

⑦ The port to use for all VXLAN packets. The default value is **4789**.

⑧ The parameters for this object specify the Kubernetes network proxy (kube-proxy) configuration.

⑨ The refresh period for **iptables** rules. The default value is **30s**. Valid suffixes include **s**, **m**, and **h** and are described in the Go time package documentation.

⑩ The minimum duration before refreshing **iptables** rules. This parameter ensures that the refresh does not happen too frequently. Valid suffixes include **s**, **m**, and **h** and are described in the Go time package

CHAPTER 4. DNS OPERATOR IN OPENSHIFT CONTAINER PLATFORM

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

4.1. 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.

Procedure

The DNS Operator is deployed during installation as a Kubernetes **Deployment**.

1. Use the **oc get** command to view the Deployment status:

   ```
   $ oc get -n openshift-dns-operator deployment/dns-operator
   NAME           READY     UP-TO-DATE   AVAILABLE   AGE
   dns-operator   1/1       1            1           23h
   ```

   ClusterOperator is the Custom Resource object which holds the current state of an operator. This object is used by operators to convey their state to the rest of the cluster.

2. Use the **oc get** command to view the state of the DNS Operator:

   ```
   $ oc get clusteroperator/dns
   NAME      VERSION     AVAILABLE   PROGRESSING   DEGRADED   SINCE
   dns       4.1.0-0.11  True        False         False      92m
   ```

   AVAILABLE, **PROGRESSING** and **DEGRADED** provide information about the status of the operator. **AVAILABLE** is **True** when at least 1 pod from the CoreDNS DaemonSet is reporting an **Available** status condition.

4.2. VIEW THE DEFAULT DNS

Every new OpenShift Container Platform installation has a **dns.operator** named **default**. It cannot be customized, replaced, or supplemented with additional **dnses**.

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.

2. To find the Service CIDR of your cluster, use the **oc get** command:

```
$ oc get networks.config/cluster -o jsonpath='{$.status.serviceNetwork}'
[172.30.0.0/16]
```

NOTE

Configuration of the CoreDNS Corefile or Kubernetes plugin is not supported.

4.3. DNS OPERATOR STATUS

You can inspect the status and view the details of the DNS Operator using the **oc describe** command.

Procedure

View the status of the DNS Operator:

```
$ oc describe clusteroperators/dns
```

4.4. DNS OPERATOR LOGS

You can view DNS Operator logs by using the **oc logs** command.

Procedure

View the logs of the DNS Operator:

```
$ oc logs --namespace=openshift-dns-operator deployment/dns-operator
```

CHAPTER 5. INGRESS OPERATOR IN OPENSHIFT CONTAINER PLATFORM

The Ingress Operator implements the **ingresscontroller** API and is the component responsible for enabling external access to OpenShift Container Platform cluster services. The operator makes this possible by deploying and managing one or more HAProxy-based Ingress Controllers to handle routing. You can use the Ingress Operator to route traffic by specifying OpenShift Container Platform **Route** and Kubernetes **Ingress** resources.

5.1. THE INGRESS CONFIGURATION ASSET

The installation program generates an asset with an **Ingress** resource in the **config.openshift.io** API group, **cluster-ingress-02-config.yml**.

YAML Definition of the Ingress resource

```
apiVersion: config.openshift.io/v1
kind: Ingress
metadata:
  name: cluster
spec:
  domain: apps.openshiftdemos.com
```

The installation program stores this asset in the **cluster-ingress-02-config.yml** file in the **manifests/** directory. This **Ingress** resource defines the cluster-wide configuration for Ingress. This Ingress configuration is used as follows:

- The Ingress Operator uses the domain from the cluster Ingress configuration as the domain for the default Ingress Controller.
- The OpenShift API server operator uses the domain from the cluster Ingress configuration as the domain used when generating a default host for a **Route** resource that does not specify an explicit host.

5.2. INGRESS CONTROLLER CONFIGURATION PARAMETERS

The **ingresscontrollers.operator.openshift.io** resource offers the following configuration parameters.

Parameter	Description

Parameter	Description
domain	**domain** is a DNS name serviced by the Ingress controller and is used to configure multiple features: ● For the **LoadBalancerService** endpoint publishing strategy, **domain** is used to configure DNS records. See **endpointPublishingStrategy**. ● When using a generated default certificate, the certificate is valid for **domain** and its **subdomains**. See **defaultCertificate**. ● The value is published to individual Route statuses so that users know where to target external DNS records. The **domain** value must be unique among all Ingress controllers and cannot be updated. If empty, the default value is **ingress.config.openshift.io/cluster .spec.domain**.
replicas	**replicas** is the desired number of Ingress controller replicas. If not set, the default value is **2**.
endpointPublishingStrategy	**endpointPublishingStrategy** is used to publish the Ingress controller endpoints to other networks, enable load balancer integrations, and provide access to other systems. If not set, the default value is based on **infrastructure.config.openshift.io/cluster .status.platform**: ● AWS: **LoadBalancerService** (with external scope) ● Azure: **LoadBalancerService** (with external scope) ● GCP: **LoadBalancerService** (with external scope) ● Other: **HostNetwork**. The **endpointPublishingStrategy** value cannot be updated.
defaultCertificate	The **defaultCertificate** value is a reference to a secret that contains the default certificate that is served by the Ingress controller. When Routes do not specify their own certificate, **defaultCertificate** is used. The secret must contain the following keys and data: * **tls.crt**: certificate file contents * **tls.key**: key file contents If not set, a wildcard certificate is automatically generated and used. The certificate is valid for the Ingress controller **domain** and **subdomains**, and the generated certificate's CA is automatically integrated with the cluster's trust store. The in-use certificate, whether generated or user-specified, is automatically integrated with OpenShift Container Platform built-in OAuth server.

<table>
<thead>
<tr><th>Parameter</th><th>Description</th></tr>
</thead>
<tbody>
<tr><td><b>namespaceSelector</b></td><td><b>namespaceSelector</b> is used to filter the set of namespaces serviced by the Ingress controller. This is useful for implementing shards.</td></tr>
<tr><td><b>routeSelector</b></td><td><b>routeSelector</b> is used to filter the set of Routes serviced by the Ingress controller. This is useful for implementing shards.</td></tr>
<tr><td><b>nodePlacement</b></td><td>

nodePlacement enables explicit control over the scheduling of the Ingress controller.

If not set, the defaults values are used.

NOTE

The **nodePlacement** parameter includes two parts, **nodeSelector** and **tolerations**. For example:

```
nodePlacement:
 nodeSelector:
   matchLabels:
     beta.kubernetes.io/os: linux
 tolerations:
 - effect: NoSchedule
   operator: Exists
```

</td></tr>
</tbody>
</table>

NOTE

All parameters are optional.

5.2.1. Ingress controller endpoint publishing strategy

An Ingress controller with the **HostNetwork** endpoint publishing strategy can have only one Pod replica per node. If you want *n* replicas, you must use at least *n* nodes where those replicas can be scheduled. Because each Pod replica requests ports **80** and **443** on the node host where it is scheduled, a replica cannot be scheduled to a node if another Pod on the same node is using those ports.

5.3. VIEW THE DEFAULT INGRESS CONTROLLER

The Ingress Operator is a core feature of OpenShift Container Platform and is enabled out of the box.

Every new OpenShift Container Platform installation has an **ingresscontroller** named default. It can be supplemented with additional Ingress Controllers. If the default **ingresscontroller** is deleted, the Ingress Operator will automatically recreate it within a minute.

Procedure

- View the default Ingress Controller:

```
$ oc describe --namespace=openshift-ingress-operator ingresscontroller/default
```

5.4. VIEW INGRESS OPERATOR STATUS

You can view and inspect the status of your Ingress Operator.

Procedure

- View your Ingress Operator status:

```
$ oc describe clusteroperators/ingress
```

5.5. VIEW INGRESS CONTROLLER LOGS

You can view your Ingress Controller logs.

Procedure

- View your Ingress Controller logs:

```
$ oc logs --namespace=openshift-ingress-operator deployments/ingress-operator
```

5.6. VIEW INGRESS CONTROLLER STATUS

Your can view the status of a particular Ingress Controller.

Procedure

- View the status of an Ingress Controller:

```
$ oc describe --namespace=openshift-ingress-operator ingresscontroller/<name>
```

5.7. SETTING A CUSTOM DEFAULT CERTIFICATE

As an administrator, you can configure an Ingress Controller to use a custom certificate by creating a **Secret** resource and editing the **IngressController** custom resource (CR).

