# Administer a Cluster

Learn common tasks for administering a cluster.

# 1 - Administration with kubeadm

# 1.1 - Certificate Management with kubeadm

**FEATURE STATE:** `Kubernetes v1.15 [stable]`

Client certificates generated by kubeadm expire after 1 year. This page explains how to manage certificate renewals with kubeadm.

## Before you begin

You should be familiar with PKI certificates and requirements in Kubernetes.

## Using custom certificates

By default, kubeadm generates all the certificates needed for a cluster to run. You can override this behavior by providing your own certificates.

To do so, you must place them in whatever directory is specified by the `--cert-dir` flag or the `certificatesDir` field of kubeadm's `ClusterConfiguration`. By default this is `/etc/kubernetes/pki`.

If a given certificate and private key pair exists before running `kubeadm init`, kubeadm does not overwrite them. This means you can, for example, copy an existing CA into `/etc/kubernetes/pki/ca.crt` and `/etc/kubernetes/pki/ca.key`, and kubeadm will use this CA for signing the rest of the certificates.

## External CA mode

It is also possible to provide only the `ca.crt` file and not the `ca.key` file (this is only available for the root CA file, not other cert pairs). If all other certificates and kubeconfig files are in place, kubeadm recognizes this condition and activates the "External CA" mode. kubeadm will proceed without the CA key on disk.

Instead, run the controller-manager standalone with `--controllers=csrsigner` and point to the CA certificate and key.

PKI certificates and requirements includes guidance on setting up a cluster to use an external CA.

## Check certificate expiration

You can use the `check-expiration` subcommand to check when certificates expire:

```
kubeadm certs check-expiration
```

The output is similar to this:

```
CERTIFICATE                EXPIRES                  RESIDUAL TIME   CERTIFICATE AUTHORITY   EXTERN
admin.conf                 Dec 30, 2020 23:36 UTC   364d                                    no
apiserver                  Dec 30, 2020 23:36 UTC   364d            ca                      no
apiserver-etcd-client      Dec 30, 2020 23:36 UTC   364d            etcd-ca                 no
apiserver-kubelet-client   Dec 30, 2020 23:36 UTC   364d            ca                      no
controller-manager.conf    Dec 30, 2020 23:36 UTC   364d                                    no
etcd-healthcheck-client    Dec 30, 2020 23:36 UTC   364d            etcd-ca                 no
etcd-peer                  Dec 30, 2020 23:36 UTC   364d            etcd-ca                 no
etcd-server                Dec 30, 2020 23:36 UTC   364d            etcd-ca                 no
front-proxy-client         Dec 30, 2020 23:36 UTC   364d            front-proxy-ca          no
scheduler.conf             Dec 30, 2020 23:36 UTC   364d                                    no

CERTIFICATE AUTHORITY   EXPIRES                  RESIDUAL TIME   EXTERNALLY MANAGED
ca                      Dec 28, 2029 23:36 UTC   9y              no
etcd-ca                 Dec 28, 2029 23:36 UTC   9y              no
front-proxy-ca          Dec 28, 2029 23:36 UTC   9y              no
```

The command shows expiration/residual time for the client certificates in the `/etc/kubernetes/pki` folder and for the client certificate embedded in the KUBECONFIG files used by kubeadm (`admin.conf`, `controller-manager.conf` and `scheduler.conf`).

Additionally, kubeadm informs the user if the certificate is externally managed; in this case, the user should take care of managing certificate renewal manually/using other tools.

> **Warning:** `kubeadm` cannot manage certificates signed by an external CA.

> **Note:** `kubelet.conf` is not included in the list above because kubeadm configures kubelet for automatic certificate renewal with rotatable certificates under `/var/lib/kubelet/pki`. To repair an expired kubelet client certificate see Kubelet client certificate rotation fails.

> **Warning:**
> On nodes created with `kubeadm init`, prior to kubeadm version 1.17, there is a bug where you manually have to modify the contents of `kubelet.conf`. After `kubeadm init` finishes, you should update `kubelet.conf` to point to the rotated kubelet client certificates, by replacing `client-certificate-data` and `client-key-data` with:
>
> ```
> client-certificate: /var/lib/kubelet/pki/kubelet-client-current.pem
> client-key: /var/lib/kubelet/pki/kubelet-client-current.pem
> ```

# Automatic certificate renewal

kubeadm renews all the certificates during control plane upgrade.

This feature is designed for addressing the simplest use cases; if you don't have specific requirements on certificate renewal and perform Kubernetes version upgrades regularly (less than 1 year in between each upgrade), kubeadm will take care of keeping your cluster up to date and reasonably secure.

> **Note:** It is a best practice to upgrade your cluster frequently in order to stay secure.

If you have more complex requirements for certificate renewal, you can opt out from the default behavior by passing `--certificate-renewal=false` to `kubeadm upgrade apply` or to `kubeadm upgrade node`.

> **Warning:** Prior to kubeadm version 1.17 there is a bug where the default value for `--certificate-renewal` is `false` for the `kubeadm upgrade node` command. In that case, you should explicitly set `--certificate-renewal=true`.

# Manual certificate renewal

You can renew your certificates manually at any time with the `kubeadm certs renew` command.

This command performs the renewal using CA (or front-proxy-CA) certificate and key stored in `/etc/kubernetes/pki`.

> **Warning:** If you are running an HA cluster, this command needs to be executed on all the control-plane nodes.

> **Note:** `certs renew` uses the existing certificates as the authoritative source for attributes (Common Name, Organization, SAN, etc.) instead of the kubeadm-config ConfigMap. It is strongly recommended to keep them both in sync.

`kubeadm certs renew` provides the following options:

The Kubernetes certificates normally reach their expiration date after one year.

- `--csr-only` can be used to renew certificates with an external CA by generating certificate signing requests (without actually renewing certificates in place); see next paragraph for more information.
- It's also possible to renew a single certificate instead of all.

# Renew certificates with the Kubernetes certificates API

This section provides more details about how to execute manual certificate renewal using the Kubernetes certificates API.

> **Caution:** These are advanced topics for users who need to integrate their organization's certificate infrastructure into a kubeadm-built cluster. If the default kubeadm configuration satisfies your needs, you should let kubeadm manage certificates instead.

## Set up a signer

The Kubernetes Certificate Authority does not work out of the box. You can configure an external signer such as cert-manager, or you can use the built-in signer.

The built-in signer is part of `kube-controller-manager`.

To activate the built-in signer, you must pass the `--cluster-signing-cert-file` and `--cluster-signing-key-file` flags.

If you're creating a new cluster, you can use a kubeadm configuration file:

```
apiVersion: kubeadm.k8s.io/v1beta2
kind: ClusterConfiguration
controllerManager:
  extraArgs:
    cluster-signing-cert-file: /etc/kubernetes/pki/ca.crt
    cluster-signing-key-file: /etc/kubernetes/pki/ca.key
```

### Create certificate signing requests (CSR)

See Create CertificateSigningRequest for creating CSRs with the Kubernetes API.

## Renew certificates with external CA

This section provide more details about how to execute manual certificate renewal using an external CA.

To better integrate with external CAs, kubeadm can also produce certificate signing requests (CSRs). A CSR represents a request to a CA for a signed certificate for a client. In kubeadm terms, any certificate that would normally be signed by an on-disk CA can be produced as a CSR instead. A CA, however, cannot be produced as a CSR.

### Create certificate signing requests (CSR)

You can create certificate signing requests with `kubeadm certs renew --csr-only`.

Both the CSR and the accompanying private key are given in the output. You can pass in a directory with `--csr-dir` to output the CSRs to the specified location. If `--csr-dir` is not specified, the default certificate directory (`/etc/kubernetes/pki`) is used.

Certificates can be renewed with `kubeadm certs renew --csr-only`. As with `kubeadm init`, an output directory can be specified with the `--csr-dir` flag. </>

A CSR contains a certificate's name, domains, and IPs, but it does not specify usages. It is the responsibility of the CA to specify the correct cert usages when issuing a certificate.

- In `openssl` this is done with the `openssl ca` command.
- In `cfssl` you specify usages in the config file.

After a certificate is signed using your preferred method, the certificate and the private key must be copied to the PKI directory (by default `/etc/kubernetes/pki`).

## Certificate authority (CA) rotation

Kubeadm does not support rotation or replacement of CA certificates out of the box.

For more information about manual rotation or replacement of CA, see manual rotation of CA certificates.

## Enabling signed kubelet serving certificates

By default the kubelet serving certificate deployed by kubeadm is self-signed. This means a connection from external services like the metrics-server to a kubelet cannot be secured with TLS.

To configure the kubelets in a new kubeadm cluster to obtain properly signed serving certificates you must pass the following minimal configuration to `kubeadm init`:

```
apiVersion: kubeadm.k8s.io/v1beta2
kind: ClusterConfiguration
---
apiVersion: kubelet.config.k8s.io/v1beta1
kind: KubeletConfiguration
serverTLSBootstrap: true
```

If you have already created the cluster you must adapt it by doing the following:

- Find and edit the `kubelet-config-1.21` ConfigMap in the `kube-system` namespace. In that ConfigMap, the `kubelet` key has a KubeletConfiguration document as its value. Edit the KubeletConfiguration document to set `serverTLSBootstrap: true`.
- On each node, add the `serverTLSBootstrap: true` field in `/var/lib/kubelet/config.yaml` and restart the kubelet with `systemctl restart kubelet`

The field `serverTLSBootstrap: true` will enable the bootstrap of kubelet serving certificates by requesting them from the `certificates.k8s.io` API. One known limitation is that the CSRs (Certificate Signing Requests) for these certificates cannot be automatically approved by the default signer in the kube-controller-manager - `kubernetes.io/kubelet-serving`. This will require action from the user or a third party controller.

These CSRs can be viewed using:

```
kubectl get csr
NAME        AGE     SIGNERNAME                      REQUESTOR                    CONDITION
csr-9wvgt   112s    kubernetes.io/kubelet-serving   system:node:worker-1         Pending
csr-lz97v   1m58s   kubernetes.io/kubelet-serving   system:node:control-plane-1  Pending
```

To approve them you can do the following:

```
kubectl certificate approve <CSR-name>
```

By default, these serving certificate will expire after one year. Kubeadm sets the `KubeletConfiguration` field `rotateCertificates` to `true`, which means that close to expiration a new set of CSRs for the serving certificates will be created and must be approved to complete the rotation. To understand more see Certificate Rotation.

If you are looking for a solution for automatic approval of these CSRs it is recommended that you contact your cloud provider and ask if they have a CSR signer that verifies the node identity with an out of band mechanism.

> **Caution:** This section links to third party projects that provide functionality required by Kubernetes. The Kubernetes project authors aren't responsible for these projects. This page follows CNCF website guidelines by listing projects alphabetically. To add a project to this list, read the content guide before submitting a change.

Third party custom controllers can be used:

- kubelet-rubber-stamp

Such a controller is not a secure mechanism unless it not only verifies the CommonName in the CSR but also verifies the requested IPs and domain names. This would prevent a malicious actor that has access to a kubelet client certificate to create CSRs requesting serving certificates for any IP or domain name.

</>

</>

# 1.2 - Configuring a cgroup driver

This page explains how to configure the kubelet cgroup driver to match the container runtime cgroup driver for kubeadm clusters.

## Before you begin

You should be familiar with the Kubernetes container runtime requirements.

## Configuring the container runtime cgroup driver

The Container runtimes page explains that the `systemd` driver is recommended for kubeadm based setups instead of the `cgroupfs` driver, because kubeadm manages the kubelet as a systemd service.

The page also provides details on how to setup a number of different container runtimes with the `systemd` driver by default.

## Configuring the kubelet cgroup driver

kubeadm allows you to pass a `KubeletConfiguration` structure during `kubeadm init`. This `KubeletConfiguration` can include the `cgroupDriver` field which controls the cgroup driver of the kubelet.

> **Note:**
>
> **FEATURE STATE:** `Kubernetes v1.21 [stable]`
>
> If the user is not setting the `cgroupDriver` field under `KubeletConfiguration`, `kubeadm init` will default it to `systemd`.

A minimal example of configuring the field explicitly:

```yaml
# kubeadm-config.yaml
kind: ClusterConfiguration
apiVersion: kubeadm.k8s.io/v1beta2
kubernetesVersion: v1.21.0
---
kind: KubeletConfiguration
apiVersion: kubelet.config.k8s.io/v1beta1
cgroupDriver: systemd
```

Such a configuration file can then be passed to the kubeadm command:

```
kubeadm init --config kubeadm-config.yaml
```

> **Note:**
> Kubeadm uses the same `KubeletConfiguration` for all nodes in the cluster. The `KubeletConfiguration` is stored in a ConfigMap object under the `kube-system` namespace.
>
> Executing the sub commands `init`, `join` and `upgrade` would result in kubeadm writing the `KubeletConfiguration` as a file under `/var/lib/kubelet/config.yaml` and passing it to the local node kubelet.

## Using the `cgroupfs` driver

As this guide explains using the `cgroupfs` driver with kubeadm is not recommended.

To continue using `cgroupfs` and to prevent `kubeadm upgrade` from modifying the `KubeletConfiguration` cgroup driver on existing setups, you must be explicit about its value. This applies to a case where you do not wish future versions of kubeadm to apply the `systemd` driver by default.

See the below section on "Modify the kubelet ConfigMap" for details on how to be explicit about the value.

If you wish to configure a container runtime to use the `cgroupfs` driver, you must refer to the documentation of the container runtime of your choice.

# Migrating to the `systemd` driver

To change the cgroup driver of an existing kubeadm cluster to `systemd` in-place, a similar procedure to a kubelet upgrade is required. This must include both steps outlined below.

> **Note:** Alternatively, it is possible to replace the old nodes in the cluster with new ones that use the `systemd` driver. This requires executing only the first step below before joining the new nodes and ensuring the workloads can safely move to the new nodes before deleting the old nodes.

## Modify the kubelet ConfigMap

- Find the kubelet ConfigMap name using `kubectl get cm -n kube-system | grep kubelet-config`.
- Call `kubectl edit cm kubelet-config-x.yy -n kube-system` (replace `x.yy` with the Kubernetes version).
- Either modify the existing `cgroupDriver` value or add a new field that looks like this:

```
cgroupDriver: systemd
```

  This field must be present under the `kubelet:` section of the ConfigMap.

## Update the cgroup driver on all nodes

For each node in the cluster:

- Drain the node using `kubectl drain <node-name> --ignore-daemonsets`
- Stop the kubelet using `systemctl stop kubelet`
- Stop the container runtime
- Modify the container runtime cgroup driver to `systemd`
- Set `cgroupDriver: systemd` in `/var/lib/kubelet/config.yaml`
- Start the container runtime
- Start the kubelet using `systemctl start kubelet`
- Uncordon the node using `kubectl uncordon <node-name>`

Execute these steps on nodes one at a time to ensure workloads have sufficient time to schedule on different nodes.

Once the process is complete ensure that all nodes and workloads are healthy.

# 1.3 - Upgrading kubeadm clusters

This page explains how to upgrade a Kubernetes cluster created with kubeadm from version 1.20.x to version 1.21.x, and from version 1.21.x to 1.21.y (where `y > x`). Skipping MINOR versions when upgrading is unsupported.

To see information about upgrading clusters created using older versions of kubeadm, please refer to following pages instead:

- Upgrading a kubeadm cluster from 1.19 to 1.20
- Upgrading a kubeadm cluster from 1.18 to 1.19
- Upgrading a kubeadm cluster from 1.17 to 1.18
- Upgrading a kubeadm cluster from 1.16 to 1.17

The upgrade workflow at high level is the following:

1. Upgrade a primary control plane node.
2. Upgrade additional control plane nodes.
3. Upgrade worker nodes.

## Before you begin

- Make sure you read the release notes carefully.
- The cluster should use a static control plane and etcd pods or external etcd.
- Make sure to back up any important components, such as app-level state stored in a database. `kubeadm upgrade` does not touch your workloads, only components internal to Kubernetes, but backups are always a best practice.
- Swap must be disabled.

### Additional information

- Draining nodes before kubelet MINOR version upgrades is required. In the case of control plane nodes, they could be running CoreDNS Pods or other critical workloads.
- All containers are restarted after upgrade, because the container spec hash value is changed.

## Determine which version to upgrade to

Find the latest stable 1.21 version using the OS package manager:

Ubuntu, Debian or HypriotOS | CentOS, RHEL or Fedora

```
apt update
apt-cache madison kubeadm
# find the latest 1.21 version in the list
# it should look like 1.21.x-00, where x is the latest patch
```

## Upgrading control plane nodes

The upgrade procedure on control plane nodes should be executed one node at a time. Pick a control plane node that you wish to upgrade first. It must have the `/etc/kubernetes/admin.conf` file.

### Call "kubeadm upgrade"

**For the first control plane node**

- Upgrade kubeadm:

Ubuntu, Debian or HypriotOS | CentOS, RHEL or Fedora

```
# replace x in 1.21.x-00 with the latest patch version
apt-mark unhold kubeadm && \
apt-get update && apt-get install -y kubeadm=1.21.x-00 && \
apt-mark hold kubeadm
-
# since apt-get version 1.1 you can also use the following method
apt-get update && \
apt-get install -y --allow-change-held-packages kubeadm=1.21.x-00
```

- Verify that the download works and has the expected version:

```
kubeadm version
```

- Verify the upgrade plan:

```
kubeadm upgrade plan
```

This command checks that your cluster can be upgraded, and fetches the versions you can upgrade to. It also shows a table with the component config version states.

**Note:** `kubeadm upgrade` also automatically renews the certificates that it manages on this node. To opt-out of certificate renewal the flag `--certificate-renewal=false` can be used. For more information see the certificate management guide.

**Note:** If `kubeadm upgrade plan` shows any component configs that require manual upgrade, users must provide a config file with replacement configs to `kubeadm upgrade apply` via the `--config` command line flag. Failing to do so will cause `kubeadm upgrade apply` to exit with an error and not perform an upgrade.

- Choose a version to upgrade to, and run the appropriate command. For example:

```
# replace x with the patch version you picked for this upgrade
sudo kubeadm upgrade apply v1.21.x
```

Once the command finishes you should see:

```
[upgrade/successful] SUCCESS! Your cluster was upgraded to "v1.21.x". Enjoy!

[upgrade/kubelet] Now that your control plane is upgraded, please proceed with upgrading your
```

- Manually upgrade your CNI provider plugin.

Your Container Network Interface (CNI) provider may have its own upgrade instructions to follow. Check the addons page to find your CNI provider and see whether additional upgrade steps are required.

This step is not required on additional control plane nodes if the CNI provider runs as a DaemonSet.

**For the other control plane nodes**

Same as the first control plane node but use:

```
sudo kubeadm upgrade node
```

instead of:

```
sudo kubeadm upgrade apply
```

Also calling `kubeadm upgrade plan` and upgrading the CNI provider plugin is no longer needed.

## Drain the node

- Prepare the node for maintenance by marking it unschedulable and evicting the workloads:

```
# replace <node-to-drain> with the name of your node you are draining
kubectl drain <node-to-drain> --ignore-daemonsets
```

## Upgrade kubelet and kubectl

- Upgrade the kubelet and kubectl

Ubuntu, Debian or HypriotOS | CentOS, RHEL or Fedora

```
    # replace x in 1.21.x-00 with the latest patch version
    apt-mark unhold kubelet kubectl && \
    apt-get update && apt-get install -y kubelet=1.21.x-00 kubectl=1.21.x-00 && \
    apt-mark hold kubelet kubectl
    -
    # since apt-get version 1.1 you can also use the following method
    apt-get update && \
    apt-get install -y --allow-change-held-packages kubelet=1.21.x-00 kubectl=1.21.x-00
```

- Restart the kubelet:

```
sudo systemctl daemon-reload
sudo systemctl restart kubelet
```

## Uncordon the node

- Bring the node back online by marking it schedulable:

```
# replace <node-to-drain> with the name of your node
kubectl uncordon <node-to-drain>
```

# Upgrade worker nodes

The upgrade procedure on worker nodes should be executed one node at a time or few nodes at a time, without compromising the minimum required capacity for running your workloads.

## Upgrade kubeadm

- Upgrade kubeadm:

Ubuntu, Debian or HypriotOS | CentOS, RHEL or Fedora

```
# replace x in 1.21.x-00 with the latest patch version
apt-mark unhold kubeadm && \
apt-get update && apt-get install -y kubeadm=1.21.x-00 && \
apt-mark hold kubeadm
-
# since apt-get version 1.1 you can also use the following method
apt-get update && \
apt-get install -y --allow-change-held-packages kubeadm=1.21.x-00
```

## Call "kubeadm upgrade"

- For worker nodes this upgrades the local kubelet configuration:

```
sudo kubeadm upgrade node
```

## Drain the node

- Prepare the node for maintenance by marking it unschedulable and evicting the workloads:

```
# replace <node-to-drain> with the name of your node you are draining
kubectl drain <node-to-drain> --ignore-daemonsets
```

## Upgrade kubelet and kubectl

- Upgrade the kubelet and kubectl:

Ubuntu, Debian or HypriotOS | CentOS, RHEL or Fedora

```
# replace x in 1.21.x-00 with the latest patch version
apt-mark unhold kubelet kubectl && \
apt-get update && apt-get install -y kubelet=1.21.x-00 kubectl=1.21.x-00 && \
apt-mark hold kubelet kubectl
-
# since apt-get version 1.1 you can also use the following method
apt-get update && \
apt-get install -y --allow-change-held-packages kubelet=1.21.x-00 kubectl=1.21.x-00
```

- Restart the kubelet:

```
sudo systemctl daemon-reload
sudo systemctl restart kubelet
```

## Uncordon the node

- Bring the node back online by marking it schedulable:

```
# replace <node-to-drain> with the name of your node
kubectl uncordon <node-to-drain>
```

## Verify the status of the cluster

After the kubelet is upgraded on all nodes verify that all nodes are available again by running the following command from anywhere kubectl can access the cluster:

```
kubectl get nodes
```

The `STATUS` column should show `Ready` for all your nodes, and the version number should be updated.

## Recovering from a failure state

If `kubeadm upgrade` fails and does not roll back, for example because of an unexpected shutdown during execution, you can run `kubeadm upgrade` again. This command is idempotent and eventually makes sure that the actual state is the desired state you declare.

To recover from a bad state, you can also run `kubeadm upgrade apply --force` without changing the version that your cluster is running.

During upgrade kubeadm writes the following backup folders under `/etc/kubernetes/tmp`:

- `kubeadm-backup-etcd-<date>-<time>`
- `kubeadm-backup-manifests-<date>-<time>`

`kubeadm-backup-etcd` contains a backup of the local etcd member data for this control plane Node. In case of an etcd upgrade failure and if the automatic rollback does not work, the contents of this folder can be manually restored in `/var/lib/etcd`. In case external etcd is used this backup folder will be empty.

`kubeadm-backup-manifests` contains a backup of the static Pod manifest files for this control plane Node. In case of a upgrade failure and if the automatic rollback does not work, the contents of this folder can be manually restored in `/etc/kubernetes/manifests`. If for some reason there is no difference between a pre-upgrade and post-upgrade manifest file for a certain component, a backup file for it will not be written.

## How it works

`kubeadm upgrade apply` does the following:

- Checks that your cluster is in an upgradeable state:
  - The API server is reachable
  - All nodes are in the `Ready` state
  - The control plane is healthy
- Enforces the version skew policies.
- Makes sure the control plane images are available or available to pull to the machine.
- Generates replacements and/or uses user supplied overwrites if component configs require version upgrades.

- Upgrades the control plane components or rollbacks if any of them fails to come up.
- Applies the new `CoreDNS` and `kube-proxy` manifests and makes sure that all necessary RBAC rules are created.
- Creates new certificate and key files of the API server and backs up old files if they're about to expire in 180 days.

`kubeadm upgrade node` does the following on additional control plane nodes:

- Fetches the kubeadm `ClusterConfiguration` from the cluster.
- Optionally backups the kube-apiserver certificate.
- Upgrades the static Pod manifests for the control plane components.
- Upgrades the kubelet configuration for this node.

`kubeadm upgrade node` does the following on worker nodes:

- Fetches the kubeadm `ClusterConfiguration` from the cluster.
- Upgrades the kubelet configuration for this node.

# 1.4 - Adding Windows nodes

**FEATURE STATE:** `Kubernetes v1.18 [beta]`

You can use Kubernetes to run a mixture of Linux and Windows nodes, so you can mix Pods that run on Linux on with Pods that run on Windows. This page shows how to register Windows nodes to your cluster.

## Before you begin

Your Kubernetes server must be at or later than version 1.17. To check the version, enter `kubectl version`.

- Obtain a Windows Server 2019 license (or higher) in order to configure the Windows node that hosts Windows containers. If you are using VXLAN/Overlay networking you must have also have KB4489899 installed.
- A Linux-based Kubernetes kubeadm cluster in which you have access to the control plane (see Creating a single control-plane cluster with kubeadm).

## Objectives

- Register a Windows node to the cluster
- Configure networking so Pods and Services on Linux and Windows can communicate with each other

## Getting Started: Adding a Windows Node to Your Cluster

### Networking Configuration

Once you have a Linux-based Kubernetes control-plane node you are ready to choose a networking solution. This guide illustrates using Flannel in VXLAN mode for simplicity.

### Configuring Flannel

1. Prepare Kubernetes control plane for Flannel

   Some minor preparation is recommended on the Kubernetes control plane in our cluster. It is recommended to enable bridged IPv4 traffic to iptables chains when using Flannel. The following command must be run on all Linux nodes:

   ```
   sudo sysctl net.bridge.bridge-nf-call-iptables=1
   ```

2. Download & configure Flannel for Linux

   Download the most recent Flannel manifest:

   ```
   wget https://raw.githubusercontent.com/coreos/flannel/master/Documentation/kube-flannel.yml
   ```

   Modify the `net-conf.json` section of the flannel manifest in order to set the VNI to 4096 and the Port to 4789. It should look as follows:

   ```
   net-conf.json: |
       {
         "Network": "10.244.0.0/16",
         "Backend": {
           "Type": "vxlan",
           "VNI": 4096,
           "Port": 4789
         }
       }
   ```

   **Note:** The VNI must be set to 4096 and port 4789 for Flannel on Linux to interoperate with Flannel on Windows. See the VXLAN documentation. for an explanation of these fields.

   **Note:** To use L2Bridge/Host-gateway mode instead change the value of `Type` to `"host-gw"` and omit `VNI` and `Port`.

3. Apply the Flannel manifest and validate

Let's apply the Flannel configuration:

```
kubectl apply -f kube-flannel.yml
```

After a few minutes, you should see all the pods as running if the Flannel pod network was deployed.

```
kubectl get pods -n kube-system
```

The output should include the Linux flannel DaemonSet as running:

```
NAMESPACE     NAME                          READY         STATUS    RESTARTS    AGE
...
kube-system   kube-flannel-ds-54954         1/1           Running   0           1m
```

4. Add Windows Flannel and kube-proxy DaemonSets

   Now you can add Windows-compatible versions of Flannel and kube-proxy. In order to ensure that you get a compatible version of kube-proxy, you'll need to substitute the tag of the image. The following example shows usage for Kubernetes v1.21.0, but you should adjust the version for your own deployment.

```
curl -L https://github.com/kubernetes-sigs/sig-windows-tools/releases/latest/download/kube-pr
kubectl apply -f https://github.com/kubernetes-sigs/sig-windows-tools/releases/latest/downloa
```

> **Note:** If you're using host-gateway use https://github.com/kubernetes-sigs/sig-windows-tools/releases/latest/download/flannel-host-gw.yml instead

> **Note:**
> If you're using a different interface rather than Ethernet (i.e. "Ethernet0 2") on the Windows nodes, you have to modify the line:

```
wins cli process run --path /k/flannel/setup.exe --args "--mode=overlay --interface=Ether
```

in the `flannel-host-gw.yml` or `flannel-overlay.yml` file and specify your interface accordingly.

```
# Example
curl -L https://github.com/kubernetes-sigs/sig-windows-tools/releases/latest/download/fla
```

## Joining a Windows worker node

> **Note:** All code snippets in Windows sections are to be run in a PowerShell environment with elevated permissions (Administrator) on the Windows worker node.

Docker EE | CRI-containerD

### Install Docker EE

Install the `Containers` feature

```
Install-WindowsFeature -Name containers
```

Install Docker Instructions to do so are available at Install Docker Engine - Enterprise on Windows Servers.

### Install wins, kubelet, and kubeadm

```
curl.exe -LO https://github.com/kubernetes-sigs/sig-windows-tools/releases/latest/download/Pr
.\PrepareNode.ps1 -KubernetesVersion v1.21.0
```

### Run `kubeadm` to join the node

Use the command that was given to you when you ran `kubeadm init` on a control plane host. If you no longer have this command, or the token has expired, you can run `kubeadm token create --print-join-command` (on a control plane host) to generate a new token and join command.

## Verifying your installation

You should now be able to view the Windows node in your cluster by running:

```
kubectl get nodes -o wide
```

If your new node is in the `NotReady` state it is likely because the flannel image is still downloading. You can check the progress as before by checking on the flannel pods in the `kube-system` namespace:

```
kubectl -n kube-system get pods -l app=flannel
```

Once the flannel Pod is running, your node should enter the `Ready` state and then be available to handle workloads.

# What's next

- Upgrading Windows kubeadm nodes

# 1.5 - Upgrading Windows nodes

**FEATURE STATE:** `Kubernetes v1.18 [beta]`

This page explains how to upgrade a Windows node created with kubeadm.

## Before you begin

You need to have a Kubernetes cluster, and the kubectl command-line tool must be configured to communicate with your cluster. It is recommended to run this tutorial on a cluster with at least two nodes that are not acting as control plane hosts. If you do not already have a cluster, you can create one by using minikube or you can use one of these Kubernetes playgrounds:

- Katacoda
- Play with Kubernetes

Your Kubernetes server must be at or later than version 1.17. To check the version, enter `kubectl version`.

- Familiarize yourself with the process for upgrading the rest of your kubeadm cluster. You will want to upgrade the control plane nodes before upgrading your Windows nodes.

## Upgrading worker nodes

### Upgrade kubeadm

1. From the Windows node, upgrade kubeadm:

```
# replace v1.21.0 with your desired version
curl.exe -Lo C:\k\kubeadm.exe https://dl.k8s.io/v1.21.0/bin/windows/amd64/kubeadm.exe
```

### Drain the node

1. From a machine with access to the Kubernetes API, prepare the node for maintenance by marking it unschedulable and evicting the workloads:

```
# replace <node-to-drain> with the name of your node you are draining
kubectl drain <node-to-drain> --ignore-daemonsets
```

You should see output similar to this:

```
node/ip-172-31-85-18 cordoned
node/ip-172-31-85-18 drained
```

### Upgrade the kubelet configuration

1. From the Windows node, call the following command to sync new kubelet configuration:

```
kubeadm upgrade node
```

### Upgrade kubelet

1. From the Windows node, upgrade and restart the kubelet:

```
stop-service kubelet
curl.exe -Lo C:\k\kubelet.exe https://dl.k8s.io/v1.21.0/bin/windows/amd64/kubelet.exe
restart-service kubelet
```

### Uncordon the node

1. From a machine with access to the Kubernetes API, bring the node back online by marking it schedulable:

```
# replace <node-to-drain> with the name of your node
kubectl uncordon <node-to-drain>
```

## Upgrade kube-proxy

1. From a machine with access to the Kubernetes API, run the following, again replacing v1.21.0 with your desired version:

```
curl -L https://github.com/kubernetes-sigs/sig-windows-tools/releases/latest/download/kube-pr
```

# 2 - Migrating from dockershim

This section presents information you need to know when migrating from dockershim to other container runtimes.

Since the announcement of dockershim deprecation in Kubernetes 1.20, there were questions on how this will affect various workloads and Kubernetes installations. You can find this blog post useful to understand the problem better: Dockershim Deprecation FAQ

It is recommended to migrate from dockershim to alternative container runtimes. Check out container runtimes section to know your options. Make sure to report issues you encountered with the migration. So the issue can be fixed in a timely manner and your cluster would be ready for dockershim removal.

# 2.1 - Check whether Dockershim deprecation affects you

The `dockershim` component of Kubernetes allows to use Docker as a Kubernetes's container runtime. Kubernetes' built-in `dockershim` component was deprecated in release v1.20.

This page explains how your cluster could be using Docker as a container runtime, provides details on the role that `dockershim` plays when in use, and shows steps you can take to check whether any workloads could be affected by `dockershim` deprecation.

## Finding if your app has a dependencies on Docker

If you are using Docker for building your application containers, you can still run these containers on any container runtime. This use of Docker does not count as a dependency on Docker as a container runtime.

When alternative container runtime is used, executing Docker commands may either not work or yield unexpected output. This is how you can find whether you have a dependency on Docker:

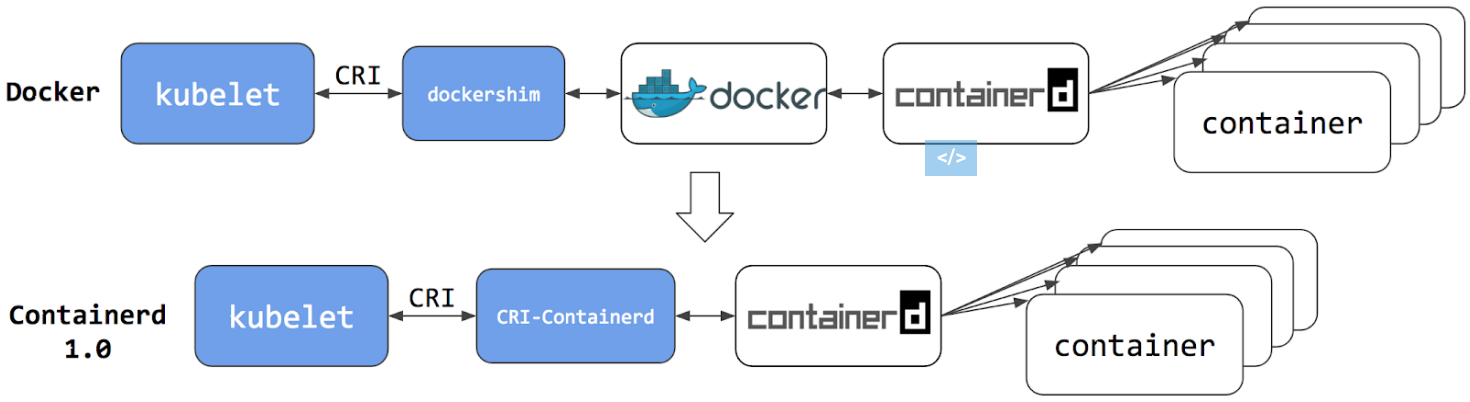
1. Make sure no privileged Pods execute Docker commands.
2. Check that scripts and apps running on nodes outside of Kubernetes infrastructure do not execute Docker commands. It might be:
   - SSH to nodes to troubleshoot;
   - Node startup scripts;
   - Monitoring and security agents installed on nodes directly.
3. Third-party tools that perform above mentioned privileged operations. See Migrating telemetry and security agents from dockershim for more information.
4. Make sure there is no indirect dependencies on dockershim behavior. This is an edge case and unlikely to affect your application. Some tooling may be configured to react to Docker-specific behaviors, for example, raise alert on specific metrics or search for a specific log message as part of troubleshooting instructions. If you have such tooling configured, test the behavior on test cluster before migration.

## Dependency on Docker explained

A container runtime is software that can execute the containers that make up a Kubernetes pod. Kubernetes is responsible for orchestration and scheduling of Pods; on each node, the kubelet uses the container runtime interface as an abstraction so that you can use any compatible container runtime.

In its earliest releases, Kubernetes offered compatibility with one container runtime: Docker. Later in the Kubernetes project's history, cluster operators wanted to adopt additional container runtimes. The CRI was designed to allow this kind of flexibility - and the kubelet began supporting CRI. However, because Docker existed before the CRI specification was invented, the Kubernetes project created an adapter component, `dockershim`. The dockershim adapter allows the kubelet to interact with Docker as if Docker were a CRI compatible runtime.

You can read about it in Kubernetes Containerd integration goes GA blog post.




Switching to Containerd as a container runtime eliminates the middleman. All the same containers can be run by container runtimes like Containerd as before. But now, since containers schedule directly with the container runtime, they are not visible to Docker. So any Docker tooling or fancy UI you might have used before to check on these containers is no longer available.

You cannot get container information using `docker ps` or `docker inspect` commands. As you cannot list containers, you cannot get logs, stop containers, or execute something inside container using `docker exec`.

> **Note:** If you're running workloads via Kubernetes, the best way to stop a container is through the Kubernetes API rather than directly through the container runtime (this advice applies for all container runtimes, not only Docker).

You can still pull images or build them using `docker build` command. But images built or pulled by Docker would not be visible to container runtime and Kubernetes. They needed to be pushed to some registry to allow them to be used by Kubernetes.

# 2.2 - Migrating telemetry and security agents from dockershim

With Kubernetes 1.20 dockershim was deprecated. From the Dockershim Deprecation FAQ you might already know that most apps do not have a direct dependency on runtime hosting containers. However, there are still a lot of telemetry and security agents that has a dependency on docker to collect containers metadata, logs and metrics. This document aggregates information on how to detect these dependencies and links on how to migrate these agents to use generic tools or alternative runtimes.

## Telemetry and security agents

There are a few ways agents may run on Kubernetes cluster. Agents may run on nodes directly or as DaemonSets.

### Why do telemetry agents rely on Docker?

Historically, Kubernetes was built on top of Docker. Kubernetes is managing networking and scheduling, Docker was placing and operating containers on a node. So you can get scheduling-related metadata like a pod name from Kubernetes and containers state information from Docker. Over time more runtimes were created to manage containers. Also there are projects and Kubernetes features that generalize container status information extraction across many runtimes.

Some agents are tied specifically to the Docker tool. The agents may run commands like `docker ps` or `docker top` to list containers and processes or docker logs to subscribe on docker logs. With the deprecating of Docker as a container runtime, these commands will not work any longer.

### Identify DaemonSets that depend on Docker

If a pod wants to make calls to the `dockerd` running on the node, the pod must either:

- mount the filesystem containing the Docker daemon's privileged socket, as a volume; or
- mount the specific path of the Docker daemon's privileged socket directly, also as a volume.

For example: on COS images, Docker exposes its Unix domain socket at `/var/run/docker.sock` This means that the pod spec will include a `hostPath` volume mount of `/var/run/docker.sock`.

Here's a sample shell script to find Pods that have a mount directly mapping the Docker socket. This script outputs the namespace and name of the pod. You can remove the grep `/var/run/docker.sock` to review other mounts.

```
kubectl get pods --all-namespaces \
-o=jsonpath='{range .items[*]}{"\n"}{.metadata.namespace}{":\t"}{.metadata.name}{":\t"}{range .sp
| sort \
| grep '/var/run/docker.sock'
```

**Note:** There are alternative ways for a pod to access Docker on the host. For instance, the parent directory `/var/run` may be mounted instead of the full path (like in this example). The script above only detects the most common uses.

### Detecting Docker dependency from node agents

In case your cluster nodes are customized and install additional security and telemetry agents on the node, make sure to check with the vendor of the agent whether it has dependency on Docker.

### Telemetry and security agent vendors

We keep the work in progress version of migration instructions for various telemetry and security agent vendors in Google doc. Please contact the vendor to get up to date instructions for migrating from dockershim.

# 3 - Certificates

When using client certificate authentication, you can generate certificates manually through `easyrsa`, `openssl` or `cfssl`.

## easyrsa

**easyrsa** can manually generate certificates for your cluster.

1. Download, unpack, and initialize the patched version of easyrsa3.

```
curl -LO https://storage.googleapis.com/kubernetes-release/easy-rsa/easy-rsa.tar.gz
tar xzf easy-rsa.tar.gz
cd easy-rsa-master/easyrsa3
./easyrsa init-pki
```

2. Generate a new certificate authority (CA). `--batch` sets automatic mode; `--req-cn` specifies the Common Name (CN) for the CA's new root certificate.

```
./easyrsa --batch "--req-cn=${MASTER_IP}@`date +%s`" build-ca nopass
```

3. Generate server certificate and key. The argument `--subject-alt-name` sets the possible IPs and DNS names the API server will be accessed with. The `MASTER_CLUSTER_IP` is usually the first IP from the service CIDR that is specified as the `--service-cluster-ip-range` argument for both the API server and the controller manager component. The argument `--days` is used to set the number of days after which the certificate expires. The sample below also assumes that you are using `cluster.local` as the default DNS domain name.

```
./easyrsa --subject-alt-name="IP:${MASTER_IP},"\
"IP:${MASTER_CLUSTER_IP},"\
"DNS:kubernetes,"\
"DNS:kubernetes.default,"\
"DNS:kubernetes.default.svc,"\
"DNS:kubernetes.default.svc.cluster,"\
"DNS:kubernetes.default.svc.cluster.local" \
--days=10000 \
build-server-full server nopass
```

4. Copy `pki/ca.crt`, `pki/issued/server.crt`, and `pki/private/server.key` to your directory.

5. Fill in and add the following parameters into the API server start parameters:

```
--client-ca-file=/yourdirectory/ca.crt
--tls-cert-file=/yourdirectory/server.crt
--tls-private-key-file=/yourdirectory/server.key
```

## openssl

**openssl** can manually generate certificates for your cluster.

1. Generate a ca.key with 2048bit:

```
openssl genrsa -out ca.key 2048
```

2. According to the ca.key generate a ca.crt (use -days to set the certificate effective time):

```
openssl req -x509 -new -nodes -key ca.key -subj "/CN=${MASTER_IP}" -days 10000 -out ca.crt
```

3. Generate a server.key with 2048bit:

```
openssl genrsa -out server.key 2048
```

4. Create a config file for generating a Certificate Signing Request (CSR). Be sure to substitute the values marked with angle brackets (e.g. `<MASTER_IP>`) with real values before saving this to a file (e.g. `csr.conf`). Note that the value for `MASTER_CLUSTER_IP` is the service cluster IP for the API server as described in previous subsection. The sample below also assumes that you are using `cluster.local` as the default DNS domain name.

```
[ req ]
default_bits = 2048
prompt = no
default_md = sha256
req_extensions = req_ext
distinguished_name = dn

[ dn ]
C = <country>
ST = <state>
L = <city>
O = <organization>
OU = <organization unit>
CN = <MASTER_IP>

[ req_ext ]
subjectAltName = @alt_names

[ alt_names ]
DNS.1 = kubernetes
DNS.2 = kubernetes.default
DNS.3 = kubernetes.default.svc
DNS.4 = kubernetes.default.svc.cluster
DNS.5 = kubernetes.default.svc.cluster.local
IP.1 = <MASTER_IP>
IP.2 = <MASTER_CLUSTER_IP>

[ v3_ext ]
authorityKeyIdentifier=keyid,issuer:always
basicConstraints=CA:FALSE
keyUsage=keyEncipherment,dataEncipherment
extendedKeyUsage=serverAuth,clientAuth
subjectAltName=@alt_names
```

5. Generate the certificate signing request based on the config file:

```
openssl req -new -key server.key -out server.csr -config csr.conf
```

6. Generate the server certificate using the ca.key, ca.crt and server.csr:

```
openssl x509 -req -in server.csr -CA ca.crt -CAkey ca.key \
-CAcreateserial -out server.crt -days 10000 \
-extensions v3_ext -extfile csr.conf
```

7. View the certificate:

```
openssl x509  -noout -text -in ./server.crt
```

Finally, add the same parameters into the API server start parameters.