Prerequisites

- You must have a certificate/key pair in PEM-encoded files, where the certificate is signed by a trusted certificate authority and valid for the Ingress domain.
- You must have an **IngressController** CR. You may use the default one:

```
$ oc --namespace openshift-ingress-operator get ingresscontrollers
NAME      AGE
default   10m
```

WARNING

If the default certificate is replaced, it **must** be signed by a public certificate authority already included in the CA bundle as provided by the container userspace.

NOTE

If you have intermediate certificates, they must be included in the **tls.crt** file of the secret containing a custom default certificate. Order matters when specifying a certificate; list your intermediate certificate(s) after any server certificate(s).

Procedure

The following assumes that the custom certificate and key pair are in the **tls.crt** and **tls.key** files in the current working directory. Substitute the actual path names for **tls.crt** and **tls.key**. You also may substitute another name for **custom-certs-default** when creating the **Secret** resource and referencing it in the **IngressController** CR.

NOTE

This action will cause the Ingress Controller to be redeployed, using a rolling deployment strategy.

1. Create a **Secret** resource containing the custom certificate in the **openshift-ingress** namespace using the **tls.crt** and **tls.key** files.

```
$ oc --namespace openshift-ingress create secret tls custom-certs-default --cert=tls.crt --
key=tls.key
```

2. Update the **IngressController** CR to reference the new certificate secret:

```
$ oc patch --type=merge --namespace openshift-ingress-operator ingresscontrollers/default \
  --patch '{"spec":{"defaultCertificate":{"name":"custom-certs-default"}}}'
```

3. Verify the update was effective:

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

The output should look like:

```
map[name:custom-certs-default]
```

The certificate secret name should match the value used to update the CR.

Once the **IngressController** CR has been modified, the Ingress Operator will update the Ingress Controller's deployment to use the custom certificate.

5.8. SCALING AN INGRESS CONTROLLER

Manually scale an Ingress Controller to meeting routing performance or availability requirements such as the requirement to increase throughput. **oc** commands are used to scale the **IngressController** resource. The following procedure provides an example for scaling up the default **IngressController**.

Procedure

1. View the current number of available replicas for the default **IngressController**:

```
$ oc get -n openshift-ingress-operator ingresscontrollers/default -o
jsonpath='{$.status.availableReplicas}'
2
```

2. Scale the default **IngressController** to the desired number of replicas using the **oc patch** command. The following example scales the default **IngressController** to 3 replicas:

```
$ oc patch -n openshift-ingress-operator ingresscontroller/default --patch '{"spec":{"replicas":
3}}' --type=merge
ingresscontroller.operator.openshift.io/default patched
```

3. Verify that the default **IngressController** scaled to the number of replicas that you specified:

```
$ oc get -n openshift-ingress-operator ingresscontrollers/default -o
jsonpath='{$.status.availableReplicas}'
3
```

> **NOTE**
>
> Scaling is not an immediate action, as it takes time to create the desired number of replicas.

5.9. CONFIGURING INGRESS CONTROLLER SHARDING BY USING ROUTE LABELS

Ingress Controller sharding by using route labels means that the Ingress Controller serves any route in any namespace that is selected by the route selector.

Ingress Controller sharding is useful when balancing incoming traffic load among a set of Ingress Controllers and when isolating traffic to a specific Ingress Controller. For example, company A goes to one Ingress Controller and company B to another.

Procedure

1. Edit the **router-internal.yaml** file:

```
# cat router-internal.yaml
apiVersion: v1
items:
- apiVersion: operator.openshift.io/v1
  kind: IngressController
  metadata:
    name: sharded
    namespace: openshift-ingress-operator
  spec:
    domain: <apps-sharded.basedomain.example.net>
    nodePlacement:
      nodeSelector:
        matchLabels:
          node-role.kubernetes.io/worker: ""
    routeSelector:
      matchLabels:
        type: sharded
  status: {}
kind: List
metadata:
  resourceVersion: ""
  selfLink: ""
```

2. Apply the Ingress Controller **router-internal.yaml** file:

```
# oc apply -f router-internal.yaml
```

The Ingress Controller selects routes in any namespace that have the label **type: sharded**.

5.10. CONFIGURING INGRESS CONTROLLER SHARDING BY USING NAMESPACE LABELS

Ingress Controller sharding by using namespace labels means that the Ingress Controller serves any route in any namespace that is selected by the namespace selector.

Ingress Controller sharding is useful when balancing incoming traffic load among a set of Ingress Controllers and when isolating traffic to a specific Ingress Controller. For example, company A goes to one Ingress Controller and company B to another.

Procedure

1. Edit the **router-internal.yaml** file:

```
# cat router-internal.yaml
apiVersion: v1
items:
- apiVersion: operator.openshift.io/v1
  kind: IngressController
  metadata:
    name: sharded
    namespace: openshift-ingress-operator
  spec:
    domain: <apps-sharded.basedomain.example.net>
    nodePlacement:
      nodeSelector:
        matchLabels:
          node-role.kubernetes.io/worker: ""
    namespaceSelector:
      matchLabels:
        type: sharded
  status: {}
```

```
kind: List
metadata:
  resourceVersion: ""
  selfLink: ""
```

2. Apply the Ingress Controller **router-internal.yaml** file:

   ```
   # oc apply -f router-internal.yaml
   ```

   The Ingress Controller selects routes in any namespace that is selected by the namespace selector that have the label **type: sharded**.

CHAPTER 6. MANAGING MULTIPLE NETWORKS

6.1. OVERVIEW

Multus CNI provides the capability to attach multiple network interfaces to Pods in OpenShift Container Platform. This gives you flexibility when you must configure Pods that deliver network functionality, such as switching or routing.

Multus CNI is useful in situations where network isolation is needed, including data plane and control plane separation. Isolating network traffic is useful for the following performance and security reasons:

Performance

You can send traffic along two different planes in order to manage how much traffic is along 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.

All of the Pods in the cluster will still use the cluster-wide default network to maintain connectivity across the cluster. Every Pod has an **eth0** interface which is attached to the cluster-wide Pod network. You can view the interfaces for a Pod using the **oc exec -it <pod_name> -- ip a** command. If you add additional network interfaces using Multus CNI, they will be named **net1**, **net2**, ..., **netN**.

To attach additional network interfaces to a Pod, you must create configurations which define how the interfaces are attached. Each interface is specified using a Custom Resource (CR) that has a **NetworkAttachmentDefinition** type. A CNI configuration inside each of these CRs defines how that interface will be created. Multus CNI is a CNI plug-in that can call other CNI plug-ins. This allows the use of other CNI plug-ins to create additional network interfaces. For high performance networking, use the SR-IOV Device Plugin with Multus CNI.

Execute the following steps to attach additional network interfaces to Pods:

1. Create a CNI configuration as a custom resource.
2. Annotate the Pod with the configuration name.
3. Verify that the attachment was successful by viewing the status annotation.

6.2. CNI CONFIGURATIONS

CNI configurations are JSON data with only a single required field, **type**. The configuration in the **additional** field is free-form JSON data, which allows CNI plug-ins to make the configurations in the form that they require. Different CNI plug-ins use different configurations. See the documentation specific to the CNI plug-in that you want to use.

An example CNI configuration:

```
{
  "cniVersion": "0.3.0", ❶
  "type": "loopback", ❷
  "additional": "<plugin-specific-json-data>" ❸
}
```

1 **cniVersion**: Specifies the CNI version that is used. The CNI plug-in uses this information to check whether it is using a valid version.

2 **type**: Specifies which CNI plug-in binary to call on disk. In this example, the loopback binary is specified, Therefore, it creates a loopback-type network interface.

3 **additional**: The **<information>** value provided in the code above is an example. Each CNI plug-in specifies the configuration parameters it needs in JSON. These are specific to the CNI plug-in binary that is named in the **type** field.

6.3. CREATING ADDITIONAL NETWORK INTERFACES

Additional interfaces for Pods are defined in CNI configurations that are stored as Custom Resources (CRs). These CRs can be created, listed, edited, and deleted using the **oc** tool.

The following procedure configures a **macvlan** interface on a Pod. This configuration might not apply to all production environments, but you can use the same procedure for other CNI plug-ins.

6.3.1. Creating a CNI configuration for an additional interface as a CR

NOTE

If you want to attach an additional interface to a Pod, the CR that defines the interface must be in the same project (namespace) as the Pod.