## cfssl

**cfssl** is another tool for certificate generation.

1. Download, unpack and prepare the command line tools as shown below. Note that you may need to adapt the sample commands based on the hardware architecture and cfssl version you are using.

```
curl -L https://github.com/cloudflare/cfssl/releases/download/v1.5.0/cfssl_1.5.0_linux_amd64
chmod +x cfssl
curl -L https://github.com/cloudflare/cfssl/releases/download/v1.5.0/cfssljson_1.5.0_linux_am
chmod +x cfssljson
curl -L https://github.com/cloudflare/cfssl/releases/download/v1.5.0/cfssl-certinfo_1.5.0_lin
chmod +x cfssl-certinfo
```

2. Create a directory to hold the artifacts and initialize cfssl:

```
mkdir cert
cd cert
../cfssl print-defaults config > config.json
../cfssl print-defaults csr > csr.json
```

3. Create a JSON config file for generating the CA file, for example, `ca-config.json`:

```
{
  "signing": {
    "default": {
      "expiry": "8760h"
    },
    "profiles": {
      "kubernetes": {
        "usages": [
          "signing",
          "key encipherment",
          "server auth",
          "client auth"
        ],
        "expiry": "8760h"
      }
    }
  }
}
```

4. Create a JSON config file for CA certificate signing request (CSR), for example, `ca-csr.json`. Be sure to replace the values marked with angle brackets with real values you want to use.

```
{
  "CN": "kubernetes",
  "key": {
    "algo": "rsa",
    "size": 2048
  },
  "names":[{
    "C": "<country>",
    "ST": "<state>",
    "L": "<city>",
    "O": "<organization>",
    "OU": "<organization unit>"
  }]
}
```

5. Generate CA key (`ca-key.pem`) and certificate (`ca.pem`):

```
../cfssl gencert -initca ca-csr.json | ../cfssljson -bare ca
```

6. Create a JSON config file for generating keys and certificates for the API server, for example, `server-csr.json`. Be sure to replace the values in angle brackets with real values you want to use. The `MASTER_CLUSTER_IP` is the service cluster IP for the API server as described in previous subsection. The sample below also assumes that you are using `cluster.local` as the default DNS domain name.

```
{
  "CN": "kubernetes",
  "hosts": [
    "127.0.0.1",
    "<MASTER_IP>",
    "<MASTER_CLUSTER_IP>",
    "kubernetes",
    "kubernetes.default",
    "kubernetes.default.svc",
    "kubernetes.default.svc.cluster",
    "kubernetes.default.svc.cluster.local"
  ],
  "key": {
    "algo": "rsa",
    "size": 2048
  },
  "names": [{
    "C": "<country>",
    "ST": "<state>",
    "L": "<city>",
    "O": "<organization>",
    "OU": "<organization unit>"
  }]
}
```

7. Generate the key and certificate for the API server, which are by default saved into file `server-key.pem` and `server.pem` respectively:

```
../cfssl gencert -ca=ca.pem -ca-key=ca-key.pem \
--config=ca-config.json -profile=kubernetes \
server-csr.json | ../cfssljson -bare server
```

# Distributing Self-Signed CA Certificate

A client node may refuse to recognize a self-signed CA certificate as valid. For a non-production deployment, or for a deployment that runs behind a company firewall, you can distribute a self-signed CA certificate to all clients and refresh the local list for valid certificates.

On each client, perform the following operations:

```
sudo cp ca.crt /usr/local/share/ca-certificates/kubernetes.crt
sudo update-ca-certificates
```

```
Updating certificates in /etc/ssl/certs...
1 added, 0 removed; done.
Running hooks in /etc/ca-certificates/update.d....
done.
```

# Certificates API

You can use the `certificates.k8s.io` API to provision x509 certificates to use for authentication as documented here.

# 4 - Manage Memory, CPU, and API Resources

# 4.1 - Configure Default Memory Requests and Limits for a Namespace

This page shows how to configure default memory requests and limits for a namespace. If a Container is created in a namespace that has a default memory limit, and the Container does not specify its own memory limit, then the Container is assigned the default memory limit. Kubernetes assigns a default memory request under certain conditions that are explained later in this topic.

## Before you begin

You need to have a Kubernetes cluster, and the kubectl command-line tool must be configured to communicate with your cluster. It is recommended to run this tutorial on a cluster with at least two nodes that are not acting as control plane hosts. If you do not already have a cluster, you can create one by using minikube or you can use one of these Kubernetes playgrounds:

- Katacoda
- Play with Kubernetes

To check the version, enter `kubectl version`. </>

Each node in your cluster must have at least 2 GiB of memory.

## Create a namespace

Create a namespace so that the resources you create in this exercise are isolated from the rest of your cluster.

```
kubectl create namespace default-mem-example
```

## Create a LimitRange and a Pod

Here's the configuration file for a LimitRange object. The configuration specifies a default memory request and a default memory limit.

admin/resource/memory-defaults.yaml

```yaml
apiVersion: v1
kind: LimitRange
metadata:
  name: mem-limit-range
spec:
  limits:
  - default:
      memory: 512Mi
    defaultRequest:
      memory: 256Mi
    type: Container
```

Create the LimitRange in the default-mem-example namespace:

```
kubectl apply -f https://k8s.io/examples/admin/resource/memory-defaults.yaml --namespace=default-
```

Now if a Container is created in the default-mem-example namespace, and the Container does not specify its own values for memory request and memory limit, the Container is given a default memory request of 256 MiB and a default memory limit of 512 MiB.

Here's the configuration file for a Pod that has one Container. The Container does not specify a memory request and limit.

admin/resource/memory-defaults-pod.yaml

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: default-mem-demo
spec:
  containers:
  - name: default-mem-demo-ctr
    image: nginx
```

Create the Pod.

```
kubectl apply -f https://k8s.io/examples/admin/resource/memory-defaults-pod.yaml --namespace=defa
```

View detailed information about the Pod:

```
kubectl get pod default-mem-demo --output=yaml --namespace=default-mem-example
```

The output shows that the Pod's Container has a memory request of 256 MiB and a memory limit of 512 MiB. These are the default values specified by the LimitRange.

```
containers:
- image: nginx
  imagePullPolicy: Always
  name: default-mem-demo-ctr
  resources:
    limits:
      memory: 512Mi
    requests:
      memory: 256Mi
```

Delete your Pod:

```
kubectl delete pod default-mem-demo --namespace=default-mem-example
```

## What if you specify a Container's limit, but not its request?

Here's the configuration file for a Pod that has one Container. The Container specifies a memory limit, but not a request:

admin/resource/memory-defaults-pod-2.yaml

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: default-mem-demo-2
spec:
  containers:
  - name: default-mem-demo-2-ctr
    image: nginx
    resources:
      limits:
        memory: "1Gi"
```

Create the Pod:

```
kubectl apply -f https://k8s.io/examples/admin/resource/memory-defaults-pod-2.yaml --namespace=de
```

View detailed information about the Pod:

```
kubectl get pod default-mem-demo-2 --output=yaml --namespace=default-mem-example
```

The output shows that the Container's memory request is set to match its memory limit. Notice that the Container was not assigned the default memory request value of 256Mi.

```
resources:
  limits:
    memory: 1Gi
  requests:
    memory: 1Gi
```

# What if you specify a Container's request, but not its limit?

Here's the configuration file for a Pod that has one Container. The Container specifies a memory request, but not a limit:

admin/resource/memory-defaults-pod-3.yaml

```
apiVersion: v1
kind: Pod
metadata:
  name: default-mem-demo-3
spec:
  containers:
  - name: default-mem-demo-3-ctr
    image: nginx
    resources:
      requests:
        memory: "128Mi"
```

Create the Pod:

```
kubectl apply -f https://k8s.io/examples/admin/resource/memory-defaults-pod-3.yaml --namespace=de
```

View the Pod's specification:

```
kubectl get pod default-mem-demo-3 --output=yaml --namespace=default-mem-example
```

The output shows that the Container's memory request is set to the value specified in the Container's configuration file. The Container's memory limit is set to 512Mi, which is the default memory limit for the namespace.

```
resources:
  limits:
    memory: 512Mi
  requests:
    memory: 128Mi
```

# Motivation for default memory limits and requests

If your namespace has a resource quota, it is helpful to have a default value in place for memory limit. Here are two of the restrictions that a resource quota imposes on a namespace:

- Every Container that runs in the namespace must have its own memory limit.
- The total amount of memory used by all Containers in the namespace must not exceed a specified limit.

If a Container does not specify its own memory limit, it is given the default limit, and then it can be allowed to run in a namespace that is restricted by a quota.

# Clean up

Delete your namespace:

```
kubectl delete namespace default-mem-example
```

# What's next

## For cluster administrators

- Configure Default CPU Requests and Limits for a Namespace
- Configure Minimum and Maximum Memory Constraints for a Namespace
- Configure Minimum and Maximum CPU Constraints for a Namespace
- Configure Memory and CPU Quotas for a Namespace
- Configure a Pod Quota for a Namespace
- Configure Quotas for API Objects

## For app developers

- Assign Memory Resources to Containers and Pods
- Assign CPU Resources to Containers and Pods
- Configure Quality of Service for Pods

# 4.2 - Configure Default CPU Requests and Limits for a Namespace

This page shows how to configure default CPU requests and limits for a namespace. A Kubernetes cluster can be divided into namespaces. If a Container is created in a namespace that has a default CPU limit, and the Container does not specify its own CPU limit, then the Container is assigned the default CPU limit. Kubernetes assigns a default CPU request under certain conditions that are explained later in this topic.

## Before you begin

You need to have a Kubernetes cluster, and the kubectl command-line tool must be configured to communicate with your cluster. It is recommended to run this tutorial on a cluster with at least two nodes that are not acting as control plane hosts. If you do not already have a cluster, you can create one by using minikube or you can use one of these Kubernetes playgrounds:

- Katacoda
- Play with Kubernetes

To check the version, enter `kubectl version`.

## Create a namespace

Create a namespace so that the resources you create in this exercise are isolated from the rest of your cluster.

```
kubectl create namespace default-cpu-example
```

## Create a LimitRange and a Pod

Here's the configuration file for a LimitRange object. The configuration specifies a default CPU request and a default CPU limit.

admin/resource/cpu-defaults.yaml

```yaml
apiVersion: v1
kind: LimitRange
metadata:
  name: cpu-limit-range
spec:
  limits:
  - default:
      cpu: 1
    defaultRequest:
      cpu: 0.5
    type: Container
```

Create the LimitRange in the default-cpu-example namespace:

```
kubectl apply -f https://k8s.io/examples/admin/resource/cpu-defaults.yaml --namespace=default-cpu-
```

Now if a Container is created in the default-cpu-example namespace, and the Container does not specify its own values for CPU request and CPU limit, the Container is given a default CPU request of 0.5 and a default CPU limit of 1.

Here's the configuration file for a Pod that has one Container. The Container does not specify a CPU request and limit.

admin/resource/cpu-defaults-pod.yaml

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: default-cpu-demo
```

```
spec:
  containers:
  - name: default-cpu-demo-ctr
    image: nginx
```

Create the Pod.

```
kubectl apply -f https://k8s.io/examples/admin/resource/cpu-defaults-pod.yaml --namespace=default-
```

View the Pod's specification:

```
kubectl get pod default-cpu-demo --output=yaml --namespace=default-cpu-example
```

The output shows that the Pod's Container has a CPU request of 500 millicpus and a CPU limit of 1 cpu. These are the default values specified by the LimitRange.

```
containers:
- image: nginx
  imagePullPolicy: Always
  name: default-cpu-demo-ctr
  resources:
    limits:
      cpu: "1"
    requests:
      cpu: 500m
```

## What if you specify a Container's limit, but not its request?

Here's the configuration file for a Pod that has one Container. The Container specifies a CPU limit, but not a request:

admin/resource/cpu-defaults-pod-2.yaml

```
apiVersion: v1
kind: Pod
metadata:
  name: default-cpu-demo-2
spec:
  containers:
  - name: default-cpu-demo-2-ctr
    image: nginx
    resources:
      limits:
        cpu: "1"
```

Create the Pod:

```
kubectl apply -f https://k8s.io/examples/admin/resource/cpu-defaults-pod-2.yaml --namespace=defau
```

View the Pod specification:

```
kubectl get pod default-cpu-demo-2 --output=yaml --namespace=default-cpu-example
```

The output shows that the Container's CPU request is set to match its CPU limit. Notice that the Container was not assigned the default CPU request value of 0.5 cpu.

```
resources:
  limits:
    cpu: "1"
  requests:
    cpu: "1"
```

## What if you specify a Container's request, but not its limit?

Here's the configuration file for a Pod that has one Container. The Container specifies a CPU request, but not a limit:

admin/resource/cpu-defaults-pod-3.yaml

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: default-cpu-demo-3
spec:
  containers:
  - name: default-cpu-demo-3-ctr
    image: nginx
    resources:
      requests:
        cpu: "0.75"
```

Create the Pod:

```
kubectl apply -f https://k8s.io/examples/admin/resource/cpu-defaults-pod-3.yaml --namespace=defau
```

View the Pod specification:

```
kubectl get pod default-cpu-demo-3 --output=yaml --namespace=default-cpu-example
```

The output shows that the Container's CPU request is set to the value specified in the Container's configuration file. The Container's CPU limit is set to 1 cpu, which is the default CPU limit for the namespace.

```
resources:
  limits:
    cpu: "1"
  requests:
    cpu: 750m
```

## Motivation for default CPU limits and requests

If your namespace has a resource quota, it is helpful to have a default value in place for CPU limit. Here are two of the restrictions that a resource quota imposes on a namespace:

- Every Container that runs in the namespace must have its own CPU limit.
- The total amount of CPU used by all Containers in the namespace must not exceed a specified limit.

If a Container does not specify its own CPU limit, it is given the default limit, and then it can be allowed to run in a namespace that is restricted by a quota.

## Clean up

Delete your namespace:

```
kubectl delete namespace default-cpu-example
```

# What's next

## For cluster administrators

- Configure Default Memory Requests and Limits for a Namespace
- Configure Minimum and Maximum Memory Constraints for a Namespace
- Configure Minimum and Maximum CPU Constraints for a Namespace
- Configure Memory and CPU Quotas for a Namespace
- Configure a Pod Quota for a Namespace
- Configure Quotas for API Objects

## For app developers

- Assign Memory Resources to Containers and Pods
- Assign CPU Resources to Containers and Pods
- Configure Quality of Service for Pods

# 4.3 - Configure Minimum and Maximum Memory Constraints for a Namespace

This page shows how to set minimum and maximum values for memory used by Containers running in a namespace. You specify minimum and maximum memory values in a LimitRange object. If a Pod does not meet the constraints imposed by the LimitRange, it cannot be created in the namespace.

## Before you begin

You need to have a Kubernetes cluster, and the kubectl command-line tool must be configured to communicate with your cluster. It is recommended to run this tutorial on a cluster with at least two nodes that are not acting as control plane hosts. If you do not already have a cluster, you can create one by using minikube or you can use one of these Kubernetes playgrounds:

- Katacoda
- Play with Kubernetes

To check the version, enter `kubectl version`.

Each node in your cluster must have at least 1 GiB of memory.

## Create a namespace

Create a namespace so that the resources you create in this exercise are isolated from the rest of your cluster.

```
kubectl create namespace constraints-mem-example
```

## Create a LimitRange and a Pod

Here's the configuration file for a LimitRange:

admin/resource/memory-constraints.yaml

```yaml
apiVersion: v1
kind: LimitRange
metadata:
  name: mem-min-max-demo-lr
spec:
  limits:
  - max:
      memory: 1Gi
    min:
      memory: 500Mi
    type: Container
```

Create the LimitRange:

```
kubectl apply -f https://k8s.io/examples/admin/resource/memory-constraints.yaml --namespace=const
```

View detailed information about the LimitRange:

```
kubectl get limitrange mem-min-max-demo-lr --namespace=constraints-mem-example --output=yaml
```

The output shows the minimum and maximum memory constraints as expected. But notice that even though you didn't specify default values in the configuration file for the LimitRange, they were created automatically.

```
limits:
- default:
    memory: 1Gi
  defaultRequest:
    memory: 1Gi
  max:
    memory: 1Gi
  min:
    memory: 500Mi
  type: Container
```

Now whenever a Container is created in the constraints-mem-example namespace, Kubernetes performs these steps:

- If the Container does not specify its own memory request and limit, assign the default memory request and limit to the Container.
- Verify that the Container has a memory request that is greater than or equal to 500 MiB.
- Verify that the Container has a memory limit that is less than or equal to 1 GiB.

Here's the configuration file for a Pod that has one Container. The Container manifest specifies a memory request of 600 MiB and a memory limit of 800 MiB. These satisfy the minimum and maximum memory constraints imposed by the LimitRange.

admin/resource/memory-constraints-pod.yaml

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: constraints-mem-demo
spec:
  containers:
  - name: constraints-mem-demo-ctr
    image: nginx
    resources:
      limits:
        memory: "800Mi"
      requests:
        memory: "600Mi"
```

Create the Pod:

```
kubectl apply -f https://k8s.io/examples/admin/resource/memory-constraints-pod.yaml --namespace=c
```

Verify that the Pod's Container is running:

```
kubectl get pod constraints-mem-demo --namespace=constraints-mem-example
```

View detailed information about the Pod:

```
kubectl get pod constraints-mem-demo --output=yaml --namespace=constraints-mem-example
```

The output shows that the Container has a memory request of 600 MiB and a memory limit of 800 MiB. These satisfy the constraints imposed by the LimitRange.

```yaml
resources:
  limits:
     memory: 800Mi
  requests:
    memory: 600Mi
```

Delete your Pod:

```
kubectl delete pod constraints-mem-demo --namespace=constraints-mem-example
```

## Attempt to create a Pod that exceeds the maximum memory constraint

Here's the configuration file for a Pod that has one Container. The Container specifies a memory request of 800 MiB and a memory limit of 1.5 GiB.

admin/resource/memory-constraints-pod-2.yaml

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: constraints-mem-demo-2
spec:
  containers:
  - name: constraints-mem-demo-2-ctr
    image: nginx
    resources:
      limits:
        memory: "1.5Gi"
      requests:
        memory: "800Mi"
```

Attempt to create the Pod:

```
kubectl apply -f https://k8s.io/examples/admin/resource/memory-constraints-pod-2.yaml --namespace
```

The output shows that the Pod does not get created, because the Container specifies a memory limit that is too large:

```
Error from server (Forbidden): error when creating "examples/admin/resource/memory-constraints-poc
pods "constraints-mem-demo-2" is forbidden: maximum memory usage per Container is 1Gi, but limit
```

## Attempt to create a Pod that does not meet the minimum memory request

Here's the configuration file for a Pod that has one Container. The Container specifies a memory request of 100 MiB and a memory limit of 800 MiB.

admin/resource/memory-constraints-pod-3.yaml

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: constraints-mem-demo-3
spec:
  containers:
  - name: constraints-mem-demo-3-ctr
    image: nginx
    resources:
      limits:
        memory: "800Mi"
      requests:
        memory: "100Mi"
```

Attempt to create the Pod:

```
kubectl apply -f https://k8s.io/examples/admin/resource/memory-constraints-pod-3.yaml --namespace
```

The output shows that the Pod does not get created, because the Container specifies a memory request that is too small:

```
Error from server (Forbidden): error when creating "examples/admin/resource/memory-constraints-poc
pods "constraints-mem-demo-3" is forbidden: minimum memory usage per Container is 500Mi, but reque
```

## Create a Pod that does not specify any memory request or limit

Here's the configuration file for a Pod that has one Container. The Container does not specify a memory request, and it does not specify a memory limit.

admin/resource/memory-constraints-pod-4.yaml

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: constraints-mem-demo-4
spec:
  containers:
  - name: constraints-mem-demo-4-ctr
    image: nginx
```

Create the Pod:

```
kubectl apply -f https://k8s.io/examples/admin/resource/memory-constraints-pod-4.yaml --namespace=
```

View detailed information about the Pod:

```
kubectl get pod constraints-mem-demo-4 --namespace=constraints-mem-example --output=yaml
```

The output shows that the Pod's Container has a memory request of 1 GiB and a memory limit of 1 GiB. How did the Container get those values?

```
resources:
  limits:
    memory: 1Gi
  requests:
    memory: 1Gi
```

Because your Container did not specify its own memory request and limit, it was given the default memory request and limit from the LimitRange.

At this point, your Container might be running or it might not be running. Recall that a prerequisite for this task is that your Nodes have at least 1 GiB of memory. If each of your Nodes has only 1 GiB of memory, then there is not enough allocatable memory on any Node to accommodate a memory request of 1 GiB. If you happen to be using Nodes with 2 GiB of memory, then you probably have enough space to accommodate the 1 GiB request.

Delete your Pod:

```
kubectl delete pod constraints-mem-demo-4 --namespace=constraints-mem-example
```

## Enforcement of minimum and maximum memory constraints

The maximum and minimum memory constraints imposed on a namespace by a LimitRange are enforced only when a Pod is created or updated. If you change the LimitRange, it does not affect Pods that were created previously.

# Motivation for minimum and maximum memory constraints

As a cluster administrator, you might want to impose restrictions on the amount of memory that Pods can use. For example:

- Each Node in a cluster has 2 GB of memory. You do not want to accept any Pod that requests more than 2 GB of memory, because no Node in the cluster can support the request.
- A cluster is shared by your production and development departments. You want to allow production workloads to consume up to 8 GB of memory, but you want development workloads to be limited to 512 MB. You create separate namespaces for production and development, and you apply memory constraints to each namespace.