1. Create a project to store CNI configurations as CRs and the Pods that will use the CRs.

```
$ oc new-project multinetwork-example
```

2. Create the CR that will define an additional network interface. Create a YAML file called **macvlan-conf.yaml** with the following contents:

```
apiVersion: "k8s.cni.cncf.io/v1"
kind: NetworkAttachmentDefinition 1
metadata:
  name: macvlan-conf 2
spec:
  config: '{ 3
      "cniVersion": "0.3.0",
      "type": "macvlan",
      "master": "eth0",
      "mode": "bridge",
      "ipam": {
        "type": "host-local",
        "subnet": "192.168.1.0/24",
        "rangeStart": "192.168.1.200",
        "rangeEnd": "192.168.1.216",
        "routes": [
          { "dst": "0.0.0.0/0" }
        ],
      "gateway": "192.168.1.1"
    }
  }'
```

1. **kind: NetworkAttachmentDefinition**. This is the name for the CR where this configuration will be stored. It is a custom extension of Kubernetes that defines how networks are attached to Pods.
2. **name** maps to the annotation, which is used in the next step.
3. **config**: The CNI configuration is packaged in the **config** field.

The configuration is specific to a plug-in, which enables **macvlan**. Note the **type** line in the CNI configuration portion. Aside from the IPAM (IP address management) parameters for networking, in this example the **master** field must reference a network interface that resides on the node(s) hosting the Pod(s).

3. Run the following command to create the CR:

```
$ oc create -f macvlan-conf.yaml
```

NOTE

This example is based on a **macvlan** CNI plug-in. Note that in AWS environments, macvlan traffic might be filtered and, therefore, might not reach the desired destination.

6.3.2. Managing the CRs for additional interfaces

You can manage the CRs for additional interfaces using the **oc** CLI.

Use the following command to list the CRs for additional interfaces:

```
$ oc get network-attachment-definitions.k8s.cni.cncf.io
```

Use the following command to delete CRs for additional interfaces:

```
$ oc delete network-attachment-definitions.k8s.cni.cncf.io macvlan-conf
```

6.3.3. Creating an annotated Pod that uses the CR

To create a Pod that uses the additional interface, use an annotation that refers to the CR. Create a YAML file called **samplepod.yaml** for a Pod with the following contents:

```
apiVersion: v1
kind: Pod
metadata:
  name: samplepod
  annotations:
    k8s.v1.cni.cncf.io/networks: macvlan-conf (1)
spec:
  containers:
```

```
  - name: samplepod
    command: ["/bin/bash", "-c", "sleep 2000000000000"]
    image: centos/tools
```

1 The **annotations** field contains **k8s.v1.cni.cncf.io/networks: macvlan-conf**, which correlates to the **name** field in the CR defined earlier.

Run the following command to create the **samplepod** Pod:

```
$ oc create -f samplepod.yaml
```

To verify that an additional network interface has been created and attached to the Pod, use the following command to list the IPv4 address information:

```
$ oc exec -it samplepod -- ip -4 addr
```

Three interfaces are listed in the output:

```
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN qlen 1000 1
    inet 127.0.0.1/8 scope host lo
       valid_lft forever preferred_lft forever
3: eth0@if6: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1450 qdisc noqueue state UP  link-netnsid 0 2
    inet 10.244.1.4/24 scope global eth0
       valid_lft forever preferred_lft forever
4: net1@if2: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue state UNKNOWN  link-netnsid 0 3
    inet 192.168.1.203/24 scope global net1
       valid_lft forever preferred_lft forever
```

1 **lo**: A loopback interface.

2 **eth0**: The interface that connects to the cluster-wide default network.

3 **net1**: The new interface that you just created.

6.3.3.1. Attaching multiple interfaces to a Pod

To attach more than one additional interface to a Pod, specify multiple names, in comma-delimited format, in the **annotations** field in the Pod definition.

The following **annotations** field in a Pod definition specifies different CRs for the additional interfaces:

```
  annotations:
    k8s.v1.cni.cncf.io/networks: macvlan-conf, tertiary-conf, quaternary-conf
```

The following **annotations** field in a Pod definition specifies the same CR for the additional interfaces:

```
  annotations:
    k8s.v1.cni.cncf.io/networks: macvlan-conf, macvlan-conf
```

6.3.4. Viewing the interface configuration in a running Pod

After the Pod is running, you can review the configurations of the additional interfaces created. To view the sample Pod from the earlier example, execute the following command.

```
$ oc describe pod samplepod
```

The **metadata** section of the output contains a list of annotations, which are displayed in JSON format:

```
Annotations:
  k8s.v1.cni.cncf.io/networks: macvlan-conf
  k8s.v1.cni.cncf.io/networks-status:
    [{
        "name": "openshift-sdn",
        "ips": [
            "10.131.0.10"
        ],
        "default": true,
        "dns": {}
    },{
        "name": "macvlan-conf", 1
        "interface": "net1", 2
        "ips": [ 3
            "192.168.1.200"
        ],
        "mac": "72:00:53:b4:48:c4", 4
        "dns": {} 5
    }]
```

1. **name** refers to the custom resource name, **macvlan-conf**.
2. **interface** refers to the name of the interface in the Pod.
3. **ips** is a list of IP addresses as assigned to the Pod.
4. **mac** is the MAC address of the interface.
5. **dns** refers DNS for the interface.

The first annotation, **k8s.v1.cni.cncf.io/networks: macvlan-conf**, refers to the CR created in the example. This annotation was specified in the Pod definition.

The second annotation is **k8s.v1.cni.cncf.io/networks-status**. There are two interfaces listed under **k8s.v1.cni.cncf.io/networks-status**.

- The first interface describes the interface for the default network, **openshift-sdn**. This interface is created as **eth0**. It is used for communications within the cluster.

- The second interface is the additional interface that you created, **net1**. The output above lists some key values that were configured when the interface was created, for example, the IP addresses that were assigned to the Pod.

6.4. CONFIGURING ADDITIONAL INTERFACES USING HOST DEVICES

The host-device plug-in connects an existing network device on a node directly to a Pod.

The code below creates a dummy device using a dummy module to back a virtual device, and assigns the dummy device **name** to **exampledevice0**.

```
$ modprobe dummy
$ lsmod | grep dummy
$ ip link add exampledevice0 type dummy
```

Procedure

1. To connect the dummy network device to a Pod, label the host, so that you can assign a Pod to the node where the device exists.

```
$ oc label nodes <your-worker-node-name> exampledevice=true
$ oc get nodes --show-labels
```

2. Create a YAML file called **hostdevice-example.yaml** for a custom resource to refer to this configuration:

```
apiVersion: "k8s.cni.cncf.io/v1"
kind: NetworkAttachmentDefinition
metadata:
  name: hostdevice-example
spec:
  config: '{
      "cniVersion": "0.3.0",
      "type": "host-device",
      "device": "exampledevice0"
    }'
```

3. Run the following command to create the **hostdevice-example** CR:

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

4. Create a YAML file for a Pod which refers to this name in the annotation. Include **nodeSelector** to assign the Pod to the machine where you created the alias.

```
apiVersion: v1
kind: Pod
metadata:
  name: hostdevicesamplepod
  annotations:
    k8s.v1.cni.cncf.io/networks: hostdevice-example
spec:
  containers:
  - name: hostdevicesamplepod
    command: ["/bin/bash", "-c", "sleep 2000000000000"]
    image: centos/tools
  nodeSelector:
    exampledevice: "true"
```

5. Run the following command to create the **hostdevicesamplepod** Pod:

```
$ oc create -f hostdevicesamplepod.yaml
```

6. View the additional interface that you created:

```
$ oc exec hostdevicesamplepod -- ip a
```

IMPORTANT

SR-IOV multinetwork support is a Technology Preview feature only. Technology Preview features are not supported with Red Hat production service level agreements (SLAs) and might not be functionally complete. Red Hat does not recommend using them in production. These features provide early access to upcoming product features, enabling customers to test functionality and provide feedback during the development process.

For more information about the support scope of Red Hat Technology Preview features, see https://access.redhat.com/support/offerings/techpreview/.

6.5. CONFIGURING SR-IOV

OpenShift Container Platform includes the capability to use SR-IOV hardware on OpenShift Container Platform nodes, which enables you to attach SR-IOV virtual function (VF) interfaces to Pods in addition to other network interfaces.

Two components are required to provide this capability: the SR-IOV network device plug-in and the SR-IOV CNI plug-in.

- The SR-IOV network device plug-in is a Kubernetes device plug-in for discovering, advertising, and allocating SR-IOV network virtual function (VF) resources. Device plug-ins are used in Kubernetes to enable the use of limited resources, typically in physical devices. Device plug-ins give the Kubernetes scheduler awareness of which resources are exhausted, allowing Pods to be scheduled to worker nodes that have sufficient resources available.

- The SR-IOV CNI plug-in plumbs VF interfaces allocated from the SR-IOV device plug-in directly into a Pod.