# Clean up

Delete your namespace:

```
kubectl delete namespace constraints-mem-example
```

# What's next

## For cluster administrators

- Configure Default Memory Requests and Limits for a Namespace
- Configure Default CPU Requests and Limits for a Namespace
- Configure Minimum and Maximum CPU Constraints for a Namespace
- Configure Memory and CPU Quotas for a Namespace
- Configure a Pod Quota for a Namespace
- Configure Quotas for API Objects

## For app developers

- Assign Memory Resources to Containers and Pods
- Assign CPU Resources to Containers and Pods
- Configure Quality of Service for Pods

# 4.4 - Configure Minimum and Maximum CPU Constraints for a Namespace

This page shows how to set minimum and maximum values for the CPU resources used by Containers and Pods in a namespace. You specify minimum and maximum CPU values in a LimitRange object. If a Pod does not meet the constraints imposed by the LimitRange, it cannot be created in the namespace.

## Before you begin

You need to have a Kubernetes cluster, and the kubectl command-line tool must be configured to communicate with your cluster. It is recommended to run this tutorial on a cluster with at least two nodes that are not acting as control plane hosts. If you do not already have a cluster, you can create one by using minikube or you can use one of these Kubernetes playgrounds:

- Katacoda
- Play with Kubernetes

To check the version, enter `kubectl version`.

Your cluster must have at least 1 CPU available for use to run the task examples.

## Create a namespace

Create a namespace so that the resources you create in this exercise are isolated from the rest of your cluster.

```
kubectl create namespace constraints-cpu-example
```

## Create a LimitRange and a Pod

Here's the configuration file for a LimitRange:

admin/resource/cpu-constraints.yaml

```
apiVersion: v1
kind: LimitRange
metadata:
  name: cpu-min-max-demo-lr
spec:
  limits:
  - max:
      cpu: "800m"
    min:
      cpu: "200m"
    type: Container
```

Create the LimitRange:

```
kubectl apply -f https://k8s.io/examples/admin/resource/cpu-constraints.yaml --namespace=constrai
```

View detailed information about the LimitRange:

```
kubectl get limitrange cpu-min-max-demo-lr --output=yaml --namespace=constraints-cpu-example
```

The output shows the minimum and maximum CPU constraints as expected. But notice that even though you didn't specify default values in the configuration file for the LimitRange, they were created automatically.

```
limits:
- default:
    cpu: 800m
  defaultRequest:
    cpu: 800m
```

```
  max:
    cpu: 800m
  min:
    cpu: 200m
  type: Container
```

Now whenever a Container is created in the constraints-cpu-example namespace, Kubernetes performs these steps:

- If the Container does not specify its own CPU request and limit, assign the default CPU request and limit to the Container.
- Verify that the Container specifies a CPU request that is greater than or equal to 200 millicpu.
- Verify that the Container specifies a CPU limit that is less than or equal to 800 millicpu.

> **Note:** When creating a `LimitRange` object, you can specify limits on huge-pages or GPUs as well. However, when both `default` and `defaultRequest` are specified on these resources, the two values must be the same.

Here's the configuration file for a Pod that has one Container. The Container manifest specifies a CPU request of 500 millicpu and a CPU limit of 800 millicpu. These satisfy the minimum and maximum CPU constraints imposed by the LimitRange.

admin/resource/cpu-constraints-pod.yaml

```
apiVersion: v1
kind: Pod
metadata:
  name: constraints-cpu-demo
spec:
  containers:
  - name: constraints-cpu-demo-ctr
    image: nginx
    resources:
      limits:
        cpu: "800m"
      requests:
        cpu: "500m"
```

Create the Pod:

```
kubectl apply -f https://k8s.io/examples/admin/resource/cpu-constraints-pod.yaml --namespace=cons
```

Verify that the Pod's Container is running:

```
kubectl get pod constraints-cpu-demo --namespace=constraints-cpu-example
```

View detailed information about the Pod:

```
kubectl get pod constraints-cpu-demo --output=yaml --namespace=constraints-cpu-example
```

The output shows that the Container has a CPU request of 500 millicpu and CPU limit of 800 millicpu. These satisfy the constraints imposed by the LimitRange.

```
resources:
  limits:
    cpu: 800m
  requests:
    cpu: 500m
```

## Delete the Pod

```
kubectl delete pod constraints-cpu-demo --namespace=constraints-cpu-example
```

## Attempt to create a Pod that exceeds the maximum CPU constraint

Here's the configuration file for a Pod that has one Container. The Container specifies a CPU request of 500 millicpu and a cpu limit of 1.5 cpu.

admin/resource/cpu-constraints-pod-2.yaml

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: constraints-cpu-demo-2
spec:
  containers:
  - name: constraints-cpu-demo-2-ctr
    image: nginx
    resources:
      limits:
        cpu: "1.5"
      requests:
        cpu: "500m"
```

Attempt to create the Pod:

```
kubectl apply -f https://k8s.io/examples/admin/resource/cpu-constraints-pod-2.yaml --namespace=co
```

The output shows that the Pod does not get created, because the Container specifies a CPU limit that is too large:

```
Error from server (Forbidden): error when creating "examples/admin/resource/cpu-constraints-pod-2.
pods "constraints-cpu-demo-2" is forbidden: maximum cpu usage per Container is 800m, but limit is
```

## Attempt to create a Pod that does not meet the minimum CPU request

Here's the configuration file for a Pod that has one Container. The Container specifies a CPU request of 100 millicpu and a CPU limit of 800 millicpu.

admin/resource/cpu-constraints-pod-3.yaml

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: constraints-cpu-demo-3
spec:
  containers:
  - name: constraints-cpu-demo-3-ctr
    image: nginx
    resources:
      limits:
        cpu: "800m"
      requests:
        cpu: "100m"
```

Attempt to create the Pod:

```
kubectl apply -f https://k8s.io/examples/admin/resource/cpu-constraints-pod-3.yaml --namespace=co
```

The output shows that the Pod does not get created, because the Container specifies a CPU request that is too small:

```
Error from server (Forbidden): error when creating "examples/admin/resource/cpu-constraints-pod-3.
pods "constraints-cpu-demo-3" is forbidden: minimum cpu usage per Container is 200m, but request i
```

# Create a Pod that does not specify any CPU request or limit

Here's the configuration file for a Pod that has one Container. The Container does not specify a CPU request, and it does not specify a CPU limit.

admin/resource/cpu-constraints-pod-4.yaml

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: constraints-cpu-demo-4
spec:
  containers:
  - name: constraints-cpu-demo-4-ctr
    image: vish/stress
```

Create the Pod:

```
kubectl apply -f https://k8s.io/examples/admin/resource/cpu-constraints-pod-4.yaml --namespace=co
```

View detailed information about the Pod:

```
kubectl get pod constraints-cpu-demo-4 --namespace=constraints-cpu-example --output=yaml
```

The output shows that the Pod's Container has a CPU request of 800 millicpu and a CPU limit of 800 millicpu. How did the Container get those values?

```yaml
resources:
  limits:
    cpu: 800m
  requests:
    cpu: 800m
```

Because your Container did not specify its own CPU request and limit, it was given the default CPU request and limit from the LimitRange.

At this point, your Container might be running or it might not be running. Recall that a prerequisite for this task is that your cluster must have at least 1 CPU available for use. If each of your Nodes has only 1 CPU, then there might not be enough allocatable CPU on any Node to accommodate a request of 800 millicpu. If you happen to be using Nodes with 2 CPU, then you probably have enough CPU to accommodate the 800 millicpu request.

Delete your Pod:

```
kubectl delete pod constraints-cpu-demo-4 --namespace=constraints-cpu-example
```

# Enforcement of minimum and maximum CPU constraints

The maximum and minimum CPU constraints imposed on a namespace by a LimitRange are enforced only when a Pod is created or updated. If you change the LimitRange, it does not affect Pods that were created previously.

# Motivation for minimum and maximum CPU constraints

As a cluster administrator, you might want to impose restrictions on the CPU resources that Pods can use. For example:

- Each Node in a cluster has 2 CPU. You do not want to accept any Pod that requests more than 2 CPU, because no Node in the cluster can support the request.
- A cluster is shared by your production and development departments. You want to allow production workloads to consume up to 3 CPU, but you want development workloads to be limited to 1 CPU. You create separate namespaces for production and development, and you apply CPU constraints to each namespace.

# Clean up

Delete your namespace:

```
kubectl delete namespace constraints-cpu-example
```

# What's next

## For cluster administrators

- Configure Default Memory Requests and Limits for a Namespace
- Configure Default CPU Requests and Limits for a Namespace
- Configure Minimum and Maximum Memory Constraints for a Namespace
- Configure Memory and CPU Quotas for a Namespace
- Configure a Pod Quota for a Namespace
- Configure Quotas for API Objects

## For app developers

- Assign Memory Resources to Containers and Pods
- Assign CPU Resources to Containers and Pods
- Configure Quality of Service for Pods

# 4.5 - Configure Memory and CPU Quotas for a Namespace

This page shows how to set quotas for the total amount memory and CPU that can be used by all Containers running in a namespace. You specify quotas in a ResourceQuota object.

## Before you begin

You need to have a Kubernetes cluster, and the kubectl command-line tool must be configured to communicate with your cluster. It is recommended to run this tutorial on a cluster with at least two nodes that are not acting as control plane hosts. If you do not already have a cluster, you can create one by using minikube or you can use one of these Kubernetes playgrounds:

- Katacoda
- Play with Kubernetes

To check the version, enter `kubectl version`.

Each node in your cluster must have at least 1 GiB of memory.

## Create a namespace

Create a namespace so that the resources you create in this exercise are isolated from the rest of your cluster.

```
kubectl create namespace quota-mem-cpu-example
```

## Create a ResourceQuota

Here is the configuration file for a ResourceQuota object:

admin/resource/quota-mem-cpu.yaml

```yaml
apiVersion: v1
kind: ResourceQuota
metadata:
  name: mem-cpu-demo
spec:
  hard:
    requests.cpu: "1"
    requests.memory: 1Gi
    limits.cpu: "2"
    limits.memory: 2Gi
```

Create the ResourceQuota:

```
kubectl apply -f https://k8s.io/examples/admin/resource/quota-mem-cpu.yaml --namespace=quota-mem-
```

View detailed information about the ResourceQuota:

```
kubectl get resourcequota mem-cpu-demo --namespace=quota-mem-cpu-example --output=yaml
```

The ResourceQuota places these requirements on the quota-mem-cpu-example namespace:

- Every Container must have a memory request, memory limit, cpu request, and cpu limit.
- The memory request total for all Containers must not exceed 1 GiB.
- The memory limit total for all Containers must not exceed 2 GiB.
- The CPU request total for all Containers must not exceed 1 cpu.
- The CPU limit total for all Containers must not exceed 2 cpu.

# Create a Pod

Here is the configuration file for a Pod:

admin/resource/quota-mem-cpu-pod.yaml

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: quota-mem-cpu-demo
spec:
  containers:
  - name: quota-mem-cpu-demo-ctr
    image: nginx
    resources:
      limits:
        memory: "800Mi"
        cpu: "800m"
      requests:
        memory: "600Mi"
        cpu: "400m"
```

Create the Pod:

```
kubectl apply -f https://k8s.io/examples/admin/resource/quota-mem-cpu-pod.yaml --namespace=quota-
```

Verify that the Pod's Container is running:

```
kubectl get pod quota-mem-cpu-demo --namespace=quota-mem-cpu-example
```

Once again, view detailed information about the ResourceQuota:

```
kubectl get resourcequota mem-cpu-demo --namespace=quota-mem-cpu-example --output=yaml
```

The output shows the quota along with how much of the quota has been used. You can see that the memory and CPU requests and limits for your Pod do not exceed the quota.

```
status:
  hard:
    limits.cpu: "2"
    limits.memory: 2Gi
    requests.cpu: "1"
    requests.memory: 1Gi
  used:
    limits.cpu: 800m
    limits.memory: 800Mi
    requests.cpu: 400m
    requests.memory: 600Mi
```

# Attempt to create a second Pod

Here is the configuration file for a second Pod:

admin/resource/quota-mem-cpu-pod-2.yaml

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: quota-mem-cpu-demo-2
spec:
  containers:
  - name: quota-mem-cpu-demo-2-ctr
    image: redis
    resources:
      limits:
        memory: "1Gi"
        cpu: "800m"
```

```
    requests:
      memory: "700Mi"
      cpu: "400m"
```

In the configuration file, you can see that the Pod has a memory request of 700 MiB. Notice that the sum of the used memory request and this new memory request exceeds the memory request quota. 600 MiB + 700 MiB > 1 GiB.

Attempt to create the Pod:

```
kubectl apply -f https://k8s.io/examples/admin/resource/quota-mem-cpu-pod-2.yaml --namespace=quot
```

The second Pod does not get created. The output shows that creating the second Pod would cause the memory request total to exceed the memory request quota.

```
Error from server (Forbidden): error when creating "examples/admin/resource/quota-mem-cpu-pod-2.ya
pods "quota-mem-cpu-demo-2" is forbidden: exceeded quota: mem-cpu-demo,
requested: requests.memory=700Mi,used: requests.memory=600Mi, limited: requests.memory=1Gi
```

# Discussion

As you have seen in this exercise, you can use a ResourceQuota to restrict the memory request total for all Containers running in a namespace. You can also restrict the totals for memory limit, cpu request, and cpu limit.

If you want to restrict individual Containers, instead of totals for all Containers, use a LimitRange.

# Clean up

Delete your namespace:

```
kubectl delete namespace quota-mem-cpu-example
```

# What's next

## For cluster administrators

- Configure Default Memory Requests and Limits for a Namespace
- Configure Default CPU Requests and Limits for a Namespace
- Configure Minimum and Maximum Memory Constraints for a Namespace
- Configure Minimum and Maximum CPU Constraints for a Namespace
- Configure a Pod Quota for a Namespace
- Configure Quotas for API Objects

## For app developers

- Assign Memory Resources to Containers and Pods
- Assign CPU Resources to Containers and Pods
- Configure Quality of Service for Pods

# 4.6 - Configure a Pod Quota for a Namespace

This page shows how to set a quota for the total number of Pods that can run in a namespace. You specify quotas in a ResourceQuota object.

## Before you begin

You need to have a Kubernetes cluster, and the kubectl command-line tool must be configured to communicate with your cluster. It is recommended to run this tutorial on a cluster with at least two nodes that are not acting as control plane hosts. If you do not already have a cluster, you can create one by using minikube or you can use one of these Kubernetes playgrounds:

- Katacoda
- Play with Kubernetes

To check the version, enter `kubectl version`.

## Create a namespace

Create a namespace so that the resources you create in this exercise are isolated from the rest of your cluster.

```
kubectl create namespace quota-pod-example
```

## Create a ResourceQuota

Here is the configuration file for a ResourceQuota object:

admin/resource/quota-pod.yaml

```yaml
apiVersion: v1
kind: ResourceQuota
metadata:
  name: pod-demo
spec:
  hard:
    pods: "2"
```

Create the ResourceQuota:

```
kubectl apply -f https://k8s.io/examples/admin/resource/quota-pod.yaml --namespace=quota-pod-exam
```

View detailed information about the ResourceQuota:

```
kubectl get resourcequota pod-demo --namespace=quota-pod-example --output=yaml
```

The output shows that the namespace has a quota of two Pods, and that currently there are no Pods; that is, none of the quota is used.

```yaml
spec:
  hard:
    pods: "2"
status:
  hard:
    pods: "2"
  used:
    pods: "0"
```

Here is the configuration file for a Deployment:

admin/resource/quota-pod-deployment.yaml

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: pod-quota-demo
spec:
  selector:
    matchLabels:
      purpose: quota-demo
  replicas: 3
  template:
    metadata:
      labels:
        purpose: quota-demo
    spec:
      containers:
      - name: pod-quota-demo
        image: nginx
```

In the configuration file, `replicas: 3` tells Kubernetes to attempt to create three Pods, all running the same application.

Create the Deployment:

```
kubectl apply -f https://k8s.io/examples/admin/resource/quota-pod-deployment.yaml --namespace=quo
```

View detailed information about the Deployment:

```
kubectl get deployment pod-quota-demo --namespace=quota-pod-example --output=yaml
```

The output shows that even though the Deployment specifies three replicas, only two Pods were created because of the quota.

```yaml
spec:
  ...
  replicas: 3
...
status:
  availableReplicas: 2
...
lastUpdateTime: 2017-07-07T20:57:05Z
    message: 'unable to create pods: pods "pod-quota-demo-1650323038-" is forbidden:
      exceeded quota: pod-demo, requested: pods=1, used: pods=2, limited: pods=2'
```

# Clean up

Delete your namespace:

```
kubectl delete namespace quota-pod-example
```

# What's next

## For cluster administrators

- Configure Default Memory Requests and Limits for a Namespace
- Configure Default CPU Requests and Limits for a Namespace
- Configure Minimum and Maximum Memory Constraints for a Namespace
- Configure Minimum and Maximum CPU Constraints for a Namespace
- Configure Memory and CPU Quotas for a Namespace
- Configure Quotas for API Objects

## For app developers

- Assign Memory Resources to Containers and Pods
- Assign CPU Resources to Containers and Pods
- Configure Quality of Service for Pods

</>

</>

</>

</>

# 5 - Install a Network Policy Provider

# 5.1 - Use Antrea for NetworkPolicy

This page shows how to install and use Antrea CNI plugin on Kubernetes. For background on Project Antrea, read the Introduction to Antrea.

## Before you begin

You need to have a Kubernetes cluster. Follow the kubeadm getting started guide to bootstrap one.

## Deploying Antrea with kubeadm

Follow Getting Started guide to deploy Antrea for kubeadm.

## What's next

Once your cluster is running, you can follow the Declare Network Policy to try out Kubernetes NetworkPolicy.

# 5.2 - Use Calico for NetworkPolicy

This page shows a couple of quick ways to create a Calico cluster on Kubernetes.

## Before you begin

Decide whether you want to deploy a cloud or local cluster.

## Creating a Calico cluster with Google Kubernetes Engine (GKE)

**Prerequisite**: gcloud.

1. To launch a GKE cluster with Calico, include the `--enable-network-policy` flag.

   **Syntax**

   ```
   gcloud container clusters create [CLUSTER_NAME] --enable-network-policy
   ```

   **Example**

   ```
   gcloud container clusters create my-calico-cluster --enable-network-policy
   ```

2. To verify the deployment, use the following command.

   ```
   kubectl get pods --namespace=kube-system
   ```

   The Calico pods begin with `calico`. Check to make sure each one has a status of `Running`.

## Creating a local Calico cluster with kubeadm

To get a local single-host Calico cluster in fifteen minutes using kubeadm, refer to the Calico Quickstart.

## What's next

Once your cluster is running, you can follow the Declare Network Policy to try out Kubernetes NetworkPolicy.

# 5.3 - Use Cilium for NetworkPolicy

This page shows how to use Cilium for NetworkPolicy.

For background on Cilium, read the Introduction to Cilium.

## Before you begin

You need to have a Kubernetes cluster, and the kubectl command-line tool must be configured to communicate with your cluster. It is recommended to run this tutorial on a cluster with at least two nodes that are not acting as control plane hosts. If you do not already have a cluster, you can create one by using minikube or you can use one of these Kubernetes playgrounds:

- Katacoda
- Play with Kubernetes

To check the version, enter `kubectl version`.

## Deploying Cilium on Minikube for Basic Testing

To get familiar with Cilium easily you can follow the Cilium Kubernetes Getting Started Guide to perform a basic DaemonSet installation of Cilium in minikube.

To start minikube, minimal version required is >= v1.3.1, run the with the following arguments:

```
minikube version
```

```
minikube version: v1.3.1
```

```
minikube start --network-plugin=cni --memory=4096
```

Mount the BPF filesystem:

```
minikube ssh -- sudo mount bpffs -t bpf /sys/fs/bpf
```

For minikube you can deploy this simple "all-in-one" YAML file that includes DaemonSet configurations for Cilium as well as appropriate RBAC settings:

```
kubectl create -f https://raw.githubusercontent.com/cilium/cilium/v1.8/install/kubernetes/quick-i
```

```
configmap/cilium-config created
serviceaccount/cilium created
serviceaccount/cilium-operator created
clusterrole.rbac.authorization.k8s.io/cilium created
clusterrole.rbac.authorization.k8s.io/cilium-operator created
clusterrolebinding.rbac.authorization.k8s.io/cilium created
clusterrolebinding.rbac.authorization.k8s.io/cilium-operator created
daemonset.apps/cilium create
deployment.apps/cilium-operator created
```

The remainder of the Getting Started Guide explains how to enforce both L3/L4 (i.e., IP address + port) security policies, as well as L7 (e.g., HTTP) security policies using an example application.

## Deploying Cilium for Production Use

For detailed instructions around deploying Cilium for production, see: Cilium Kubernetes Installation Guide This documentation includes detailed requirements, instructions and example production DaemonSet files.

## Understanding Cilium components

Deploying a cluster with Cilium adds Pods to the `kube-system` namespace. To see this list of Pods run:

```
kubectl get pods --namespace=kube-system
```

You'll see a list of Pods similar to this:

```
NAME            READY   STATUS    RESTARTS   AGE
cilium-6rxbd    1/1     Running   0          1m
...
```

A `cilium` Pod runs on each node in your cluster and enforces network policy on the traffic to/from Pods on that node using Linux BPF.

# What's next

Once your cluster is running, you can follow the Declare Network Policy to try out Kubernetes NetworkPolicy with Cilium. Have fun, and if you have questions, contact us using the Cilium Slack Channel.

# 5.4 - Use Kube-router for NetworkPolicy

This page shows how to use Kube-router for NetworkPolicy.

## Before you begin

You need to have a Kubernetes cluster running. If you do not already have a cluster, you can create one by using any of the cluster installers like Kops, Bootkube, Kubeadm etc.

## Installing Kube-router addon

The Kube-router Addon comes with a Network Policy Controller that watches Kubernetes API server for any NetworkPolicy and pods updated and configures iptables rules and ipsets to allow or block traffic as directed by the policies. Please follow the trying Kube-router with cluster installers guide to install Kube-router addon.

## What's next

Once you have installed the Kube-router addon, you can follow the Declare Network Policy to try out Kubernetes NetworkPolicy.

# 5.5 - Romana for NetworkPolicy

This page shows how to use Romana for NetworkPolicy.

## Before you begin

Complete steps 1, 2, and 3 of the kubeadm getting started guide.

## Installing Romana with kubeadm

Follow the containerized installation guide for kubeadm.

</>

## Applying network policies

To apply network policies use one of the following:

- Romana network policies.
  - Example of Romana network policy.
- The NetworkPolicy API.

## What's next

Once you have installed Romana, you can follow the Declare Network Policy to try out Kubernetes NetworkPolicy.

</>

</>

# 5.6 - Weave Net for NetworkPolicy

This page shows how to use Weave Net for NetworkPolicy.

## Before you begin

You need to have a Kubernetes cluster. Follow the kubeadm getting started guide to bootstrap one.

## Install the Weave Net addon

Follow the Integrating Kubernetes via the Addon guide.

The Weave Net addon for Kubernetes comes with a Network Policy Controller that automatically monitors Kubernetes for any NetworkPolicy annotations on all namespaces and configures `iptables` rules to allow or block traffic as directed by the policies.

## Test the installation

Verify that the weave works.

Enter the following command:

```
kubectl get pods -n kube-system -o wide
```

The output is similar to this:

```
NAME                              READY     STATUS    RESTARTS   AGE       IP              N
weave-net-1t1qg                   2/2       Running   0          9d        192.168.2.10    w
weave-net-231d7                   2/2       Running   1          7d        10.2.0.17       w
weave-net-7nmwt                   2/2       Running   3          9d        192.168.2.131   n
weave-net-pmw8w                   2/2       Running   0          9d        192.168.2.216   w
```

Each Node has a weave Pod, and all Pods are `Running` and `2/2 READY`. (`2/2` means that each Pod has `weave` and `weave-npc`.)

## What's next

Once you have installed the Weave Net addon, you can follow the Declare Network Policy to try out Kubernetes NetworkPolicy. If you have any question, contact us at #weave-community on Slack or Weave User Group.

# 6 - Access Clusters Using the Kubernetes API

This page shows how to access clusters using the Kubernetes API.

## Before you begin

You need to have a Kubernetes cluster, and the kubectl command-line tool must be configured to communicate with your cluster. It is recommended to run this tutorial on a cluster with at least two nodes that are not acting as control plane hosts. If you do not already have a cluster, you can create one by using minikube or you can use one of these Kubernetes playgrounds:

- Katacoda
- Play with Kubernetes

To check the version, enter `kubectl version`.

## Accessing the Kubernetes API

### Accessing for the first time with kubectl

When accessing the Kubernetes API for the first time, use the Kubernetes command-line tool, `kubectl`.

To access a cluster, you need to know the location of the cluster and have credentials to access it. Typically, this is automatically set-up when you work through a Getting started guide, or someone else setup the cluster and provided you with credentials and a location.

Check the location and credentials that kubectl knows about with this command:

```
kubectl config view
```

Many of the examples provide an introduction to using kubectl. Complete documentation is found in the kubectl manual.

### Directly accessing the REST API

kubectl handles locating and authenticating to the API server. If you want to directly access the REST API with an http client like `curl` or `wget`, or a browser, there are multiple ways you can locate and authenticate against the API server:

1. Run kubectl in proxy mode (recommended). This method is recommended, since it uses the stored apiserver location and verifies the identity of the API server using a self-signed cert. No man-in-the-middle (MITM) attack is possible using this method.
2. Alternatively, you can provide the location and credentials directly to the http client. This works with client code that is confused by proxies. To protect against man in the middle attacks, you'll need to import a root cert into your browser.

Using the Go or Python client libraries provides accessing kubectl in proxy mode.

#### Using kubectl proxy

The following command runs kubectl in a mode where it acts as a reverse proxy. It handles locating the API server and authenticating.

Run it like this:

```
kubectl proxy --port=8080 &
```

See kubectl proxy for more details.

Then you can explore the API with curl, wget, or a browser, like so:

```
curl http://localhost:8080/api/
```

The output is similar to this:

```
{
```

```
  "versions": [
    "v1"
  ],
  "serverAddressByClientCIDRs": [
    {
      "clientCIDR": "0.0.0.0/0",
      "serverAddress": "10.0.1.149:443"
    }
  ]
}
```

## Without kubectl proxy

It is possible to avoid using kubectl proxy by passing an authentication token directly to the API server, like this:

Using `grep/cut` approach:

```
# Check all possible clusters, as your .KUBECONFIG may have multiple contexts:
kubectl config view -o jsonpath='{"Cluster name\tServer\n"}{range .clusters[*]}{.name}{"\t"}{.clu

# Select name of cluster you want to interact with from above output:
export CLUSTER_NAME="some_server_name"

# Point to the API server referring the cluster name
APISERVER=$(kubectl config view -o jsonpath="{.clusters[?(@.name==\"$CLUSTER_NAME\")].cluster.ser

# Gets the token value
TOKEN=$(kubectl get secrets -o jsonpath="{.items[?(@.metadata.annotations['kubernetes\.io/service

# Explore the API with TOKEN
curl -X GET $APISERVER/api --header "Authorization: Bearer $TOKEN" --insecure
```

The output is similar to this:

```
{
  "kind": "APIVersions",
  "versions": [
    "v1"
  ],
  "serverAddressByClientCIDRs": [
    {
      "clientCIDR": "0.0.0.0/0",
      "serverAddress": "10.0.1.149:443"
    }
  ]
}
```

Using `jsonpath` approach:

```
APISERVER=$(kubectl config view --minify -o jsonpath='{.clusters[0].cluster.server}')
TOKEN=$(kubectl get secret $(kubectl get serviceaccount default -o jsonpath='{.secrets[0].name}')
curl $APISERVER/api --header "Authorization: Bearer $TOKEN" --insecure
{
  "kind": "APIVersions",
  "versions": [
    "v1"
  ],
  "serverAddressByClientCIDRs": [
    {
      "clientCIDR": "0.0.0.0/0",
      "serverAddress": "10.0.1.149:443"
    }
  ]
}
```

The above example uses the `--insecure` flag. This leaves it subject to MITM attacks. When kubectl accesses the cluster it uses a stored root certificate and client certificates to access the server. (These are installed in the `~/.kube` directory). Since cluster certificates are typically self-signed, it may take special configuration to get your http client to use root certificate.

On some clusters, the API server does not require authentication; it may serve on localhost, or be protected by a firewall. There is not a standard for this. Controlling Access to the Kubernetes API describes how you can configure this as a cluster administrator.

## Programmatic access to the API

Kubernetes officially supports client libraries for Go, Python, Java, dotnet, Javascript, and Haskell. There are other client libraries that are provided and maintained by their authors, not the Kubernetes team. See client libraries for accessing the API from other languages and how they authenticate.

### Go client

- To get the library, run the following command: `go get k8s.io/client-go@kubernetes-<kubernetes-version-number>` See https://github.com/kubernetes/client-go/releases to see which versions are supported.
- Write an application atop of the client-go clients.

> **Note:** client-go defines its own API objects, so if needed, import API definitions from client-go rather than from the main repository. For example, `import "k8s.io/client-go/kubernetes"` is correct.

The Go client can use the same kubeconfig file as the kubectl CLI does to locate and authenticate to the API server. See this example:

```go
package main

import (
  "context"
  "fmt"
  "k8s.io/apimachinery/pkg/apis/meta/v1"
  "k8s.io/client-go/kubernetes"
  "k8s.io/client-go/tools/clientcmd"
)

func main() {
  // uses the current context in kubeconfig
  // path-to-kubeconfig -- for example, /root/.kube/config
  config, _ := clientcmd.BuildConfigFromFlags("", "<path-to-kubeconfig>")
  // creates the clientset
  clientset, _ := kubernetes.NewForConfig(config)
  // access the API to list pods
  pods, _ := clientset.CoreV1().Pods("").List(context.TODO(), v1.ListOptions{})
  fmt.Printf("There are %d pods in the cluster\n", len(pods.Items))
}
```

If the application is deployed as a Pod in the cluster, see Accessing the API from within a Pod.

### Python client

To use Python client, run the following command: `pip install kubernetes` See Python Client Library page for more installation options.

The Python client can use the same kubeconfig file as the kubectl CLI does to locate and authenticate to the API server. See this example:

```python
from kubernetes import client, config

config.load_kube_config()

v1=client.CoreV1Api()
print("Listing pods with their IPs:")
ret = v1.list_pod_for_all_namespaces(watch=False)
for i in ret.items:
    print("%s\t%s\t%s" % (i.status.pod_ip, i.metadata.namespace, i.metadata.name))
```

### Java client

To install the Java Client, run:

```
# Clone java library
git clone --recursive https://github.com/kubernetes-client/java

# Installing project artifacts, POM etc:
cd java
mvn install
```

See https://github.com/kubernetes-client/java/releases to see which versions are supported.

The Java client can use the same kubeconfig file as the kubectl CLI does to locate and authenticate to the API server. See this example:

```
package io.kubernetes.client.examples;

import io.kubernetes.client.ApiClient;
import io.kubernetes.client.ApiException;
import io.kubernetes.client.Configuration;
import io.kubernetes.client.apis.CoreV1Api;
import io.kubernetes.client.models.V1Pod;
import io.kubernetes.client.models.V1PodList;
import io.kubernetes.client.util.ClientBuilder;
import io.kubernetes.client.util.KubeConfig;
import java.io.FileReader;
import java.io.IOException;

/**
 * A simple example of how to use the Java API from an application outside a kubernetes cluster
 *
 * <p>Easiest way to run this: mvn exec:java
 * -Dexec.mainClass="io.kubernetes.client.examples.KubeConfigFileClientExample"
 *
 */
public class KubeConfigFileClientExample {
  public static void main(String[] args) throws IOException, ApiException {

    // file path to your KubeConfig
    String kubeConfigPath = "~/.kube/config";

    // loading the out-of-cluster config, a kubeconfig from file-system
    ApiClient client =
        ClientBuilder.kubeconfig(KubeConfig.loadKubeConfig(new FileReader(kubeConfigPath))).build

    // set the global default api-client to the in-cluster one from above
    Configuration.setDefaultApiClient(client);

    // the CoreV1Api loads default api-client from global configuration.
    CoreV1Api api = new CoreV1Api();

    // invokes the CoreV1Api client
    V1PodList list = api.listPodForAllNamespaces(null, null, null, null, null, null, null, null,
    System.out.println("Listing all pods: ");
    for (V1Pod item : list.getItems()) {
      System.out.println(item.getMetadata().getName());
    }
  }
}
```

## dotnet client

To use dotnet client, run the following command: `dotnet add package KubernetesClient --version 1.6.1` See dotnet Client Library page for more installation options. See https://github.com/kubernetes-client/csharp/releases to see which versions are supported.

The dotnet client can use the same kubeconfig file as the kubectl CLI does to locate and authenticate to the API server. See this example:

```
using System;
using k8s;

namespace simple
{
    internal class PodList
    {
        private static void Main(string[] args)
        {
            var config = KubernetesClientConfiguration.BuildDefaultConfig();
            IKubernetes client = new Kubernetes(config);
            Console.WriteLine("Starting Request!");

            var list = client.ListNamespacedPod("default");
            foreach (var item in list.Items)
            {
                Console.WriteLine(item.Metadata.Name);
            }
            if (list.Items.Count == 0)
            {
                Console.WriteLine("Empty!");
            }
```

```
        }
    }
}
```

### JavaScript client

To install JavaScript client, run the following command: `npm install @kubernetes/client-node`. See https://github.com/kubernetes-client/javascript/releases to see which versions are supported.

The JavaScript client can use the same kubeconfig file as the kubectl CLI does to locate and authenticate to the API server. See this example:

```
const k8s = require('@kubernetes/client-node');

const kc = new k8s.KubeConfig();
kc.loadFromDefault();

const k8sApi = kc.makeApiClient(k8s.CoreV1Api);

k8sApi.listNamespacedPod('default').then((res) => {
    console.log(res.body);
});
```

### Haskell client

See https://github.com/kubernetes-client/haskell/releases to see which versions are supported.

The Haskell client can use the same kubeconfig file as the kubectl CLI does to locate and authenticate to the API server. See this example:

```
exampleWithKubeConfig :: IO ()
exampleWithKubeConfig = do
    oidcCache <- atomically $ newTVar $ Map.fromList []
    (mgr, kcfg) <- mkKubeClientConfig oidcCache $ KubeConfigFile "/path/to/kubeconfig"
    dispatchMime
            mgr
            kcfg
            (CoreV1.listPodForAllNamespaces (Accept MimeJSON))
        >>= print
```

## What's next

- Accessing the Kubernetes API from a Pod

# 7 - Access Services Running on Clusters

This page shows how to connect to services running on the Kubernetes cluster.

## Before you begin

You need to have a Kubernetes cluster, and the kubectl command-line tool must be configured to communicate with your cluster. It is recommended to run this tutorial on a cluster with at least two nodes that are not acting as control plane hosts. If you do not already have a cluster, you can create one by using minikube or you can use one of these Kubernetes playgrounds:

- Katacoda
- Play with Kubernetes

To check the version, enter `kubectl version`.

## Accessing services running on the cluster

In Kubernetes, nodes, pods and services all have their own IPs. In many cases, the node IPs, pod IPs, and some service IPs on a cluster will not be routable, so they will not be reachable from a machine outside the cluster, such as your desktop machine.

### Ways to connect

You have several options for connecting to nodes, pods and services from outside the cluster:

- Access services through public IPs.
  - Use a service with type `NodePort` or `LoadBalancer` to make the service reachable outside the cluster. See the services and kubectl expose documentation.
  - Depending on your cluster environment, this may only expose the service to your corporate network, or it may expose it to the internet. Think about whether the service being exposed is secure. Does it do its own authentication?
  - Place pods behind services. To access one specific pod from a set of replicas, such as for debugging, place a unique label on the pod and create a new service which selects this label.
  - In most cases, it should not be necessary for application developer to directly access nodes via their nodeIPs.
- Access services, nodes, or pods using the Proxy Verb.
  - Does apiserver authentication and authorization prior to accessing the remote service. Use this if the services are not secure enough to expose to the internet, or to gain access to ports on the node IP, or for debugging.
  - Proxies may cause problems for some web applications.
  - Only works for HTTP/HTTPS.
  - Described here.
- Access from a node or pod in the cluster.
  - Run a pod, and then connect to a shell in it using kubectl exec. Connect to other nodes, pods, and services from that shell.
  - Some clusters may allow you to ssh to a node in the cluster. From there you may be able to access cluster services. This is a non-standard method, and will work on some clusters but not others. Browsers and other tools may or may not be installed. Cluster DNS may not work.

### Discovering builtin services

Typically, there are several services which are started on a cluster by kube-system. Get a list of these with the `kubectl cluster-info` command:

```
kubectl cluster-info
```

The output is similar to this:

```
Kubernetes master is running at https://104.197.5.247
elasticsearch-logging is running at https://104.197.5.247/api/v1/namespaces/kube-system/services/e
kibana-logging is running at https://104.197.5.247/api/v1/namespaces/kube-system/services/kibana-l
kube-dns is running at https://104.197.5.247/api/v1/namespaces/kube-system/services/kube-dns/proxy
grafana is running at https://104.197.5.247/api/v1/namespaces/kube-system/services/monitoring-grat
heapster is running at https://104.197.5.247/api/v1/namespaces/kube-system/services/monitoring-hea
```

This shows the proxy-verb URL for accessing each service. For example, this cluster has cluster-level logging enabled (using Elasticsearch), which can be reached at `https://104.197.5.247/api/v1/namespaces/kube-system/services/elasticsearch-logging/proxy/` if suitable credentials are passed, or through a kubectl proxy at, for example: `http://localhost:8080/api/v1/namespaces/kube-system/services/elasticsearch-logging/proxy/`.

> **Note:** See Access Clusters Using the Kubernetes API for how to pass credentials or use kubectl proxy.

## Manually constructing apiserver proxy URLs

As mentioned above, you use the `kubectl cluster-info` command to retrieve the service's proxy URL. To create proxy URLs that include service endpoints, suffixes, and parameters, you append to the service's proxy URL: `http:// kubernetes_master_address /api/v1/namespaces/ namespace_name /services/ [https:]service_name [:port_name] /proxy`

If you haven't specified a name for your port, you don't have to specify *port_name* in the URL.

### Examples

- To access the Elasticsearch service endpoint `_search?q=user:kimchy`, you would use:

```
http://104.197.5.247/api/v1/namespaces/kube-system/services/elasticsearch-logging/proxy/_sear
```

- To access the Elasticsearch cluster health information `_cluster/health?pretty=true`, you would use:

```
https://104.197.5.247/api/v1/namespaces/kube-system/services/elasticsearch-logging/proxy/_clu
```

  The health information is similar to this:

```
{
  "cluster_name" : "kubernetes_logging",
  "status" : "yellow",
  "timed_out" : false,
  "number_of_nodes" : 1,
  "number_of_data_nodes" : 1,
  "active_primary_shards" : 5,
  "active_shards" : 5,
  "relocating_shards" : 0,
  "initializing_shards" : 0,
  "unassigned_shards" : 5
}
```

- To access the *https* Elasticsearch service health information `_cluster/health?pretty=true`, you would use:

```
https://104.197.5.247/api/v1/namespaces/kube-system/services/https:elasticsearch-logging/prox
```

## Using web browsers to access services running on the cluster

You may be able to put an apiserver proxy URL into the address bar of a browser. However:

- Web browsers cannot usually pass tokens, so you may need to use basic (password) auth. Apiserver can be configured to accept basic auth, but your cluster may not be configured to accept basic auth.
- Some web apps may not work, particularly those with client side javascript that construct URLs in a way that is unaware of the proxy path prefix.

# 8 - Advertise Extended Resources for a Node

This page shows how to specify extended resources for a Node. Extended resources allow cluster administrators to advertise node-level resources that would otherwise be unknown to Kubernetes.

## Before you begin

You need to have a Kubernetes cluster, and the kubectl command-line tool must be configured to communicate with your cluster. It is recommended to run this tutorial on a cluster with at least two nodes that are not acting as control plane hosts. If you do not already have a cluster, you can create one by using minikube or you can use one of these Kubernetes playgrounds:

- Katacoda
- Play with Kubernetes

To check the version, enter `kubectl version`.

## Get the names of your Nodes

```
kubectl get nodes
```

Choose one of your Nodes to use for this exercise.

## Advertise a new extended resource on one of your Nodes

To advertise a new extended resource on a Node, send an HTTP PATCH request to the Kubernetes API server. For example, suppose one of your Nodes has four dongles attached. Here's an example of a PATCH request that advertises four dongle resources for your Node.

```
PATCH /api/v1/nodes/<your-node-name>/status HTTP/1.1
Accept: application/json
Content-Type: application/json-patch+json
Host: k8s-master:8080

[
  {
    "op": "add",
    "path": "/status/capacity/example.com~1dongle",
    "value": "4"
  }
]
```

Note that Kubernetes does not need to know what a dongle is or what a dongle is for. The preceding PATCH request tells Kubernetes that your Node has four things that you call dongles.

Start a proxy, so that you can easily send requests to the Kubernetes API server:

```
kubectl proxy
```

In another command window, send the HTTP PATCH request. Replace `<your-node-name>` with the name of your Node:

```
curl --header "Content-Type: application/json-patch+json" \
--request PATCH \
--data '[{"op": "add", "path": "/status/capacity/example.com~1dongle", "value": "4"}]' \
http://localhost:8001/api/v1/nodes/<your-node-name>/status
```

> **Note:** In the preceding request, `~1` is the encoding for the character / in the patch path. The operation path value in JSON-Patch is interpreted as a JSON-Pointer. For more details, see IETF RFC 6901, section 3.

The output shows that the Node has a capacity of 4 dongles:

```
"capacity": {
  "cpu": "2",
  "memory": "2049008Ki",
  "example.com/dongle": "4",
```

Describe your Node:

```
kubectl describe node <your-node-name>
```

Once again, the output shows the dongle resource:

```
Capacity:
 cpu:  2
 memory:  2049008Ki
 example.com/dongle:  4
```

Now, application developers can create Pods that request a certain number of dongles. See Assign Extended Resources to a Container.

# Discussion

Extended resources are similar to memory and CPU resources. For example, just as a Node has a certain amount of memory and CPU to be shared by all components running on the Node, it can have a certain number of dongles to be shared by all components running on the Node. And just as application developers can create Pods that request a certain amount of memory and CPU, they can create Pods that request a certain number of dongles.

Extended resources are opaque to Kubernetes; Kubernetes does not know anything about what they are. Kubernetes knows only that a Node has a certain number of them. Extended resources must be advertised in integer amounts. For example, a Node can advertise four dongles, but not 4.5 dongles.

## Storage example

Suppose a Node has 800 GiB of a special kind of disk storage. You could create a name for the special storage, say example.com/special-storage. Then you could advertise it in chunks of a certain size, say 100 GiB. In that case, your Node would advertise that it has eight resources of type example.com/special-storage.

```
Capacity:
 ...
 example.com/special-storage: 8
```

If you want to allow arbitrary requests for special storage, you could advertise special storage in chunks of size 1 byte. In that case, you would advertise 800Gi resources of type example.com/special-storage.

```
Capacity:
 ...
 example.com/special-storage:  800Gi
```

Then a Container could request any number of bytes of special storage, up to 800Gi.