6.5.1. Supported Devices

The following Network Interface Card (NIC) models are supported in OpenShift Container Platform:

- Intel XXV710-DA2 25G card with vendor ID 0x8086 and device ID 0x158b

- Mellanox MT27710 Family [ConnectX-4 Lx] 25G card with vendor ID 0x15b3 and device ID 0x1015

- Mellanox MT27800 Family [ConnectX-5] 100G card with vendor ID 0x15b3 and device ID 0x1017

NOTE

For Mellanox cards, ensure that SR-IOV is enabled in the firmware before provisioning VFs on the host.

6.5.2. Creating SR-IOV plug-ins and daemonsets

NOTE

The creation of SR-IOV VFs is not handled by the SR-IOV device plug-in and SR-IOV CNI. To provision SR-IOV VF on hosts, you must configure it manually.

To use the SR-IOV network device plug-in and SR-IOV CNI plug-in, run both plug-ins in daemon mode on each node in your cluster.

1. Create a YAML file for the **openshift-sriov** namespace with the following contents:

```
apiVersion: v1
kind: Namespace
metadata:
  name: openshift-sriov
  labels:
    name: openshift-sriov
    openshift.io/run-level: "0"
  annotations:
    openshift.io/node-selector: ""
    openshift.io/description: "Openshift SR-IOV network components"
```

2. Run the following command to create the **openshift-sriov** namespace:

```
$ oc create -f openshift-sriov.yaml
```

3. Create a YAML file for the **sriov-device-plugin** service account with the following contents:

```
apiVersion: v1
kind: ServiceAccount
metadata:
  name: sriov-device-plugin
  namespace: openshift-sriov
```

4. Run the following command to create the **sriov-device-plugin** service account:

```
$ oc create -f sriov-device-plugin.yaml
```

5. Create a YAML file for the **sriov-cni** service account with the following contents:

```
apiVersion: v1
kind: ServiceAccount
metadata:
  name: sriov-cni
  namespace: openshift-sriov
```

6. Run the following command to create the **sriov-cni** service account:

```
$ oc create -f sriov-cni.yaml
```

7. Create a YAML file for the **sriov-device-plugin** DaemonSet with the following contents:

NOTE

The SR-IOV network device plug-in daemon, when launched, will discover all the configured SR-IOV VFs (of supported NIC models) on each node and advertise discovered resources. The number of available SR-IOV VF resources that are capable of being allocated can be reviewed by describing a node with the **oc describe node <node-name>** command. The resource name for the SR-IOV VF resources is **openshift.io/sriov**. When no SR-IOV VFs are available on the node, a value of zero is displayed.

```
kind: DaemonSet
apiVersion: apps/v1
metadata:
  name: sriov-device-plugin
  namespace: openshift-sriov
  annotations:
    kubernetes.io/description: |
      This daemon set launches the SR-IOV network device plugin on each node.
spec:
  selector:
    matchLabels:
      app: sriov-device-plugin
  updateStrategy:
    type: RollingUpdate
  template:
    metadata:
      labels:
        app: sriov-device-plugin
        component: network
        type: infra
        openshift.io/component: network
    spec:
      hostNetwork: true
      nodeSelector:
        beta.kubernetes.io/os: linux
      tolerations:
      - operator: Exists
      serviceAccountName: sriov-device-plugin
      containers:
      - name: sriov-device-plugin
        image: quay.io/openshift/ose-sriov-network-device-plugin:v4.0.0
        args:
        - --log-level=10
        securityContext:
          privileged: true
        volumeMounts:
        - name: devicesock
          mountPath: /var/lib/kubelet/
          readOnly: false
        - name: net
          mountPath: /sys/class/net
          readOnly: true
      volumes:
        - name: devicesock
          hostPath:
            path: /var/lib/kubelet/
        - name: net
          hostPath:
            path: /sys/class/net
```

8. Run the following command to create the **sriov-device-plugin** DaemonSet:

```
oc create -f sriov-device-plugin.yaml
```

9. Create a YAML file for the **sriov-cni** DaemonSet with the following contents:

```
kind: DaemonSet
apiVersion: apps/v1
metadata:
  name: sriov-cni
  namespace: openshift-sriov
  annotations:
    kubernetes.io/description: |
      This daemon set launches the SR-IOV CNI plugin on SR-IOV capable worker nodes.
spec:
  selector:
    matchLabels:
      app: sriov-cni
  updateStrategy:
    type: RollingUpdate
  template:
    metadata:
      labels:
        app: sriov-cni
        component: network
        type: infra
        openshift.io/component: network
    spec:
      nodeSelector:
        beta.kubernetes.io/os: linux
      tolerations:
      - operator: Exists
      serviceAccountName: sriov-cni
      containers:
      - name: sriov-cni
        image: quay.io/openshift/ose-sriov-cni:v4.0.0
        securityContext:
          privileged: true
        volumeMounts:
        - name: cnibin
          mountPath: /host/opt/cni/bin
      volumes:
        - name: cnibin
          hostPath:
            path: /var/lib/cni/bin
```

10. Run the following command to create the **sriov-cni** DaemonSet:

```
$ oc create -f sriov-cni.yaml
```

6.5.3. Configuring additional interfaces using SR-IOV

1. Create a YAML file for the Custom Resource (CR) with SR-IOV configuration. The **name** field in the following CR has the value **sriov-conf**.

```
apiVersion: "k8s.cni.cncf.io/v1"
kind: NetworkAttachmentDefinition
metadata:
  name: sriov-conf
  annotations:
    k8s.v1.cni.cncf.io/resourceName: openshift.io/sriov 1
spec:
  config: '{
      "type": "sriov", 2
      "name": "sriov-conf",
      "ipam": {
        "type": "host-local",
        "subnet": "10.56.217.0/24",
        "routes": [{
          "dst": "0.0.0.0/0"
        }],
        "gateway": "10.56.217.1"
      }
    }'
```

1 **k8s.v1.cni.cncf.io/resourceName** annotation is set to **openshift.io/sriov**.

2 **type** is set to **sriov**.

2. Run the following command to create the **sriov-conf** CR:

```
$ oc create -f sriov-conf.yaml
```

3. Create a YAML file for a Pod which references the name of the **NetworkAttachmentDefinition** and requests one **openshift.io/sriov** resource:

```
apiVersion: v1
kind: Pod
metadata:
  name: sriovsamplepod
  annotations:
    k8s.v1.cni.cncf.io/networks: sriov-conf
spec:
  containers:
  - name: sriovsamplepod
    command: ["/bin/bash", "-c", "sleep 2000000000000"]
    image: centos/tools
    resources:
      requests:
        openshift.io/sriov: '1'
      limits:
        openshift.io/sriov: '1'
```

4. Run the following command to create the **sriovsamplepod** Pod:

```
$ oc create -f sriovsamplepod.yaml
```

5. View the additional interface by executing the **ip** command:

```
$ oc exec sriovsamplepod -- ip a
```

CHAPTER 7. CONFIGURING NETWORK POLICY WITH OPENSHIFT SDN

7.1. 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 Container Platform 4.1, OpenShift SDN supports using NetworkPolicy in its default network isolation mode.

NOTE

The Kubernetes **v1** NetworkPolicy features are available in OpenShift Container Platform 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 Container Platform Ingress Controller:
  To make a project allow only connections from the OpenShift Container Platform 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.

7.2. 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.