# Clean up

</>

Here is a PATCH request that removes the dongle advertisement from a Node.

```
PATCH /api/v1/nodes/<your-node-name>/status HTTP/1.1
Accept: application/json
Content-Type: application/json-patch+json
Host: k8s-master:8080

[
  {
    "op": "remove",
    "path": "/status/capacity/example.com~1dongle",
  }
]
```

Start a proxy, so that you can easily send requests to the Kubernetes API server:

```
kubectl proxy
```

In another command window, send the HTTP PATCH request. Replace `<your-node-name>` with the name of your Node:

```
curl --header "Content-Type: application/json-patch+json" \
--request PATCH \
--data '[{"op": "remove", "path": "/status/capacity/example.com~1dongle"}]' \
http://localhost:8001/api/v1/nodes/<your-node-name>/status
```

Verify that the dongle advertisement has been removed:

```
kubectl describe node <your-node-name> | grep dongle
```

(you should not see any output)

# What's next

## For application developers

- Assign Extended Resources to a Container

## For cluster administrators

- Configure Minimum and Maximum Memory Constraints for a Namespace
- Configure Minimum and Maximum CPU Constraints for a Namespace

# 9 - Autoscale the DNS Service in a Cluster

This page shows how to enable and configure autoscaling of the DNS service in your Kubernetes cluster.

## Before you begin

- You need to have a Kubernetes cluster, and the kubectl command-line tool must be configured to communicate with your cluster. It is recommended to run this tutorial on a cluster with at least two nodes that are not acting as control plane hosts. If you do not already have a cluster, you can create one by using minikube or you can use one of these Kubernetes playgrounds:
  - Katacoda
  - Play with Kubernetes

  To check the version, enter `kubectl version`.
- This guide assumes your nodes use the AMD64 or Intel 64 CPU architecture.
- Make sure Kubernetes DNS is enabled.

## Determine whether DNS horizontal autoscaling is already enabled

List the Deployments in your cluster in the kube-system namespace:

```
kubectl get deployment --namespace=kube-system
```

The output is similar to this:

```
NAME                  READY   UP-TO-DATE   AVAILABLE   AGE
...
dns-autoscaler        1/1     1            1           ...
...
```

If you see "dns-autoscaler" in the output, DNS horizontal autoscaling is already enabled, and you can skip to Tuning autoscaling parameters.

## Get the name of your DNS Deployment

List the DNS deployments in your cluster in the kube-system namespace:

```
kubectl get deployment -l k8s-app=kube-dns --namespace=kube-system
```

The output is similar to this:

```
NAME      READY   UP-TO-DATE   AVAILABLE   AGE
...
coredns   2/2     2            2           ...
...
```

If you don't see a Deployment for DNS services, you can also look for it by name:

```
kubectl get deployment --namespace=kube-system
```

and look for a deployment named `coredns` or `kube-dns`.

Your scale target is

```
Deployment/<your-deployment-name>
```

where `<your-deployment-name>` is the name of your DNS Deployment. For example, if the name of your Deployment for DNS is coredns, your scale target is Deployment/coredns.

> **Note:** CoreDNS is the default DNS service for Kubernetes. CoreDNS sets the label `k8s-app=kube-dns` so that it can work in clusters that originally used kube-dns.

# Enable DNS horizontal autoscaling

In this section, you create a new Deployment. The Pods in the Deployment run a container based on the `cluster-proportional-autoscaler-amd64` image.

Create a file named `dns-horizontal-autoscaler.yaml` with this content:

admin/dns/dns-horizontal-autoscaler.yaml

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: dns-autoscaler
  namespace: kube-system
  labels:
    k8s-app: dns-autoscaler
spec:
  selector:
    matchLabels:
      k8s-app: dns-autoscaler
  template:
    metadata:
      labels:
        k8s-app: dns-autoscaler
    spec:
      containers:
      - name: autoscaler
        image: k8s.gcr.io/cluster-proportional-autoscaler-amd64:1.6.0
        resources:
          requests:
            cpu: 20m
            memory: 10Mi
        command:
        - /cluster-proportional-autoscaler
        - --namespace=kube-system
        - --configmap=dns-autoscaler
        - --target=<SCALE_TARGET>
        # When cluster is using large nodes(with more cores), "coresPerReplica" should dominate.
        # If using small nodes, "nodesPerReplica" should dominate.
        - --default-params={"linear":{"coresPerReplica":256,"nodesPerReplica":16,"min":1}}
        - --logtostderr=true
        - --v=2
```

In the file, replace `<SCALE_TARGET>` with your scale target.

Go to the directory that contains your configuration file, and enter this command to create the Deployment:

```
kubectl apply -f dns-horizontal-autoscaler.yaml
```

The output of a successful command is:

```
deployment.apps/dns-autoscaler created
```

DNS horizontal autoscaling is now enabled.

# Tune DNS autoscaling parameters

Verify that the dns-autoscaler ConfigMap exists:

```
kubectl get configmap --namespace=kube-system
```

The output is similar to this:

```
NAME                  DATA      AGE
...
dns-autoscaler        1         ...
...
```

Modify the data in the ConfigMap:

```
kubectl edit configmap dns-autoscaler --namespace=kube-system
```

Look for this line:

```
linear: '{"coresPerReplica":256,"min":1,"nodesPerReplica":16}'
```

Modify the fields according to your needs. The "min" field indicates the minimal number of DNS backends. The actual number of backends is calculated using this equation:

```
replicas = max( ceil( cores × 1/coresPerReplica ) , ceil( nodes × 1/nodesPerReplica ) )
```

Note that the values of both `coresPerReplica` and `nodesPerReplica` are floats.

The idea is that when a cluster is using nodes that have many cores, `coresPerReplica` dominates. When a cluster is using nodes that have fewer cores, `nodesPerReplica` dominates.

There are other supported scaling patterns. For details, see cluster-proportional-autoscaler.

# Disable DNS horizontal autoscaling

There are a few options for tuning DNS horizontal autoscaling. Which option to use depends on different conditions.

## Option 1: Scale down the dns-autoscaler deployment to 0 replicas

This option works for all situations. Enter this command:

```
kubectl scale deployment --replicas=0 dns-autoscaler --namespace=kube-system
```

The output is:

```
deployment.apps/dns-autoscaler scaled
```

Verify that the replica count is zero:

```
kubectl get rs --namespace=kube-system
```

The output displays 0 in the DESIRED and CURRENT columns:

```
NAME                                DESIRED   CURRENT   READY   AGE
...
dns-autoscaler-6b59789fc8           0         0         0       ...
...
```

## Option 2: Delete the dns-autoscaler deployment

This option works if dns-autoscaler is under your own control, which means no one will re-create it:

```
kubectl delete deployment dns-autoscaler --namespace=kube-system
```

The output is:

```
deployment.apps "dns-autoscaler" deleted
```

### Option 3: Delete the dns-autoscaler manifest file from the master node

This option works if dns-autoscaler is under control of the (deprecated) Addon Manager, and you have write access to the master node.

Sign in to the master node and delete the corresponding manifest file. The common path for this dns-autoscaler is:

```
/etc/kubernetes/addons/dns-horizontal-autoscaler/dns-horizontal-autoscaler.yaml
```

After the manifest file is deleted, the Addon Manager will delete the dns-autoscaler Deployment.

## Understanding how DNS horizontal autoscaling works

- The cluster-proportional-autoscaler application is deployed separately from the DNS service.
- An autoscaler Pod runs a client that polls the Kubernetes API server for the number of nodes and cores in the cluster.
- A desired replica count is calculated and applied to the DNS backends based on the current schedulable nodes and cores and the given scaling parameters.
- The scaling parameters and data points are provided via a ConfigMap to the autoscaler, and it refreshes its parameters table every poll interval to be up to date with the latest desired scaling parameters.
- Changes to the scaling parameters are allowed without rebuilding or restarting the autoscaler Pod.
- The autoscaler provides a controller interface to support two control patterns: *linear* and *ladder*.

## What's next

- Read about Guaranteed Scheduling For Critical Add-On Pods.
- Learn more about the implementation of cluster-proportional-autoscaler.

# 10 - Change the default StorageClass

This page shows how to change the default Storage Class that is used to provision volumes for PersistentVolumeClaims that have no special requirements.

## Before you begin

You need to have a Kubernetes cluster, and the kubectl command-line tool must be configured to communicate with your cluster. It is recommended to run this tutorial on a cluster with at least two nodes that are not acting as control plane hosts. If you do not already have a cluster, you can create one by using minikube or you can use one of these Kubernetes playgrounds:

- Katacoda
- Play with Kubernetes

To check the version, enter `kubectl version`.

## Why change the default storage class?

Depending on the installation method, your Kubernetes cluster may be deployed with an existing StorageClass that is marked as default. This default StorageClass is then used to dynamically provision storage for PersistentVolumeClaims that do not require any specific storage class. See PersistentVolumeClaim documentation for details.

The pre-installed default StorageClass may not fit well with your expected workload; for example, it might provision storage that is too expensive. If this is the case, you can either change the default StorageClass or disable it completely to avoid dynamic provisioning of storage.

Deleting the default StorageClass may not work, as it may be re-created automatically by the addon manager running in your cluster. Please consult the docs for your installation for details about addon manager and how to disable individual addons.

## Changing the default StorageClass

1. List the StorageClasses in your cluster:

   ```
   kubectl get storageclass
   ```

   The output is similar to this:

   ```
   NAME                 PROVISIONER               AGE
   standard (default)   kubernetes.io/gce-pd      1d
   gold                 kubernetes.io/gce-pd      1d
   ```

   The default StorageClass is marked by `(default)`.

2. Mark the default StorageClass as non-default:

   The default StorageClass has an annotation `storageclass.kubernetes.io/is-default-class` set to `true`. Any other value or absence of the annotation is interpreted as `false`.

   To mark a StorageClass as non-default, you need to change its value to `false`:

   ```
   kubectl patch storageclass standard -p '{"metadata": {"annotations":{"storageclass.kubernetes
   ```

   where `standard` is the name of your chosen StorageClass.

3. Mark a StorageClass as default:

   Similar to the previous step, you need to add/set the annotation `storageclass.kubernetes.io/is-default-class=true`.

   ```
   kubectl patch storageclass gold -p '{"metadata": {"annotations":{"storageclass.kubernetes.io/
   ```

Please note that at most one StorageClass can be marked as default. If two or more of them are marked as default, a `PersistentVolumeClaim` without `storageClassName` explicitly specified cannot be created.

4. Verify that your chosen StorageClass is default:

```
kubectl get storageclass
```

The output is similar to this:

```
NAME                 PROVISIONER               AGE
standard             kubernetes.io/gce-pd      1d
gold (default)       kubernetes.io/gce-pd      1d
```

# What's next

- Learn more about PersistentVolumes.

# 11 - Change the Reclaim Policy of a PersistentVolume

This page shows how to change the reclaim policy of a Kubernetes PersistentVolume.

## Before you begin

You need to have a Kubernetes cluster, and the kubectl command-line tool must be configured to communicate with your cluster. It is recommended to run this tutorial on a cluster with at least two nodes that are not acting as control plane hosts. If you do not already have a cluster, you can create one by using minikube or you can use one of these Kubernetes playgrounds:

- Katacoda
- Play with Kubernetes

To check the version, enter `kubectl version`.

## Why change reclaim policy of a PersistentVolume

PersistentVolumes can have various reclaim policies, including "Retain", "Recycle", and "Delete". For dynamically provisioned PersistentVolumes, the default reclaim policy is "Delete". This means that a dynamically provisioned volume is automatically deleted when a user deletes the corresponding PersistentVolumeClaim. This automatic behavior might be inappropriate if the volume contains precious data. In that case, it is more appropriate to use the "Retain" policy. With the "Retain" policy, if a user deletes a PersistentVolumeClaim, the corresponding PersistentVolume will not be deleted. Instead, it is moved to the Released phase, where all of its data can be manually recovered.

## Changing the reclaim policy of a PersistentVolume

1. List the PersistentVolumes in your cluster:

   ```
   kubectl get pv
   ```

   The output is similar to this:

   ```
   NAME                                       CAPACITY   ACCESSMODES   RECLAIMPOLICY   STATUS
   pvc-b6efd8da-b7b5-11e6-9d58-0ed433a7dd94   4Gi        RWO           Delete          Bound
   pvc-b95650f8-b7b5-11e6-9d58-0ed433a7dd94   4Gi        RWO           Delete          Bound
   pvc-bb3ca71d-b7b5-11e6-9d58-0ed433a7dd94   4Gi        RWO           Delete          Bound
   ```

   This list also includes the name of the claims that are bound to each volume for easier identification of dynamically provisioned volumes.

2. Choose one of your PersistentVolumes and change its reclaim policy:

   ```
   kubectl patch pv <your-pv-name> -p '{"spec":{"persistentVolumeReclaimPolicy":"Retain"}}'
   ```

   where `<your-pv-name>` is the name of your chosen PersistentVolume.

   > **Note:**
   > On Windows, you must *double* quote any JSONPath template that contains spaces (not single quote as shown above for bash). This in turn means that you must use a single quote or escaped double quote around any literals in the template. For example:
   >
   > ```
   > kubectl patch pv <your-pv-name> -p "{\"spec\":{\"persistentVolumeReclaimPolicy\":\"Retain
   > ```

3. Verify that your chosen PersistentVolume has the right policy:

   ```
   kubectl get pv
   ```

The output is similar to this:

```
NAME                                       CAPACITY   ACCESSMODES   RECLAIMPOLICY   STATUS
pvc-b6efd8da-b7b5-11e6-9d58-0ed433a7dd94   4Gi        RWO           Delete          Bound
pvc-b95650f8-b7b5-11e6-9d58-0ed433a7dd94   4Gi        RWO           Delete          Bound
pvc-bb3ca71d-b7b5-11e6-9d58-0ed433a7dd94   4Gi        RWO           Retain          Bound
```

In the preceding output, you can see that the volume bound to claim `default/claim3` has reclaim policy `Retain`. It will not be automatically deleted when a user deletes claim `default/claim3`.

# What's next

- Learn more about PersistentVolumes.
- Learn more about PersistentVolumeClaims.

## Reference

- PersistentVolume
- PersistentVolumeClaim
- See the `persistentVolumeReclaimPolicy` field of PersistentVolumeSpec.

# 12 - Cloud Controller Manager Administration

**FEATURE STATE:** `Kubernetes v1.11 [beta]`

Since cloud providers develop and release at a different pace compared to the Kubernetes project, abstracting the provider-specific code to the `cloud-controller-manager` binary allows cloud vendors to evolve independently from the core Kubernetes code.

The `cloud-controller-manager` can be linked to any cloud provider that satisfies cloudprovider.Interface. For backwards compatibility, the cloud-controller-manager provided in the core Kubernetes project uses the same cloud libraries as `kube-controller-manager`. Cloud providers already supported in Kubernetes core are expected to use the in-tree cloud-controller-manager to transition out of Kubernetes core.

## Administration

### Requirements

Every cloud has their own set of requirements for running their own cloud provider integration, it should not be too different from the requirements when running `kube-controller-manager`. As a general rule of thumb you'll need:

- cloud authentication/authorization: your cloud may require a token or IAM rules to allow access to their APIs
- kubernetes authentication/authorization: cloud-controller-manager may need RBAC rules set to speak to the kubernetes apiserver
- high availability: like kube-controller-manager, you may want a high available setup for cloud controller manager using leader election (on by default).

### Running cloud-controller-manager

Successfully running cloud-controller-manager requires some changes to your cluster configuration.

- `kube-apiserver` and `kube-controller-manager` MUST NOT specify the `--cloud-provider` flag. This ensures that it does not run any cloud specific loops that would be run by cloud controller manager. In the future, this flag will be deprecated and removed.
- `kubelet` must run with `--cloud-provider=external`. This is to ensure that the kubelet is aware that it must be initialized by the cloud controller manager before it is scheduled any work.

Keep in mind that setting up your cluster to use cloud controller manager will change your cluster behaviour in a few ways:

- kubelets specifying `--cloud-provider=external` will add a taint `node.cloudprovider.kubernetes.io/uninitialized` with an effect `NoSchedule` during initialization. This marks the node as needing a second initialization from an external controller before it can be scheduled work. Note that in the event that cloud controller manager is not available, new nodes in the cluster will be left unschedulable. The taint is important since the scheduler may require cloud specific information about nodes such as their region or type (high cpu, gpu, high memory, spot instance, etc).
- cloud information about nodes in the cluster will no longer be retrieved using local metadata, but instead all API calls to retrieve node information will go through cloud controller manager. This may mean you can restrict access to your cloud API on the kubelets for better security. For larger clusters you may want to consider if cloud controller manager will hit rate limits since it is now responsible for almost all API calls to your cloud from within the cluster.

The cloud controller manager can implement:

- Node controller - responsible for updating kubernetes nodes using cloud APIs and deleting kubernetes nodes that were deleted on your cloud.
- Service controller - responsible for loadbalancers on your cloud against services of type LoadBalancer.
- Route controller - responsible for setting up network routes on your cloud
- any other features you would like to implement if you are running an out-of-tree provider.

## Examples

If you are using a cloud that is currently supported in Kubernetes core and would like to adopt cloud controller manager, see the cloud controller manager in kubernetes core.

For cloud controller managers not in Kubernetes core, you can find the respective projects in repositories maintained by cloud vendors or by SIGs.

For providers already in Kubernetes core, you can run the in-tree cloud controller manager as a DaemonSet in your cluster, use the following as a guideline:

admin/cloud/ccm-example.yaml

```yaml
# This is an example of how to setup cloud-controller-manager as a Daemonset in your cluster.
# It assumes that your masters can run pods and has the role node-role.kubernetes.io/master
# Note that this Daemonset will not work straight out of the box for your cloud, this is
# meant to be a guideline.

---
apiVersion: v1
kind: ServiceAccount
metadata:
  name: cloud-controller-manager
  namespace: kube-system
---
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRoleBinding
metadata:
  name: system:cloud-controller-manager
roleRef:
  apiGroup: rbac.authorization.k8s.io
  kind: ClusterRole
  name: cluster-admin
subjects:
- kind: ServiceAccount
  name: cloud-controller-manager
  namespace: kube-system
---
apiVersion: apps/v1
kind: DaemonSet
metadata:
  labels:
    k8s-app: cloud-controller-manager
  name: cloud-controller-manager
  namespace: kube-system
spec:
  selector:
    matchLabels:
      k8s-app: cloud-controller-manager
  template:
    metadata:
      labels:
        k8s-app: cloud-controller-manager
    spec:
      serviceAccountName: cloud-controller-manager
      containers:
      - name: cloud-controller-manager
        # for in-tree providers we use k8s.gcr.io/cloud-controller-manager
        # this can be replaced with any other image for out-of-tree providers
        image: k8s.gcr.io/cloud-controller-manager:v1.8.0
        command:
        - /usr/local/bin/cloud-controller-manager
        - --cloud-provider=[YOUR_CLOUD_PROVIDER]  # Add your own cloud provider here!
        - --leader-elect=true
        - --use-service-account-credentials
        # these flags will vary for every cloud provider
        - --allocate-node-cidrs=true
        - --configure-cloud-routes=true
        - --cluster-cidr=172.17.0.0/16
      tolerations:
      # this is required so CCM can bootstrap itself
      - key: node.cloudprovider.kubernetes.io/uninitialized
        value: "true"
        effect: NoSchedule
      # this is to have the daemonset runnable on master nodes
      # the taint may vary depending on your cluster setup
      - key: node-role.kubernetes.io/master
        effect: NoSchedule
      # this is to restrict CCM to only run on master nodes
      # the node selector may vary depending on your cluster setup
      nodeSelector:
        node-role.kubernetes.io/master: ""
```

# Limitations

Running cloud controller manager comes with a few possible limitations. Although these limitations are being addressed in upcoming releases, it's important that you are aware of these limitations for production workloads.

## Support for Volumes

Cloud controller manager does not implement any of the volume controllers found in `kube-controller-manager` as the volume integrations also require coordination with kubelets. As we evolve CSI (container storage interface) and add stronger support for flex volume plugins, necessary support will be added to cloud controller manager so that clouds can fully integrate with volumes. Learn more about out-of-tree CSI volume plugins here.

## Scalability

The cloud-controller-manager queries your cloud provider's APIs to retrieve information for all nodes. For very large clusters, consider possible bottlenecks such as resource requirements and API rate limiting.

## Chicken and Egg

The goal of the cloud controller manager project is to decouple development of cloud features from the core Kubernetes project. Unfortunately, many aspects of the Kubernetes project has assumptions that cloud provider features are tightly integrated into the project. As a result, adopting this new architecture can create several situations where a request is being made for information from a cloud provider, but the cloud controller manager may not be able to return that information without the original request being complete.

A good example of this is the TLS bootstrapping feature in the Kubelet. TLS bootstrapping assumes that the Kubelet has the ability to ask the cloud provider (or a local metadata service) for all its address types (private, public, etc) but cloud controller manager cannot set a node's address types without being initialized in the first place which requires that the kubelet has TLS certificates to communicate with the apiserver.

As this initiative evolves, changes will be made to address these issues in upcoming releases.

## What's next

To build and develop your own cloud controller manager, read Developing Cloud Controller Manager.

# 13 - Configure Quotas for API Objects

This page shows how to configure quotas for API objects, including PersistentVolumeClaims and Services. A quota restricts the number of objects, of a particular type, that can be created in a namespace. You specify quotas in a ResourceQuota object.

## Before you begin

You need to have a Kubernetes cluster, and the kubectl command-line tool must be configured to communicate with your cluster. It is recommended to run this tutorial on a cluster with at least two nodes that are not acting as control plane hosts. If you do not already have a cluster, you can create one by using minikube or you can use one of these Kubernetes playgrounds:

- Katacoda
- Play with Kubernetes

To check the version, enter `kubectl version`.

## Create a namespace

Create a namespace so that the resources you create in this exercise are isolated from the rest of your cluster.

```
kubectl create namespace quota-object-example
```

## Create a ResourceQuota

Here is the configuration file for a ResourceQuota object:

admin/resource/quota-objects.yaml

```yaml
apiVersion: v1
kind: ResourceQuota
metadata:
  name: object-quota-demo
spec:
  hard:
    persistentvolumeclaims: "1"
    services.loadbalancers: "2"
    services.nodeports: "0"
```

Create the ResourceQuota:

```
kubectl apply -f https://k8s.io/examples/admin/resource/quota-objects.yaml --namespace=quota-obje
```

View detailed information about the ResourceQuota:

```
kubectl get resourcequota object-quota-demo --namespace=quota-object-example --output=yaml
```

The output shows that in the quota-object-example namespace, there can be at most one PersistentVolumeClaim, at most two Services of type LoadBalancer, and no Services of type NodePort.