7.3. 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
```

7.4. 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.

7.5. 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
```

7.6. 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 Container Platform 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
```

7.7. CREATING DEFAULT NETWORK POLICIES FOR A NEW PROJECT

As a cluster administrator, you can modify the new project template to automatically include NetworkPolicy objects when you create a new project.

7.7.1. Modifying the template for new projects

As a cluster administrator, you can modify the default project template so that new projects are created using your custom requirements.

To create your own custom project template:

Procedure

1. Log in as a user with **cluster-admin** privileges.

2. Generate the default project template:

```
$ oc adm create-bootstrap-project-template -o yaml > template.yaml
```

3. Use a text editor to modify the generated **template.yaml** file by adding objects or modifying existing objects.

4. The project template must be created in the **openshift-config** namespace. Load your modified template:

   ```
   $ oc create -f template.yaml -n openshift-config
   ```

5. Edit the project configuration resource using the web console or CLI.

   - Using the web console:

     i. Navigate to the **Administration** → **Cluster Settings** page.

     ii. Click **Global Configuration** to view all configuration resources.

     iii. Find the entry for **Project** and click **Edit YAML**.

   - Using the CLI:

     i. Edit the **project.config.openshift.io/cluster** resource:

     ```
     $ oc edit project.config.openshift.io/cluster
     ```

6. Update the **spec** section to include the **projectRequestTemplate** and **name** parameters, and set the name of your uploaded project template. The default name is **project-request**.

   Project configuration resource with custom project template

   ```
   apiVersion: config.openshift.io/v1
   kind: Project
   metadata:
     ...
   spec:
     projectRequestTemplate:
       name: <template_name>
   ```

7. After you save your changes, create a new project to verify that your changes were successfully applied.

7.7.2. Adding network policy objects to the new project template

As a cluster administrator, you can add network policy objects to the default template for new projects. OpenShift Container Platform will automatically create all the NetworkPolicy CRs specified in the template in the project.

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 with a user with **cluster-admin** privileges.
- You must have created a custom default project template for new projects.

Procedure

1. Edit the default template for a new project by running the following command:

   ```
   $ oc edit template <project_template> -n openshift-config
   ```

   Replace **<project_template>** with the name of the default template that you configured for your cluster. The default template name is **project-request**.

2. In the template, add each NetworkPolicy object as an element to the **objects** parameter. The **objects** parameter accepts a collection of one or more objects.
   In the following example, the **objects** parameter collection includes several NetworkPolicy objects:

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

3. Optional: Create a new project to confirm that your network policy objects are created successfully by running the following commands:

   a. Create a new project:

      ```
      $ oc new-project <project> 1
      ```

      1 Replace **<project>** with the name for the project you are creating.

   b. Confirm that the network policy objects in the new project template exist in the new project:

      ```
      $ oc get networkpolicy
      NAME                           POD-SELECTOR   AGE
      allow-from-openshift-ingress   <none>         7s
      allow-from-same-namespace      <none>         7s
      ```

CHAPTER 8. OPENSHIFT SDN

8.1. ABOUT OPENSHIFT SDN

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

OpenShift SDN provides three SDN modes for configuring the Pod network:

- The *network policy* mode allows project administrators to configure their own isolation policies using NetworkPolicy objects. NetworkPolicy is the default mode in OpenShift Container Platform 4.1.
- The *multitenant* mode provides project-level isolation for Pods and Services. Pods from different projects cannot send packets to or receive packets from Pods and Services of a different project. You can disable isolation for a project, allowing it to send network traffic to all Pods and Services in the entire cluster and receive network traffic from those Pods and Services.
- The *subnet* mode provides a flat Pod network where every Pod can communicate with every other Pod and Service. The network policy mode provides the same functionality as the subnet mode.

8.2. CONFIGURING EGRESS IPS FOR A PROJECT

As a cluster administrator, you can configure the OpenShift SDN network provider to assign one or more egress IP addresses to a project.

8.2.1. Egress IP address assignment for project egress traffic

By configuring an egress IP address for a project, all outgoing external connections from the specified project will share the same, fixed source IP address. External resources can recognize traffic from a particular project based on the egress IP address. An egress IP address assigned to a project is different from the egress router, which is used to send traffic to specific destinations.

Egress IP addresses are implemented as additional IP addresses on the primary network interface of the node and must be in the same subnet as the node’s primary IP address.

IMPORTANT

Egress IP addresses must not be configured in any Linux network configuration files, such as **ifcfg-eth0**.

Allowing additional IP addresses on the primary network interface might require extra configuration when using some cloud or VM solutions.

You can assign egress IP addresses to namespaces by setting the **egressIPs** parameter of the **NetNamespace** resource. After an egress IP is associated with a project, OpenShift SDN allows you to assign egress IPs to hosts in two ways:

- In the *automatically assigned* approach, an egress IP address range is assigned to a node.

- In the *manually assigned* approach, a list of one or more egress IP address is assigned to a node.

Namespaces that request an egress IP address are matched with nodes that can host those egress IP addresses, and then the egress IP addresses are assigned to those nodes. If the **egressIPs** parameter is set on a **NetNamespace** resource, but no node hosts that egress IP address, then egress traffic from the namespace will be dropped.

High availability of nodes is automatic. If a node that hosts an egress IP address is unreachable and there are nodes that are able to host that egress IP address, then the egress IP address will move to a new node. When the unreachable node comes back online, the egress IP address automatically moves to balance egress IP addresses across nodes.

IMPORTANT

You cannot use manually assigned and automatically assigned egress IP addresses on the same nodes. If you manually assign egress IP addresses from an IP address range, you must not make that range available for automatic IP assignment.

8.2.1.1. Considerations when using automatically assigned egress IP addresses

When using the automatic assignment approach for egress IP addresses the following considerations apply:

- You set the **egressCIDRs** parameter of each node's **HostSubnet** resource to indicate the range of egress IP addresses that can be hosted by a node. OpenShift Container Platform sets the **egressIPs** parameter of the **HostSubnet** resource based on the IP address range you specify.

- Only a single egress IP address per namespace is supported when using the automatic assignment mode.

If the node hosting the namespace's egress IP address is unreachable, OpenShift Container Platform will reassign the egress IP address to another node with a compatible egress IP address range. The automatic assignment approach works best for clusters installed in environments with flexibility in associating additional IP addresses with nodes.

8.2.1.2. Considerations when using manually assigned egress IP addresses

When using the manual assignment approach for egress IP addresses the following considerations apply:

- You set the **egressIPs** parameter of each node's **HostSubnet** resource to indicate the IP addresses that can be hosted by a node.

- Multiple egress IP addresses per namespace are supported.

When a namespace has multiple egress IP addresses, if the node hosting the first egress IP address is unreachable, OpenShift Container Platform will automatically switch to using the next available egress IP address until the first egress IP address is reachable again.

This approach is recommended for clusters installed in public cloud environments, where there can be limitations on associating additional IP addresses with nodes.

8.2.2. Configuring automatically assigned egress IP addresses for a namespace

In OpenShift Container Platform you can enable automatic assignment of an egress IP address for a specific namespace across one or more nodes.

Prerequisites

- Install the OpenShift Command-line Interface (CLI), commonly known as **oc**.
- Access to the cluster as a user with the **cluster-admin** role.

Procedure

1. Update the **NetNamespace** resource with the egress IP address using the following JSON:

```
$ oc patch netnamespace <project_name> --type=merge -p \ ❶
  '{
    "egressIPs": [
      "<ip_address>" ❷
    ]
  }'
```

❶ Specify the name of the project.

❷ Specify a single egress IP address. Using multiple IP addresses is not supported.

For example, to assign **project1** to an IP address of 192.168.1.100 and **project2** to an IP address of 192.168.1.101:

```
$ oc patch netnamespace project1 --type=merge -p \
  '{"egressIPs": ["192.168.1.100"]}'
$ oc patch netnamespace project2 --type=merge -p \
  '{"egressIPs": ["192.168.1.101"]}'
```

2. Indicate which nodes can host egress IP addresses by setting the **egressCIDRs** parameter for each host using the following JSON:

```
$ oc patch hostsubnet <node_name> --type=merge -p \ ❶
  '{
    "egressCIDRs": [
      "<ip_address_range_1>", "<ip_address_range_2>" ❷
    ]
  }'
```

❶ Specify a node name.

❷ Specify one or more IP address ranges in CIDR format.

For example, to set **node1** and **node2** to host egress IP addresses in the range 192.168.1.0 to 192.168.1.255:

```
$ oc patch hostsubnet node1 --type=merge -p \
  '{"egressCIDRs": ["192.168.1.0/24"]}'
$ oc patch hostsubnet node2 --type=merge -p \
  '{"egressCIDRs": ["192.168.1.0/24"]}'
```

OpenShift Container Platform automatically assigns specific egress IP addresses to available nodes in a balanced way. In this case, it assigns the egress IP address 192.168.1.100 to **node1** and the egress IP address 192.168.1.101 to **node2** or vice versa.

8.2.3. Configuring manually assigned egress IP addresses for a namespace

In OpenShift Container Platform you can associate one or more egress IP addresses with a namespace.

Prerequisites

- Install the OpenShift Command-line Interface (CLI), commonly known as **oc**.
- Access to the cluster as a user with the **cluster-admin** role.

Procedure

1. Update the **NetNamespace** resource by specifying the following JSON object with the desired IP addresses:

```
$ oc patch netnamespace <project> --type=merge -p \ 1
  '{
    "egressIPs": [ 2
      "<ip_address>"
      ]
  }'
```

1 Specify the name of the project.

2 Specify one or more egress IP addresses. The **egressIPs** parameter is an array.

For example, to assign the **project1** project to an IP address of **192.168.1.100**:

```
$ oc patch netnamespace project1 --type=merge \
  -p '{"egressIPs": ["192.168.1.100"]}'
```

You can set **egressIPs** to two or more IP addresses on different nodes to provide high availability. If multiple egress IP addresses are set, pods use the first IP in the list for egress, but if the node hosting that IP address fails, pods switch to using the next IP in the list after a short delay.

2. Manually assign the egress IP to the node hosts. Set the **egressIPs** parameter on the **HostSubnet** object on the node host. Using the following JSON, include as many IPs as you want to assign to that node host:

```
$ oc patch hostsubnet <node_name> --type=merge -p \ 1
  '{
    "egressIPs": [ 2
      "<ip_address_1>",
      "<ip_address_N>"
      ]
  }'
```

1 Specify the name of the project.

2 Specify one or more egress IP addresses. The **egressIPs** field is an array.

For example, to specify that **node1** should have the egress IPs **192.168.1.100**, **192.168.1.101**, and **192.168.1.102**:

```
$ oc patch hostsubnet node1 --type=merge -p \
  '{"egressIPs": ["192.168.1.100", "192.168.1.101", "192.168.1.102"]}'
```

In the previous example, all egress traffic for **project1** will be routed to the node hosting the specified egress IP, and then connected (using NAT) to that IP address.

8.3. 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 Container Platform cluster.

8.3.1. How an egress firewall works in a project

As an OpenShift Container Platform 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 Container Platform cluster.
- A Pod cannot reach specified internal subnets or hosts outside the OpenShift Container Platform 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.

8.3.1.1. 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.

8.3.1.2. 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.

8.3.1.3. 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.

8.3.2. 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.

8.3.2.1. 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.

8.3.2.2. 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.

8.3.3. Creating an egress firewall policy object

As a OpenShift Container Platform 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.

8.4. EDITING AN EGRESS FIREWALL FOR A PROJECT

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

8.4.1. Editing an EgressNetworkPolicy object

As a OpenShift Container Platform 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. Optionally, 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
```

8.4.2. EgressNetworkPolicy custom resource (CR) object

The following YAML describes an EgressNetworkPolicy CR object:

```
kind: EgressNetworkPolicy
apiVersion: v1
```

```
metadata:
  name: <name> ❶
spec:
  egress: ❷
    ...
```

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

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

8.4.2.1. 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> ❶
  to: ❷
    cidrSelector: <cidr> ❸
    dnsName: <dns-name> ❹
```

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

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

❸ Specify an IP address range in CIDR format.

❹ Specify a domain name.

8.4.2.2. Example EgressNetworkPolicy CR object

The following example defines several egress firewall policy rules:

```
kind: EgressNetworkPolicy
apiVersion: v1
metadata:
  name: default-rules ❶
spec:
  egress: ❷
  - 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
```

❶ The name for the policy object.

2 A collection of egress firewall policy rule objects.

8.5. 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 Container Platform cluster.

8.5.1. Removing an EgressNetworkPolicy object

As a OpenShift Container Platform 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>
   ```

8.6. USING MULTICAST

8.6.1. About multicast

With IP multicast, data is broadcast to many IP addresses simultaneously.

IMPORTANT

At this time, multicast is best used for low-bandwidth coordination or service discovery and not a high-bandwidth solution.

Multicast traffic between OpenShift Container Platform Pods is disabled by default. If you are using the OpenShift SDN network plug-in, you can enable multicast on a per-project basis.

When using the OpenShift SDN network plug-in in **networkpolicy** isolation mode:

- Multicast packets sent by a Pod will be delivered to all other Pods in the project, regardless of NetworkPolicy objects. Pods might be able to communicate over multicast even when they cannot communicate over unicast.
- Multicast packets sent by a Pod in one project will never be delivered to Pods in any other project, even if there are NetworkPolicy objects that allow communication between the projects.

When using the OpenShift SDN network plug-in in **multitenant** isolation mode:

- Multicast packets sent by a Pod will be delivered to all other Pods in the project.
- Multicast packets sent by a Pod in one project will be delivered to Pods in other projects only if each project is joined together and multicast is enabled in each joined project.

8.6.2. Enabling multicast between Pods

You can enable multicast between Pods for your project.

Prerequisites

- Install the OpenShift Command-line Interface (CLI), commonly known as **oc**.
- You must log in to the cluster with a user that has the **cluster-admin** role.

Procedure

- Run the following command to enable multicast for a project:

```
$ oc annotate netnamespace <namespace> \ 1
    netnamespace.network.openshift.io/multicast-enabled=true
```

1 The **namespace** for the project you want to enable multicast for.

8.6.3. Disabling multicast between Pods

You can disable multicast between Pods for your project.

Prerequisites

- Install the OpenShift Command-line Interface (CLI), commonly known as **oc**.
- You must log in to the cluster with a user that has the **cluster-admin** role.

Procedure

- Disable multicast by running the following command:

```
$ oc annotate netnamespace <namespace> \ 1
    netnamespace.network.openshift.io/multicast-enabled-
```

1 The **namespace** for the project you want to disable multicast for.

8.7. CONFIGURING NETWORK ISOLATION USING OPENSHIFT SDN

When your cluster is configured to use the multitenant isolation mode for the OpenShift SDN CNI plug-in, each project is isolated by default. Network traffic is not allowed between Pods or services in different projects in multitenant isolation mode.

You can change the behavior of multitenant isolation for a project in two ways:

- You can join one or more projects, allowing network traffic between Pods and services in different projects.
- You can disable network isolation for a project. It will be globally accessible, accepting network traffic from Pods and services in all other projects. A globally accessible project can access Pods and services in all other projects.

Prerequisites

- You must have a cluster configured to use the OpenShift SDN Container Network Interface (CNI) plug-in in multitenant isolation mode.

8.7.1. Joining projects

You can join two or more projects to allow network traffic between Pods and services in different projects.

Prerequisites

- Install the OpenShift Command-line Interface (CLI), commonly known as **oc**.
- You must log in to the cluster with a user that has the **cluster-admin** role.

Procedure

1. Use the following command to join projects to an existing project network:

   ```
   $ oc adm pod-network join-projects --to=<project1> <project2> <project3>
   ```

   Alternatively, instead of specifying specific project names, you can use the **--selector=<project_selector>** option to specify projects based upon an associated label.

2. Optional: Run the following command to view the pod networks that you have joined together:

   ```
   $ oc get netnamespaces
   ```

   Projects in the same pod-network have the same network ID in the **NETID** column.

8.7.2. Isolating a project

You can isolate a project so that Pods and services in other projects cannot access its Pods and services.

Prerequisites

- Install the OpenShift Command-line Interface (CLI), commonly known as **oc**.

- You must log in to the cluster with a user that has the **cluster-admin** role.

Procedure

- To isolate the projects in the cluster, run the following command:

  ```
  $ oc adm pod-network isolate-projects <project1> <project2>
  ```

  Alternatively, instead of specifying specific project names, you can use the **--selector=<project_selector>** option to specify projects based upon an associated label.

8.7.3. Disabling network isolation for a project

You can disable network isolation for a project.

Prerequisites

- Install the OpenShift Command-line Interface (CLI), commonly known as **oc**.
- You must log in to the cluster with a user that has the **cluster-admin** role.

Procedure

- Run the following command for the project:

  ```
  $ oc adm pod-network make-projects-global <project1> <project2>
  ```

  Alternatively, instead of specifying specific project names, you can use the **--selector=<project_selector>** option to specify projects based upon an associated label.

8.8. CONFIGURING KUBE-PROXY

The Kubernetes network proxy (kube-proxy) runs on each node and is managed by the Cluster Network Operator (CNO). kube-proxy maintains network rules for forwarding connections for endpoints associated with services.

8.8.1. About iptables rules synchronization

The synchronization period determines how frequently the Kubernetes network proxy (kube-proxy) syncs the iptables rules on a node.

A sync begins when either of the following events occurs:

- An event occurs, such as service or endpoint is added to or removed from the cluster.
- The time since the last sync exceeds the sync period defined for kube-proxy.

8.8.2. Modifying the kube-proxy configuration

You can modify the Kubernetes network proxy configuration for your cluster.

Prerequisites

- Install the OpenShift Command-line Interface (CLI), commonly known as **oc**.
- Log in to a running cluster with the **cluster-admin** role.