```yaml
status:
  hard:
    persistentvolumeclaims: "1"
    services.loadbalancers: "2"
    services.nodeports: "0"
  used:
    persistentvolumeclaims: "0"
    services.loadbalancers: "0"
    services.nodeports: "0"
```

# Create a PersistentVolumeClaim

Here is the configuration file for a PersistentVolumeClaim object:

admin/resource/quota-objects-pvc.yaml

```yaml
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: pvc-quota-demo
spec:
  storageClassName: manual
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: 3Gi
```

Create the PersistentVolumeClaim:

```
kubectl apply -f https://k8s.io/examples/admin/resource/quota-objects-pvc.yaml --namespace=quota-
```

Verify that the PersistentVolumeClaim was created:

```
kubectl get persistentvolumeclaims --namespace=quota-object-example
```

The output shows that the PersistentVolumeClaim exists and has status Pending:

```
NAME             STATUS
pvc-quota-demo   Pending
```

# Attempt to create a second PersistentVolumeClaim

Here is the configuration file for a second PersistentVolumeClaim:

admin/resource/quota-objects-pvc-2.yaml

```yaml
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: pvc-quota-demo-2
spec:
  storageClassName: manual
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: 4Gi
```

Attempt to create the second PersistentVolumeClaim:

```
kubectl apply -f https://k8s.io/examples/admin/resource/quota-objects-pvc-2.yaml --namespace=quot
```

The output shows that the second PersistentVolumeClaim was not created, because it would have exceeded the quota for the namespace.

```
persistentvolumeclaims "pvc-quota-demo-2" is forbidden:
exceeded quota: object-quota-demo, requested: persistentvolumeclaims=1,
used: persistentvolumeclaims=1, limited: persistentvolumeclaims=1
```

# Notes

These are the strings used to identify API resources that can be constrained by quotas:

| String | API Object |
| --- | --- |
| "pods" | Pod |
| "services" | Service |
| "replicationcontrollers" | ReplicationController |
| "resourcequotas" | ResourceQuota |
| "secrets" | Secret |
| "configmaps" | ConfigMap |
| "persistentvolumeclaims" | PersistentVolumeClaim |
| "services.nodeports" | Service of type NodePort |
| "services.loadbalancers" | Service of type LoadBalancer |

# Clean up

Delete your namespace:

```
kubectl delete namespace quota-object-example
```

# What's next

## For cluster administrators

- Configure Default Memory Requests and Limits for a Namespace
- Configure Default CPU Requests and Limits for a Namespace
- Configure Minimum and Maximum Memory Constraints for a Namespace
- Configure Minimum and Maximum CPU Constraints for a Namespace
- Configure Memory and CPU Quotas for a Namespace
- Configure a Pod Quota for a Namespace

## For app developers

- Assign Memory Resources to Containers and Pods
- Assign CPU Resources to Containers and Pods
- Configure Quality of Service for Pods

# 14 - Control CPU Management Policies on the Node

**FEATURE STATE:** `Kubernetes v1.12 [beta]`

Kubernetes keeps many aspects of how pods execute on nodes abstracted from the user. This is by design. However, some workloads require stronger guarantees in terms of latency and/or performance in order to operate acceptably. The kubelet provides methods to enable more complex workload placement policies while keeping the abstraction free from explicit placement directives.

## Before you begin

You need to have a Kubernetes cluster, and the kubectl command-line tool must be configured to communicate with your cluster. It is recommended to run this tutorial on a cluster with at least two nodes that are not acting as control plane hosts. If you do not already have a cluster, you can create one by using minikube or you can use one of these Kubernetes playgrounds:

- Katacoda
- Play with Kubernetes

To check the version, enter `kubectl version`.

## CPU Management Policies

By default, the kubelet uses CFS quota to enforce pod CPU limits. When the node runs many CPU-bound pods, the workload can move to different CPU cores depending on whether the pod is throttled and which CPU cores are available at scheduling time. Many workloads are not sensitive to this migration and thus work fine without any intervention.

However, in workloads where CPU cache affinity and scheduling latency significantly affect workload performance, the kubelet allows alternative CPU management policies to determine some placement preferences on the node.

### Configuration

The CPU Manager policy is set with the `--cpu-manager-policy` kubelet option. There are two supported policies:

- `none`: the default policy.
- `static`: allows pods with certain resource characteristics to be granted increased CPU affinity and exclusivity on the node.

The CPU manager periodically writes resource updates through the CRI in order to reconcile in-memory CPU assignments with cgroupfs. The reconcile frequency is set through a new Kubelet configuration value `--cpu-manager-reconcile-period`. If not specified, it defaults to the same duration as `--node-status-update-frequency`.

### None policy

The `none` policy explicitly enables the existing default CPU affinity scheme, providing no affinity beyond what the OS scheduler does automatically. Limits on CPU usage for Guaranteed pods are enforced using CFS quota.

### Static policy

The `static` policy allows containers in `Guaranteed` pods with integer CPU `requests` access to exclusive CPUs on the node. This exclusivity is enforced using the cpuset cgroup controller.

> **Note:** System services such as the container runtime and the kubelet itself can continue to run on these exclusive CPUs. The exclusivity only extends to other pods.

> **Note:** CPU Manager doesn't support offlining and onlining of CPUs at runtime. Also, if the set of online CPUs changes on the node, the node must be drained and CPU manager manually reset by deleting the state file `cpu_manager_state` in the kubelet root directory.

This policy manages a shared pool of CPUs that initially contains all CPUs in the node. The amount of exclusively allocatable CPUs is equal to the total number of CPUs in the node minus any CPU reservations by the kubelet `--kube-reserved` or `--system-reserved` options. From 1.17, the CPU reservation list can be specified explicitly by kubelet `--reserved-cpus` option. The explicit CPU list specified by `--reserved-cpus` takes precedence over the CPU reservation specified by `--kube-reserved` and `--system-reserved`. CPUs reserved by these options are taken, in integer quantity, from the initial shared pool in ascending order by physical core ID. This shared pool is the set of CPUs on which any containers in `BestEffort` and `Burstable` pods run. Containers in `Guaranteed` pods with fractional CPU `requests` also run on CPUs in the shared pool. Only containers that are both part of a `Guaranteed` pod and have integer CPU `requests` are assigned exclusive CPUs.

> **Note:** The kubelet requires a CPU reservation greater than zero be made using either `--kube-reserved` and/or `--system-reserved` or `--reserved-cpus` when the static policy is enabled. This is because zero CPU reservation would allow the shared pool to become empty.

As `Guaranteed` pods whose containers fit the requirements for being statically assigned are scheduled to the node, CPUs are removed from the shared pool and placed in the cpuset for the container. CFS quota is not used to bound the CPU usage of these containers as their usage is bound by the scheduling domain itself. In others words, the number of CPUs in the container cpuset is equal to the integer CPU `limit` specified in the pod spec. This static assignment increases CPU affinity and decreases context switches due to throttling for the CPU-bound workload.

Consider the containers in the following pod specs:

```
spec:
  containers:
  - name: nginx
    image: nginx
```

This pod runs in the `BestEffort` QoS class because no resource `requests` or `limits` are specified. It runs in the shared pool.

```
spec:
  containers:
  - name: nginx
    image: nginx
    resources:
      limits:
        memory: "200Mi"
      requests:
        memory: "100Mi"
```

This pod runs in the `Burstable` QoS class because resource `requests` do not equal `limits` and the `cpu` quantity is not specified. It runs in the shared pool.

```
spec:
  containers:
  - name: nginx
    image: nginx
    resources:
      limits:
        memory: "200Mi"
        cpu: "2"
      requests:
        memory: "100Mi"
        cpu: "1"
```

This pod runs in the `Burstable` QoS class because resource `requests` do not equal `limits`. It runs in the shared pool.

```
spec:
  containers:
  - name: nginx
    image: nginx
    resources:
      limits:
        memory: "200Mi"
        cpu: "2"
      requests:
        memory: "200Mi"
        cpu: "2"
```

This pod runs in the `Guaranteed` QoS class because `requests` are equal to `limits`. And the container's resource limit for the CPU resource is an integer greater than or equal to one. The `nginx` container is granted 2 exclusive CPUs.

```
spec:
  containers:
  - name: nginx
    image: nginx
    resources:
```

```
      limits:
        memory: "200Mi"
        cpu: "1.5"
      requests:
        memory: "200Mi"
        cpu: "1.5"
```

This pod runs in the `Guaranteed` QoS class because `requests` are equal to `limits`. But the container's resource limit for the CPU resource is a fraction. It runs in the shared pool.

```
spec:
  containers:
  - name: nginx
    image: nginx
    resources:
      limits:
        memory: "200Mi"
        cpu: "2"
```

This pod runs in the `Guaranteed` QoS class because only `limits` are specified and `requests` are set equal to `limits` when not explicitly specified. And the container's resource limit for the CPU resource is an integer greater than or equal to one. The `nginx` container is granted 2 exclusive CPUs.

# 15 - Control Topology Management Policies on a node

**FEATURE STATE:** `Kubernetes v1.18 [beta]`

An increasing number of systems leverage a combination of CPUs and hardware accelerators to support latency-critical execution and high-throughput parallel computation. These include workloads in fields such as telecommunications, scientific computing, machine learning, financial services and data analytics. Such hybrid systems comprise a high performance environment.

In order to extract the best performance, optimizations related to CPU isolation, memory and device locality are required. However, in Kubernetes, these optimizations are handled by a disjoint set of components.

*Topology Manager* is a Kubelet component that aims to co-ordinate the set of components that are responsible for these optimizations.

## Before you begin

You need to have a Kubernetes cluster, and the kubectl command-line tool must be configured to communicate with your cluster. It is recommended to run this tutorial on a cluster with at least two nodes that are not acting as control plane hosts. If you do not already have a cluster, you can create one by using minikube or you can use one of these Kubernetes playgrounds:

- Katacoda
- Play with Kubernetes

Your Kubernetes server must be at or later than version v1.18. To check the version, enter `kubectl version`.

## How Topology Manager Works

Prior to the introduction of Topology Manager, the CPU and Device Manager in Kubernetes make resource allocation decisions independently of each other. This can result in undesirable allocations on multiple-socketed systems, performance/latency sensitive applications will suffer due to these undesirable allocations. Undesirable in this case meaning for example, CPUs and devices being allocated from different NUMA Nodes thus, incurring additional latency.

The Topology Manager is a Kubelet component, which acts as a source of truth so that other Kubelet components can make topology aligned resource allocation choices.

The Topology Manager provides an interface for components, called *Hint Providers*, to send and receive topology information. Topology Manager has a set of node level policies which are explained below.

The Topology manager receives Topology information from the *Hint Providers* as a bitmask denoting NUMA Nodes available and a preferred allocation indication. The Topology Manager policies perform a set of operations on the hints provided and converge on the hint determined by the policy to give the optimal result, if an undesirable hint is stored the preferred field for the hint will be set to false. In the current policies preferred is the narrowest preferred mask. The selected hint is stored as part of the Topology Manager. Depending on the policy configured the pod can be accepted or rejected from the node based on the selected hint. The hint is then stored in the Topology Manager for use by the *Hint Providers* when making the resource allocation decisions.

### Enable the Topology Manager feature

Support for the Topology Manager requires `TopologyManager` feature gate to be enabled. It is enabled by default starting with Kubernetes 1.18.

## Topology Manager Scopes and Policies

The Topology Manager currently:

- Aligns Pods of all QoS classes.
- Aligns the requested resources that Hint Provider provides topology hints for.

If these conditions are met, the Topology Manager will align the requested resources.

In order to customise how this alignment is carried out, the Topology Manager provides two distinct knobs: `scope` and `policy`.

The `scope` defines the granularity at which you would like resource alignment to be performed (e.g. at the `pod` or `container` level). And the `policy` defines the actual strategy used to carry out the alignment (e.g. `best-effort`, `restricted`, `single-numa-node`, etc.).

Details on the various `scopes` and `policies` available today can be found below.

> **Note:** To align CPU resources with other requested resources in a Pod Spec, the CPU Manager should be enabled and proper CPU Manager policy should be configured on a Node. See control CPU Management Policies.

> **Note:** To align memory (and hugepages) resources with other requested resources in a Pod Spec, the Memory Manager should be enabled and proper Memory Manager policy should be configured on a Node. Examine Memory Manager documentation.

## Topology Manager Scopes

The Topology Manager can deal with the alignment of resources in a couple of distinct scopes:

- `container` (default)
- `pod`

Either option can be selected at a time of the kubelet startup, with `--topology-manager-scope` flag.

## container scope

The `container` scope is used by default.

Within this scope, the Topology Manager performs a number of sequential resource alignments, i.e., for each container (in a pod) a separate alignment is computed. In other words, there is no notion of grouping the containers to a specific set of NUMA nodes, for this particular scope. In effect, the Topology Manager performs an arbitrary alignment of individual containers to NUMA nodes.

The notion of grouping the containers was endorsed and implemented on purpose in the following scope, for example the `pod` scope.

## pod scope

To select the `pod` scope, start the kubelet with the command line option `--topology-manager-scope=pod`.

This scope allows for grouping all containers in a pod to a common set of NUMA nodes. That is, the Topology Manager treats a pod as a whole and attempts to allocate the entire pod (all containers) to either a single NUMA node or a common set of NUMA nodes. The following examples illustrate the alignments produced by the Topology Manager on different occasions:

- all containers can be and are allocated to a single NUMA node;
- all containers can be and are allocated to a shared set of NUMA nodes.

The total amount of particular resource demanded for the entire pod is calculated according to effective requests/limits formula, and thus, this total value is equal to the maximum of:

- the sum of all app container requests,
- the maximum of init container requests, for a resource.

Using the `pod` scope in tandem with `single-numa-node` Topology Manager policy is specifically valuable for workloads that are latency sensitive or for high-throughput applications that perform IPC. By combining both options, you are able to place all containers in a pod onto a single NUMA node; hence, the inter-NUMA communication overhead can be eliminated for that pod.

In the case of `single-numa-node` policy, a pod is accepted only if a suitable set of NUMA nodes is present among possible allocations. Reconsider the example above:

- a set containing only a single NUMA node - it leads to pod being admitted,
- whereas a set containing more NUMA nodes - it results in pod rejection (because instead of one NUMA node, two or more NUMA nodes are required to satisfy the allocation).

To recap, Topology Manager first computes a set of NUMA nodes and then tests it against Topology Manager policy, which either leads to the rejection or admission of the pod.

## Topology Manager Policies

Topology Manager supports four allocation policies. You can set a policy via a Kubelet flag, `--topology-manager-policy`. There are four supported policies:

- `none` (default)
- `best-effort`
- `restricted`
- `single-numa-node`

> **Note:** If Topology Manager is configured with the **pod** scope, the container, which is considered by the policy, is reflecting requirements of the entire pod, and thus each container from the pod will result with **the same** topology alignment decision.

## none policy

This is the default policy and does not perform any topology alignment.

## best-effort policy

For each container in a Pod, the kubelet, with `best-effort` topology management policy, calls each Hint Provider to discover their resource availability. Using this information, the Topology Manager stores the preferred NUMA Node affinity for that container. If the affinity is not preferred, Topology Manager will store this and admit the pod to the node anyway.

The *Hint Providers* can then use this information when making the resource allocation decision.

## restricted policy

For each container in a Pod, the kubelet, with `restricted` topology management policy, calls each Hint Provider to discover their resource availability. Using this information, the Topology Manager stores the preferred NUMA Node affinity for that container. If the affinity is not preferred, Topology Manager will reject this pod from the node. This will result in a pod in a `Terminated` state with a pod admission failure.

Once the pod is in a `Terminated` state, the Kubernetes scheduler will **not** attempt to reschedule the pod. It is recommended to use a ReplicaSet or Deployment to trigger a redeploy of the pod. An external control loop could be also implemented to trigger a redeployment of pods that have the `Topology Affinity` error.

If the pod is admitted, the *Hint Providers* can then use this information when making the resource allocation decision.

## single-numa-node policy

For each container in a Pod, the kubelet, with `single-numa-node` topology management policy, calls each Hint Provider to discover their resource availability. Using this information, the Topology Manager determines if a single NUMA Node affinity is possible. If it is, Topology Manager will store this and the *Hint Providers* can then use this information when making the resource allocation decision. If, however, this is not possible then the Topology Manager will reject the pod from the node. This will result in a pod in a `Terminated` state with a pod admission failure.

Once the pod is in a `Terminated` state, the Kubernetes scheduler will **not** attempt to reschedule the pod. It is recommended to use a Deployment with replicas to trigger a redeploy of the Pod. An external control loop could be also implemented to trigger a redeployment of pods that have the `Topology Affinity` error.

## Pod Interactions with Topology Manager Policies

Consider the containers in the following pod specs:

```
spec:
  containers:
  - name: nginx
    image: nginx
```

This pod runs in the `BestEffort` QoS class because no resource `requests` or `limits` are specified.

```
spec:
  containers:
  - name: nginx
    image: nginx
    resources:
      limits:
        memory: "200Mi"
      requests:
        memory: "100Mi"
```

This pod runs in the `Burstable` QoS class because requests are less than limits.

If the selected policy is anything other than `none`, Topology Manager would consider these Pod specifications. The Topology Manager would consult the Hint Providers to get topology hints. In the case of the `static`, the CPU Manager policy would return default topology hint, because these Pods do not have explicitly request CPU resources.

```
spec:
  containers:
```

```
  - name: nginx
    image: nginx
    resources:
      limits:
        memory: "200Mi"
        cpu: "2"
        example.com/device: "1"
      requests:
        memory: "200Mi"
        cpu: "2"
        example.com/device: "1"
```

This pod with integer CPU request runs in the `Guaranteed` QoS class because `requests` are equal to `limits`.

```
spec:
  containers:
  - name: nginx
    image: nginx
    resources:
      limits:
        memory: "200Mi"
        cpu: "300m"
        example.com/device: "1"
      requests:
        memory: "200Mi"
        cpu: "300m"
        example.com/device: "1"
```

This pod with sharing CPU request runs in the `Guaranteed` QoS class because `requests` are equal to `limits`.

```
spec:
  containers:
  - name: nginx
    image: nginx
    resources:
      limits:
        example.com/deviceA: "1"
        example.com/deviceB: "1"
      requests:
        example.com/deviceA: "1"
        example.com/deviceB: "1"
```

This pod runs in the `BestEffort` QoS class because there are no CPU and memory requests.

The Topology Manager would consider the above pods. The Topology Manager would consult the Hint Providers, which are CPU and Device Manager to get topology hints for the pods.

In the case of the `Guaranteed` pod with integer CPU request, the `static` CPU Manager policy would return topology hints relating to the exclusive CPU and the Device Manager would send back hints for the requested device.

In the case of the `Guaranteed` pod with sharing CPU request, the `static` CPU Manager policy would return default topology hint as there is no exclusive CPU request and the Device Manager would send back hints for the requested device.

In the above two cases of the `Guaranteed` pod, the `none` CPU Manager policy would return default topology hint.

In the case of the `BestEffort` pod, the `static` CPU Manager policy would send back the default topology hint as there is no CPU request and the Device Manager would send back the hints for each of the requested devices.

Using this information the Topology Manager calculates the optimal hint for the pod and stores this information, which will be used by the Hint Providers when they are making their resource assignments.

## Known Limitations

1. The maximum number of NUMA nodes that Topology Manager allows is 8. With more than 8 NUMA nodes there will be a state explosion when trying to enumerate the possible NUMA affinities and generating their hints.

2. The scheduler is not topology-aware, so it is possible to be scheduled on a node and then fail on the node due to the Topology Manager.

# 16 - Customizing DNS Service

This page explains how to configure your DNS Pod(s) and customize the DNS resolution process in your cluster.

## Before you begin

You need to have a Kubernetes cluster, and the kubectl command-line tool must be configured to communicate with your cluster. It is recommended to run this tutorial on a cluster with at least two nodes that are not acting as control plane hosts. If you do not already have a cluster, you can create one by using minikube or you can use one of these Kubernetes playgrounds:

- Katacoda
- Play with Kubernetes

Your cluster must be running the CoreDNS add-on. Migrating to CoreDNS explains how to use `kubeadm` to migrate from `kube-dns`.

Your Kubernetes server must be at or later than version v1.12. To check the version, enter `kubectl version`.

## Introduction

DNS is a built-in Kubernetes service launched automatically using the *addon manager* cluster add-on.

As of Kubernetes v1.12, CoreDNS is the recommended DNS Server, replacing kube-dns. If your cluster originally used kube-dns, you may still have `kube-dns` deployed rather than CoreDNS.

> **Note:** The CoreDNS Service is named `kube-dns` in the `metadata.name` field.
> This is so that there is greater interoperability with workloads that relied on the legacy `kube-dns` Service name to resolve addresses internal to the cluster. Using a Service named `kube-dns` abstracts away the implementation detail of which DNS provider is running behind that common name.

If you are running CoreDNS as a Deployment, it will typically be exposed as a Kubernetes Service with a static IP address. The kubelet passes DNS resolver information to each container with the `--cluster-dns=<dns-service-ip>` flag.

DNS names also need domains. You configure the local domain in the kubelet with the flag `--cluster-domain=<default-local-domain>`.

The DNS server supports forward lookups (A and AAAA records), port lookups (SRV records), reverse IP address lookups (PTR records), and more. For more information, see DNS for Services and Pods.

If a Pod's `dnsPolicy` is set to `default`, it inherits the name resolution configuration from the node that the Pod runs on. The Pod's DNS resolution should behave the same as the node. But see Known issues.

If you don't want this, or if you want a different DNS config for pods, you can use the kubelet's `--resolv-conf` flag. Set this flag to "" to prevent Pods from inheriting DNS. Set it to a valid file path to specify a file other than `/etc/resolv.conf` for DNS inheritance.

## CoreDNS

CoreDNS is a general-purpose authoritative DNS server that can serve as cluster DNS, complying with the dns specifications.

### CoreDNS ConfigMap options

CoreDNS is a DNS server that is modular and pluggable, and each plugin adds new functionality to CoreDNS. This can be configured by maintaining a Corefile, which is the CoreDNS configuration file. As a cluster administrator, you can modify the ConfigMap for the CoreDNS Corefile to change how DNS service discovery behaves for that cluster.

In Kubernetes, CoreDNS is installed with the following default Corefile configuration:

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: coredns
  namespace: kube-system
data:
  Corefile: |
    .:53 {
        errors
        health {
            lameduck 5s
        }
```

```
        ready
        kubernetes cluster.local in-addr.arpa ip6.arpa {
            pods insecure
            fallthrough in-addr.arpa ip6.arpa
            ttl 30
        }
        prometheus :9153
        forward . /etc/resolv.conf
        cache 30
        loop
        reload
        loadbalance
    }
```

The Corefile configuration includes the following plugins of CoreDNS:

- errors: Errors are logged to stdout.
- health: Health of CoreDNS is reported to `http://localhost:8080/health`. In this extended syntax `lameduck` will make the process unhealthy then wait for 5 seconds before the process is shut down.
- ready: An HTTP endpoint on port 8181 will return 200 OK, when all plugins that are able to signal readiness have done so.
- kubernetes: CoreDNS will reply to DNS queries based on IP of the services and pods of Kubernetes. You can find more details about that plugin on the CoreDNS website. `ttl` allows you to set a custom TTL for responses. The default is 5 seconds. The minimum TTL allowed is 0 seconds, and the maximum is capped at 3600 seconds. Setting TTL to 0 will prevent records from being cached.
  The `pods insecure` option is provided for backward compatibility with *kube-dns*. You can use the `pods verified` option, which returns an A record only if there exists a pod in same namespace with matching IP.
  The `pods disabled` option can be used if you don't use pod records.
- prometheus: Metrics of CoreDNS are available at `http://localhost:9153/metrics` in Prometheus format (also known as OpenMetrics).
- forward: Any queries that are not within the cluster domain of Kubernetes will be forwarded to predefined resolvers (/etc/resolv.conf).
- cache: This enables a frontend cache.
- loop: Detects simple forwarding loops and halts the CoreDNS process if a loop is found.
- reload: Allows automatic reload of a changed Corefile. After you edit the ConfigMap configuration, allow two minutes for your changes to take effect.
- loadbalance: This is a round-robin DNS loadbalancer that randomizes the order of A, AAAA, and MX records in the answer.