Procedure

1. Edit the **Network.operator.openshift.io** Custom Resource (CR) by running the following command:

   ```
   $ oc edit network.operator.openshift.io cluster
   ```

2. Modify the **kubeProxyConfig** parameter in the CR with your changes to the kube-proxy configuration, such as in the following example CR:

   ```
   apiVersion: operator.openshift.io/v1
   kind: Network
   metadata:
     name: cluster
   spec:
     kubeProxyConfig:
       iptablesSyncPeriod: 30s
       proxyArguments:
         iptables-min-sync-period: ["30s"]
   ```

3. Save the file and exit the text editor.
   The syntax is validated by the **oc** command when you save the file and exit the editor. If your modifications contain a syntax error, the editor opens the file and displays an error message.

4. Run the following command to confirm the configuration update:

   ```
   $ oc get networks.operator.openshift.io -o yaml
   ```

   The command returns output similar to the following example:

   ```
   apiVersion: v1
   items:
   - apiVersion: operator.openshift.io/v1
     kind: Network
     metadata:
       name: cluster
     spec:
       clusterNetwork:
       - cidr: 10.128.0.0/14
         hostPrefix: 23
       defaultNetwork:
         type: OpenShiftSDN
       kubeProxyConfig:
         iptablesSyncPeriod: 30s
         proxyArguments:
           iptables-min-sync-period:
           - 30s
       serviceNetwork:
       - 172.30.0.0/16
     status: {}
   kind: List
   ```

5. Optional: Run the following command to confirm that the Cluster Network Operator accepted the configuration change:

```
$ oc get clusteroperator network
NAME      VERSION     AVAILABLE   PROGRESSING   DEGRADED   SINCE
network   4.1.0-0.9   True        False         False      1m
```

The **AVAILABLE** field is **True** when the configuration update is applied successfully.

8.8.3. kube-proxy configuration parameters

You can modify the following **kubeProxyConfig** parameters:

Table 8.1. Parameters

Parameter	Description	Values	Default
iptablesSyncPeriod	The refresh period for **iptables** rules.	A time interval, such as **30s** or **2m**. Valid suffixes include **s**, **m**, and **h** and are described in the Go time package documentation.	**30s**
proxyArguments.iptables-min-sync-period	The minimum duration before refreshing **iptables** rules. This parameter ensures that the refresh does not happen too frequently.	A time interval, such as **30s** or **2m**. Valid suffixes include **s**, **m**, and **h** and are described in the Go time package	**30s**

CHAPTER 9. CONFIGURING ROUTES

9.1. ROUTE CONFIGURATION

9.1.1. 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
```

9.1.2. 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.

9.1.3. Troubleshooting throughput issues

Sometimes applications deployed through OpenShift Container Platform 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 Container Platform 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.
  - For information on installing and using iperf, see this Red Hat Solution .

9.1.4. Using cookies to keep route statefulness

OpenShift Container Platform 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 Container Platform 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.

9.1.4.1. 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
```

9.1.5. 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 9.1. 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.

9.2. SECURED ROUTES

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

9.2.1. 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.

9.2.2. 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 10. CONFIGURING INGRESS CLUSTER TRAFFIC

10.1. CONFIGURING INGRESS CLUSTER TRAFFIC OVERVIEW

OpenShift Container Platform provides the following methods for communicating from outside the cluster with services running in the cluster.

The methods are recommended, in order or preference:

- If you have HTTP/HTTPS, use an Ingress Controller.
- If you have a TLS-encrypted protocol other than HTTPS. For example, for TLS with the SNI header, use an Ingress Controller.
- Otherwise, use a Load Balancer, an External IP, or a **NodePort**.

Method	Purpose
Use an Ingress Controller	Allows access to HTTP/HTTPS traffic and TLS-encrypted protocols other than HTTPS (for example, TLS with the SNI header).
Automatically assign an external IP using a load balancer service	Allows traffic to non-standard ports through an IP address assigned from a pool.
Manually assign an external IP to a service	Allows traffic to non-standard ports through a specific IP address.
Configure a **NodePort**	Expose a service on all nodes in the cluster.

10.2. CONFIGURING INGRESS CLUSTER TRAFFIC USING AN INGRESS CONTROLLER

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

10.2.1. Using Ingress Controllers and routes

The Ingress Operator manages Ingress Controllers and wildcard DNS.

Using an Ingress Controller is the most common way to allow external access to an OpenShift Container Platform cluster.

An Ingress Controller is configured to accept external requests and proxy them based on the configured routes. This is limited to HTTP, HTTPS using SNI, and TLS using SNI, which is sufficient for web applications and services that work over TLS with SNI.

Work with your administrator to configure an Ingress Controller to accept external requests and proxy them based on the configured routes.

The administrator can create a wildcard DNS entry and then set up an Ingress Controller. Then, you can work with the edge Ingress Controller without having to contact the administrators.

When a set of routes is created in various projects, the overall set of routes is available to the set of Ingress Controllers. Each Ingress Controller admits routes from the set of routes. By default, all Ingress Controllers admit all routes.

The Ingress Controller:

- Has two replicas by default, which means it should be running on two worker nodes.
- Can be scaled up to have more replicas on more nodes.

> NOTE
>
> The procedures in this section require prerequisites performed by the cluster administrator.

Prerequisites

Before starting the following procedures, the administrator must:

- Set up the external port to the cluster networking environment so that requests can reach the cluster.
- Make sure there is at least one user with cluster admin role. To add this role to a user, run the following command:

  ```
  oc adm policy add-cluster-role-to-user cluster-admin username
  ```

- Have an OpenShift Container Platform cluster with at least one master and at least one node and a system outside the cluster that has network access to the cluster. This procedure assumes that the external system is on the same subnet as the cluster. The additional networking required for external systems on a different subnet is out-of-scope for this topic.

10.2.2. 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.

Procedure

1. Log in to OpenShift Container Platform.
2. Create a new project for your service:

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

   For example:

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

3. 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/openshift3/mysql-55-rhel7
```

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

```
$ oc get svc -n openshift-ingress
NAME                    TYPE         CLUSTER-IP     EXTERNAL-IP     PORT(S)
AGE
router-default          LoadBalancer   172.30.16.119   52.230.228.163
80:30745/TCP,443:32561/TCP   2d6h
router-internal-default   ClusterIP      172.30.101.15   <none>
80/TCP,443/TCP,1936/TCP      2d6h
```

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

10.2.3. Exposing the service by creating a route

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

Procedure

To expose the service:

1. Log in to OpenShift Container Platform.
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-55-rhel7
route "mysql-55-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)]>
```

10.2.4. Configuring ingress controller sharding by using route labels

Ingress Controller sharding by using route labels means that the Ingress Controller serves any route in any namespace that is selected by the route selector.

Ingress Controller sharding is useful when balancing incoming traffic load among a set of Ingress Controllers and when isolating traffic to a specific Ingress Controller. For example, company A goes to one Ingress Controller and company B to another.

Procedure

1. Edit the **router-internal.yaml** file:

```
# cat router-internal.yaml
apiVersion: v1
items:
- apiVersion: operator.openshift.io/v1
  kind: IngressController
  metadata:
    name: sharded
    namespace: openshift-ingress-operator
  spec:
    domain: <apps-sharded.basedomain.example.net>
    nodePlacement:
      nodeSelector:
        matchLabels:
          node-role.kubernetes.io/worker: ""
    routeSelector:
      matchLabels:
        type: sharded
  status: {}
kind: List
metadata:
  resourceVersion: ""
  selfLink: ""
```

2. Apply the Ingress Controller **router-internal.yaml** file:

```
# oc apply -f router-internal.yaml
```

The Ingress Controller selects routes in any namespace that have the label **type: sharded**.

10.2.5. Configuring ingress controller sharding by using namespace labels

Ingress Controller sharding by using namespace labels means that the Ingress Controller serves any route in any namespace that is selected by the namespace selector.

Ingress Controller sharding is useful when balancing incoming traffic load among a set of Ingress Controllers and when isolating traffic to a specific Ingress Controller. For example, company A goes to one Ingress Controller and company B to another.

Procedure

1. Edit the **router-internal.yaml** file:

```
# cat router-internal.yaml
apiVersion: v1
items:
- apiVersion: operator.openshift.io/v1
  kind: IngressController
  metadata:
    name: sharded
    namespace: openshift-ingress-operator
  spec:
    domain: <apps-sharded.basedomain.example.net>
    nodePlacement:
      nodeSelector:
        matchLabels:
          node-role.kubernetes.io/worker: ""
    namespaceSelector:
      matchLabels:
        type: sharded
  status: {}
kind: List
metadata:
  resourceVersion: ""
  selfLink: ""
```

2. Apply the Ingress Controller **router-internal.yaml** file:

```
# oc apply -f router-internal.yaml
```

The Ingress Controller selects routes in any namespace that is selected by the namespace selector that have the label **type: sharded**.

10.2.6. Additional resources

- The Ingress Operator manages wildcard DNS. For more information, see Ingress Operator in OpenShift Container Platform, Installing a cluster on bare metal , and Installing a cluster on vSphere.

10.3. CONFIGURING INGRESS CLUSTER TRAFFIC USING A LOAD BALANCER

OpenShift Container Platform provides methods for communicating from outside the cluster with services running in the cluster. This method uses a load balancer.

10.3.1. Using a load balancer to get traffic into the cluster

If you do not need a specific external IP address, you can configure a load balancer service to allow external access to an OpenShift Container Platform cluster.

A load balancer service allocates a unique IP. The load balancer has a single edge router IP, which can be a virtual IP (VIP), but is still a single machine for initial load balancing.

NOTE

If a pool is configured, it is done at the infrastructure level, not by a cluster administrator.

NOTE

The procedures in this section require prerequisites performed by the cluster administrator.

Prerequisites

Before starting the following procedures, the administrator must:

- Set up the external port to the cluster networking environment so that requests can reach the cluster.
- Make sure there is at least one user with cluster admin role. To add this role to a user, run the following command:

  ```
  oc adm policy add-cluster-role-to-user cluster-admin username
  ```

- Have an OpenShift Container Platform cluster with at least one master and at least one node and a system outside the cluster that has network access to the cluster. This procedure assumes that the external system is on the same subnet as the cluster. The additional networking required for external systems on a different subnet is out-of-scope for this topic.

10.3.2. 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.

Procedure

1. Log in to OpenShift Container Platform.
2. Create a new project for your service:

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

   For example:

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

3. 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/openshift3/mysql-55-rhel7
   ```

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

```
$ oc get svc -n openshift-ingress
NAME                    TYPE          CLUSTER-IP     EXTERNAL-IP      PORT(S)
AGE
router-default           LoadBalancer   172.30.16.119   52.230.228.163
80:30745/TCP,443:32561/TCP   2d6h
router-internal-default   ClusterIP      172.30.101.15   <none>
80/TCP,443/TCP,1936/TCP      2d6h
```

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

10.3.3. Exposing the service by creating a route

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

Procedure

To expose the service:

1. Log in to OpenShift Container Platform.

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-55-rhel7
route "mysql-55-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)]>
```

10.3.4. Creating a load balancer service

Use the following procedure to create a load balancer service.

Prerequisites

- Make sure that the project and service you want to expose exist.

Procedure

To create a load balancer service:

1. Log into OpenShift Container Platform.

2. Load the project where the service you want to expose is located.

   ```
   $ oc project project1
   ```

3. Open a text file on the master node and paste the following text, editing the file as needed:

   Example 10.1. Sample load balancer configuration file

   ```
   apiVersion: v1
   kind: Service
   metadata:
     name: egress-2 ①
   spec:
     ports:
     - name: db
       port: 3306 ②
     loadBalancerIP:
     type: LoadBalancer ③
     selector:
       name: mysql ④
   ```

   ① Enter a descriptive name for the load balancer service.

   ② Enter the same port that the service you want to expose is listening on.

   ③ Enter **loadbalancer** as the type.

   ④ Enter the name of the service.

4. Save and exit the file.

5. Run the following command to create the service:

   ```
   oc create -f <file-name>
   ```

   For example:

```
oc create -f mysql-lb.yaml
```

6. Execute the following command to view the new service:

```
$ oc get svc
NAME      TYPE         CLUSTER-IP      EXTERNAL-IP                          PORT(S)
AGE
egress-2   LoadBalancer   172.30.22.226   ad42f5d8b303045-487804948.example.com
3306:30357/TCP   15m
```

The service has an external IP address automatically assigned if there is a cloud provider enabled.

7. On the master, use a tool, such as cURL, to make sure you can reach the service using the public IP address:

```
$ curl <public-ip>:<port>
```

++ For example:

```
$ curl 172.29.121.74:3306
```

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 connecting with 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)]>
```

10.4. CONFIGURING INGRESS CLUSTER TRAFFIC USING A SERVICE EXTERNAL IP

OpenShift Container Platform provides methods for communicating from outside the cluster with services running in the cluster. This method uses a service external IP.

10.4.1. Using a service external IP to get traffic into the cluster

One method to expose a service is to assign an external IP address directly to the service you want to make accessible from outside the cluster.

The external IP address that you use must be provisioned on your infrastructure platform and attached to a cluster node.

With an external IP on the service, OpenShift Container Platform sets up sets up NAT rules to allow traffic arriving at any cluster node attached to that IP address to be sent to one of the internal pods. This is similar to the internal service IP addresses, but the external IP tells OpenShift Container Platform that this service should also be exposed externally at the given IP. The administrator must assign the IP address to a host (node) interface on one of the nodes in the cluster. Alternatively, the address can be used as a virtual IP (VIP).

These IPs are not managed by OpenShift Container Platform and administrators are responsible for ensuring that traffic arrives at a node with this IP.

NOTE

The procedures in this section require prerequisites performed by the cluster administrator.

Prerequisites

Before starting the following procedures, the administrator must:

- Set up the external port to the cluster networking environment so that requests can reach the cluster.
- Make sure there is at least one user with cluster admin role. To add this role to a user, run the following command:

  ```
  oc adm policy add-cluster-role-to-user cluster-admin username
  ```

- Have an OpenShift Container Platform cluster with at least one master and at least one node and a system outside the cluster that has network access to the cluster. This procedure assumes that the external system is on the same subnet as the cluster. The additional networking required for external systems on a different subnet is out-of-scope for this topic.

10.4.2. 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.

Procedure

1. Log in to OpenShift Container Platform.
2. Create a new project for your service:

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

   For example:

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

3. 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/openshift3/mysql-55-rhel7
   ```

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

```
$ oc get svc -n openshift-ingress
NAME                     TYPE           CLUSTER-IP      EXTERNAL-IP      PORT(S)
AGE
router-default           LoadBalancer   172.30.16.119   52.230.228.163
80:30745/TCP,443:32561/TCP   2d6h
router-internal-default   ClusterIP      172.30.101.15   <none>
80/TCP,443/TCP,1936/TCP      2d6h
```

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

10.4.3. Exposing the service by creating a route

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

Procedure

To expose the service:

1. Log in to OpenShift Container Platform.

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-55-rhel7
route "mysql-55-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)]>
```

10.5. CONFIGURING INGRESS CLUSTER TRAFFIC USING A NODEPORT

OpenShift Container Platform provides methods for communicating from outside the cluster with services running in the cluster. This method uses a **NodePort**.

10.5.1. Using a NodePort to get traffic into the cluster

Use a **NodePort**-type **Service** resource to expose a service on a specific port on all nodes in the cluster. The port is specified in the **Service** resource's **.spec.ports[*].nodePort** field.

IMPORTANT

Using **NodePort**s requires additional port resources.

A **NodePort** exposes the service on a static port on the node's IP address. **NodePort**s are in the **30000** to **32767** range by default, which means a **NodePort** is unlikely to match a service's intended port. For example, port **8080** may be exposed as port **31020** on the node.

The administrator must ensure the external IP addresses are routed to the nodes.

NodePorts and external IPs are independent and both can be used concurrently.

NOTE

The procedures in this section require prerequisites performed by the cluster administrator.

Prerequisites

Before starting the following procedures, the administrator must:

- Set up the external port to the cluster networking environment so that requests can reach the cluster.
- Make sure there is at least one user with cluster admin role. To add this role to a user, run the following command:

```
$ oc adm policy add-cluster-role-to-user cluster-admin <user_name>
```

- Have an OpenShift Container Platform cluster with at least one master and at least one node and a system outside the cluster that has network access to the cluster. This procedure assumes that the external system is on the same subnet as the cluster. The additional networking required for external systems on a different subnet is out-of-scope for this topic.

10.5.2. 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.

Procedure

1. Log in to OpenShift Container Platform.

2. Create a new project for your service:

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

   For example:

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

3. 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/openshift3/mysql-55-rhel7
   ```

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

   ```
   $ oc get svc -n openshift-ingress
   NAME                      TYPE           CLUSTER-IP      EXTERNAL-IP      PORT(S)
   AGE
   router-default            LoadBalancer   172.30.16.119   52.230.228.163
   80:30745/TCP,443:32561/TCP   2d6h
   router-internal-default   ClusterIP      172.30.101.15   <none>
   80/TCP,443/TCP,1936/TCP      2d6h
   ```

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

10.5.3. Exposing the service by creating a route

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

Procedure

To expose the service:

1. Log in to OpenShift Container Platform.

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-55-rhel7
   route "mysql-55-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)]>
    ```