You can modify the default CoreDNS behavior by modifying the ConfigMap.

## Configuration of Stub-domain and upstream nameserver using CoreDNS

CoreDNS has the ability to configure stubdomains and upstream nameservers using the forward plugin.

### Example

If a cluster operator has a Consul domain server located at 10.150.0.1, and all Consul names have the suffix .consul.local. To configure it in CoreDNS, the cluster administrator creates the following stanza in the CoreDNS ConfigMap.

```
consul.local:53 {
        errors
        cache 30
        forward . 10.150.0.1
    }
```

To explicitly force all non-cluster DNS lookups to go through a specific nameserver at 172.16.0.1, point the `forward` to the nameserver instead of `/etc/resolv.conf`

```
forward .  172.16.0.1
```

The final ConfigMap along with the default `Corefile` configuration looks like:

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: coredns
  namespace: kube-system
data:
  Corefile: |
    .:53 {
```

```
        errors
        health
        kubernetes cluster.local in-addr.arpa ip6.arpa {
           pods insecure
           fallthrough in-addr.arpa ip6.arpa
        }
        prometheus :9153
        forward . 172.16.0.1
        cache 30
        loop
        reload
        loadbalance
    }
    consul.local:53 {
        errors
        cache 30
        forward . 10.150.0.1
    }
```

The `kubeadm` tool supports automatic translation from the kube-dns ConfigMap to the equivalent CoreDNS ConfigMap.

> **Note:** While kube-dns accepts an FQDN for stubdomain and nameserver (eg: ns.foo.com), CoreDNS does not support this feature. During translation, all FQDN nameservers will be omitted from the CoreDNS config.

# CoreDNS configuration equivalent to kube-dns

CoreDNS supports the features of kube-dns and more. A ConfigMap created for kube-dns to support `StubDomains` and `upstreamNameservers` translates to the `forward` plugin in CoreDNS.

## Example

This example ConfigMap for kube-dns specifies stubdomains and upstreamnameservers:

```
apiVersion: v1
data:
  stubDomains: |
        {"abc.com" : ["1.2.3.4"], "my.cluster.local" : ["2.3.4.5"]}
  upstreamNameservers: |
        ["8.8.8.8", "8.8.4.4"]
kind: ConfigMap
```

The equivalent configuration in CoreDNS creates a Corefile:

- For stubDomains:

```
abc.com:53 {
    errors
    cache 30
    forward . 1.2.3.4
}
my.cluster.local:53 {
    errors
    cache 30
    forward . 2.3.4.5
}
```

The complete Corefile with the default plugins:

```
.:53 {
    errors
    health
    kubernetes cluster.local in-addr.arpa ip6.arpa {
        pods insecure
        fallthrough in-addr.arpa ip6.arpa
    }
    federation cluster.local {
        foo foo.feddomain.com
    }
    prometheus :9153
    forward . 8.8.8.8 8.8.4.4
    cache 30
}
abc.com:53 {
    errors
    cache 30
    forward . 1.2.3.4
}
my.cluster.local:53 {
    errors
    cache 30
    forward . 2.3.4.5
}
```

## Migration to CoreDNS

To migrate from kube-dns to CoreDNS, a detailed blog article is available to help users adapt CoreDNS in place of kube-dns.

You can also migrate using the official CoreDNS deploy script.

## What's next

- Read Debugging DNS Resolution

# 17 - Debugging DNS Resolution

This page provides hints on diagnosing DNS problems.

## Before you begin

You need to have a Kubernetes cluster, and the kubectl command-line tool must be configured to communicate with your cluster. It is recommended to run this tutorial on a cluster with at least two nodes that are not acting as control plane hosts. If you do not already have a cluster, you can create one by using minikube or you can use one of these Kubernetes playgrounds:

- Katacoda
- Play with Kubernetes

Your cluster must be configured to use the CoreDNS addon or its precursor, kube-dns.

Your Kubernetes server must be at or later than version v1.6. To check the version, enter `kubectl version`.

### Create a simple Pod to use as a test environment

admin/dns/dnsutils.yaml

```
apiVersion: v1
kind: Pod
metadata:
  name: dnsutils
  namespace: default
spec:
  containers:
  - name: dnsutils
    image: gcr.io/kubernetes-e2e-test-images/dnsutils:1.3
    command:
      - sleep
      - "3600"
    imagePullPolicy: IfNotPresent
  restartPolicy: Always
```

**Note:** This example creates a pod in the `default` namespace. DNS name resolution for services depends on the namespace of the pod. For more information, review DNS for Services and Pods.

Use that manifest to create a Pod:

```
kubectl apply -f https://k8s.io/examples/admin/dns/dnsutils.yaml
```

```
pod/dnsutils created
```

...and verify its status:

```
kubectl get pods dnsutils
```

```
NAME      READY     STATUS    RESTARTS   AGE
dnsutils   1/1       Running   0          <some-time>
```

Once that Pod is running, you can exec `nslookup` in that environment. If you see something like the following, DNS is working correctly.

```
kubectl exec -i -t dnsutils -- nslookup kubernetes.default
```

```
Server:    10.0.0.10
Address 1: 10.0.0.10

Name:      kubernetes.default
Address 1: 10.0.0.1
```

If the `nslookup` command fails, check the following:

## Check the local DNS configuration first

Take a look inside the resolv.conf file. (See Inheriting DNS from the node and Known issues below for more information)

```
kubectl exec -ti dnsutils -- cat /etc/resolv.conf
```

Verify that the search path and name server are set up like the following (note that search path may vary for different cloud providers):

```
search default.svc.cluster.local svc.cluster.local cluster.local google.internal c.gce_project_id.
nameserver 10.0.0.10
options ndots:5
```

Errors such as the following indicate a problem with the CoreDNS (or kube-dns) add-on or with associated Services:

```
kubectl exec -i -t dnsutils -- nslookup kubernetes.default
```

```
Server:    10.0.0.10
Address 1: 10.0.0.10

nslookup: can't resolve 'kubernetes.default'
```

or

```
kubectl exec -i -t dnsutils -- nslookup kubernetes.default
```

```
Server:    10.0.0.10
Address 1: 10.0.0.10 kube-dns.kube-system.svc.cluster.local

nslookup: can't resolve 'kubernetes.default'
```

## Check if the DNS pod is running

Use the `kubectl get pods` command to verify that the DNS pod is running.

```
kubectl get pods --namespace=kube-system -l k8s-app=kube-dns
```

```
NAME                       READY     STATUS    RESTARTS   AGE
...
coredns-7b96bf9f76-5hsxb   1/1       Running   0          1h
coredns-7b96bf9f76-mvmmt   1/1       Running   0          1h
...
```

> **Note:** The value for label `k8s-app` is `kube-dns` for both CoreDNS and kube-dns deployments.

If you see that no CoreDNS Pod is running or that the Pod has failed/completed, the DNS add-on may not be deployed by default in your current environment and you will have to deploy it manually.

## Check for errors in the DNS pod

Use the `kubectl logs` command to see logs for the DNS containers.

For CoreDNS:

```
kubectl logs --namespace=kube-system -l k8s-app=kube-dns
```

Here is an example of a healthy CoreDNS log:

```
.:53
2018/08/15 14:37:17 [INFO] CoreDNS-1.2.2
2018/08/15 14:37:17 [INFO] linux/amd64, go1.10.3, 2e322f6
CoreDNS-1.2.2
linux/amd64, go1.10.3, 2e322f6
2018/08/15 14:37:17 [INFO] plugin/reload: Running configuration MD5 = 24e6c59e83ce706f07bcc82c31b1
```

See if there are any suspicious or unexpected messages in the logs.

## Is DNS service up?

Verify that the DNS service is up by using the `kubectl get service` command.

```
kubectl get svc --namespace=kube-system
```

```
NAME         TYPE        CLUSTER-IP     EXTERNAL-IP   PORT(S)             AGE
...
kube-dns     ClusterIP   10.0.0.10      <none>        53/UDP,53/TCP        1h
...
```

**Note:** The service name is `kube-dns` for both CoreDNS and kube-dns deployments.

If you have created the Service or in the case it should be created by default but it does not appear, see debugging Services for more information.

## Are DNS endpoints exposed?

You can verify that DNS endpoints are exposed by using the `kubectl get endpoints` command.

```
kubectl get endpoints kube-dns --namespace=kube-system
```

```
NAME       ENDPOINTS                       AGE
kube-dns   10.180.3.17:53,10.180.3.17:53    1h
```

If you do not see the endpoints, see the endpoints section in the debugging Services documentation.

For additional Kubernetes DNS examples, see the cluster-dns examples in the Kubernetes GitHub repository.

## Are DNS queries being received/processed?

You can verify if queries are being received by CoreDNS by adding the `log` plugin to the CoreDNS configuration (aka Corefile). The CoreDNS Corefile is held in a ConfigMap named `coredns`. To edit it, use the command:

```
kubectl -n kube-system edit configmap coredns
```

Then add `log` in the Corefile section per the example below:

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: coredns
  namespace: kube-system
data:
  Corefile: |
    .:53 {
        log
        errors
        health
        kubernetes cluster.local in-addr.arpa ip6.arpa {
          pods insecure
          upstream
          fallthrough in-addr.arpa ip6.arpa
        }
```

```
        prometheus :9153
        forward . /etc/resolv.conf
        cache 30
        loop
        reload
        loadbalance
    }
```

After saving the changes, it may take up to minute or two for Kubernetes to propagate these changes to the CoreDNS pods.

Next, make some queries and view the logs per the sections above in this document. If CoreDNS pods are receiving the queries, you should see them in the logs.

Here is an example of a query in the log:

```
.:53
2018/08/15 14:37:15 [INFO] CoreDNS-1.2.0
2018/08/15 14:37:15 [INFO] linux/amd64, go1.10.3, 2e322f6
CoreDNS-1.2.0
linux/amd64, go1.10.3, 2e322f6
2018/09/07 15:29:04 [INFO] plugin/reload: Running configuration MD5 = 162475cdf272d8aa601e6fe67a6a
2018/09/07 15:29:04 [INFO] Reloading complete
172.17.0.18:41675 - [07/Sep/2018:15:29:11 +0000] 59925 "A IN kubernetes.default.svc.cluster.local.
```

### Are you in the right namespace for the service?

DNS queries that don't specify a namespace are limited to the pod's namespace.

If the namespace of the pod and service differ, the DNS query must include the namespace of the service.

This query is limited to the pod's namespace:

```
kubectl exec -i -t dnsutils -- nslookup <service-name>
```

This query specifies the namespace:

```
kubectl exec -i -t dnsutils -- nslookup <service-name>.<namespace>
```

To learn more about name resolution, see DNS for Services and Pods.

## Known issues

Some Linux distributions (e.g. Ubuntu) use a local DNS resolver by default (systemd-resolved). Systemd-resolved moves and replaces `/etc/resolv.conf` with a stub file that can cause a fatal forwarding loop when resolving names in upstream servers. This can be fixed manually by using kubelet's `--resolv-conf` flag to point to the correct `resolv.conf` (With `systemd-resolved`, this is `/run/systemd/resolve/resolv.conf`). kubeadm automatically detects `systemd-resolved`, and adjusts the kubelet flags accordingly.

Kubernetes installs do not configure the nodes' `resolv.conf` files to use the cluster DNS by default, because that process is inherently distribution-specific. This should probably be implemented eventually.

Linux's libc (a.k.a. glibc) has a limit for the DNS `nameserver` records to 3 by default. What's more, for the glibc versions which are older than glibc-2.17-222 (the new versions update see this issue), the allowed number of DNS `search` records has been limited to 6 (see this bug from 2005). Kubernetes needs to consume 1 `nameserver` record and 3 `search` records. This means that if a local installation already uses 3 `nameserver`s or uses more than 3 `search`es while your glibc version is in the affected list, some of those settings will be lost. To work around the DNS `nameserver` records limit, the node can run `dnsmasq`, which will provide more `nameserver` entries. You can also use kubelet's `--resolv-conf` flag. To fix the DNS `search` records limit, consider upgrading your linux distribution or upgrading to an unaffected version of glibc.

If you are using Alpine version 3.3 or earlier as your base image, DNS may not work properly due to a known issue with Alpine. Kubernetes issue 30215 details more information on this.

## What's next

- See Autoscaling the DNS Service in a Cluster.
- Read DNS for Services and Pods

# 18 - Declare Network Policy

This document helps you get started using the Kubernetes NetworkPolicy API to declare network policies that govern how pods communicate with each other.

> **Caution:** This section links to third party projects that provide functionality required by Kubernetes. The Kubernetes project authors aren't responsible for these projects. This page follows CNCF website guidelines by listing projects alphabetically. To add a project to this list, read the content guide before submitting a change.

## Before you begin

You need to have a Kubernetes cluster, and the kubectl command-line tool must be configured to communicate with your cluster. It is recommended to run this tutorial on a cluster with at least two nodes that are not acting as control plane hosts. If you do not already have a cluster, you can create one by using minikube or you can use one of these Kubernetes playgrounds:

- Katacoda
- Play with Kubernetes

Your Kubernetes server must be at or later than version v1.8. To check the version, enter `kubectl version`.

Make sure you've configured a network provider with network policy support. There are a number of network providers that support NetworkPolicy, including:

- Antrea
- Calico
- Cilium
- Kube-router
- Romana
- Weave Net

## Create an `nginx` deployment and expose it via a service

To see how Kubernetes network policy works, start off by creating an `nginx` Deployment.

```
kubectl create deployment nginx --image=nginx
```

```
deployment.apps/nginx created
```

Expose the Deployment through a Service called `nginx`.

```
kubectl expose deployment nginx --port=80
```

```
service/nginx exposed
```

The above commands create a Deployment with an nginx Pod and expose the Deployment through a Service named `nginx`. The `nginx` Pod and Deployment are found in the `default` namespace.

```
kubectl get svc,pod
```

```
NAME                           CLUSTER-IP    EXTERNAL-IP   PORT(S)    AGE
service/kubernetes             10.100.0.1    <none>        443/TCP    46m
service/nginx                  10.100.0.16   <none>        80/TCP     33s

NAME                           READY         STATUS        RESTARTS   AGE
pod/nginx-701339712-e0qfq      1/1           Running       0          35s
```

## Test the service by accessing it from another Pod

You should be able to access the new `nginx` service from other Pods. To access the `nginx` Service from another Pod in the `default` namespace, start a busybox container:

```
kubectl run busybox --rm -ti --image=busybox -- /bin/sh
```

In your shell, run the following command:

```
wget --spider --timeout=1 nginx
```

```
Connecting to nginx (10.100.0.16:80)
remote file exists
```

## Limit access to the `nginx` service

To limit the access to the `nginx` service so that only Pods with the label `access: true` can query it, create a NetworkPolicy object as follows:

service/networking/nginx-policy.yaml

```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: access-nginx
spec:
  podSelector:
    matchLabels:
      app: nginx
  ingress:
  - from:
    - podSelector:
        matchLabels:
          access: "true"
```

The name of a NetworkPolicy object must be a valid DNS subdomain name.

> **Note:** NetworkPolicy includes a `podSelector` which selects the grouping of Pods to which the policy applies. You can see this policy selects Pods with the label `app=nginx`. The label was automatically added to the Pod in the `nginx` Deployment. An empty `podSelector` selects all pods in the namespace.

## Assign the policy to the service

Use kubectl to create a NetworkPolicy from the above `nginx-policy.yaml` file:

```
kubectl apply -f https://k8s.io/examples/service/networking/nginx-policy.yaml
```

```
networkpolicy.networking.k8s.io/access-nginx created
```

## Test access to the service when access label is not defined

When you attempt to access the `nginx` Service from a Pod without the correct labels, the request times out:

```
kubectl run busybox --rm -ti --image=busybox -- /bin/sh
```

In your shell, run the command:

```
wget --spider --timeout=1 nginx
```

```
Connecting to nginx (10.100.0.16:80)
wget: download timed out
```

# Define access label and test again

You can create a Pod with the correct labels to see that the request is allowed:

```
kubectl run busybox --rm -ti --labels="access=true" --image=busybox -- /bin/sh
```

In your shell, run the command:

```
wget --spider --timeout=1 nginx
```

```
Connecting to nginx (10.100.0.16:80)
remote file exists
```

# 19 - Developing Cloud Controller Manager

**FEATURE STATE:** `Kubernetes v1.11 [beta]`

The cloud-controller-manager is a Kubernetes control plane component that embeds cloud-specific control logic. The cloud controller manager lets you link your cluster into your cloud provider's API, and separates out the components that interact with that cloud platform from components that only interact with your cluster.

By decoupling the interoperability logic between Kubernetes and the underlying cloud infrastructure, the cloud-controller-manager component enables cloud providers to release features at a different pace compared to the main Kubernetes project.

## Background

Since cloud providers develop and release at a different pace compared to the Kubernetes project, abstracting the provider-specific code to the `cloud-controller-manager` binary allows cloud vendors to evolve independently from the core Kubernetes code.

The Kubernetes project provides skeleton cloud-controller-manager code with Go interfaces to allow you (or your cloud provider) to plug in your own implementations. This means that a cloud provider can implement a cloud-controller-manager by importing packages from Kubernetes core; each cloudprovider will register their own code by calling `cloudprovider.RegisterCloudProvider` to update a global variable of available cloud providers.

## Developing

### Out of tree

To build an out-of-tree cloud-controller-manager for your cloud:

1. Create a go package with an implementation that satisfies cloudprovider.Interface.
2. Use `main.go in cloud-controller-manager` from Kubernetes core as a template for your `main.go`. As mentioned above, the only difference should be the cloud package that will be imported.
3. Import your cloud package in `main.go`, ensure your package has an `init` block to run `cloudprovider.RegisterCloudProvider`.

Many cloud providers publish their controller manager code as open source. If you are creating a new cloud-controller-manager from scratch, you could take an existing out-of-tree cloud controller manager as your starting point.

### In tree

For in-tree cloud providers, you can run the in-tree cloud controller manager as a DaemonSet in your cluster. See Cloud Controller Manager Administration for more details.

# 20 - Enable Or Disable A Kubernetes API

This page shows how to enable or disable an API version from your cluster's control plane.

Specific API versions can be turned on or off by passing `--runtime-config=api/<version>` as a command line argument to the API server. The values for this argument are a comma-separated list of API versions. Later values override earlier values.

The `runtime-config` command line argument also supports 2 special keys:

- `api/all`, representing all known APIs
- `api/legacy`, representing only legacy APIs. Legacy APIs are any APIs that have been explicitly deprecated.

For example, to turning off all API versions except v1, pass `--runtime-config=api/all=false,api/v1=true` to the `kube-apiserver`.

## What's next

Read the full documentation for the `kube-apiserver` component.

# 21 - Enabling Service Topology

**FEATURE STATE:** `Kubernetes v1.21 [deprecated]`

This feature, specifically the alpha `topologyKeys` field, is deprecated since Kubernetes v1.21. Topology Aware Hints, introduced in Kubernetes v1.21, provide similar functionality.

*Service Topology* enables a Service to route traffic based upon the Node topology of the cluster. For example, a service can specify that traffic be preferentially routed to endpoints that are on the same Node as the client, or in the same availability zone.

## Before you begin

You need to have a Kubernetes cluster, and the kubectl command-line tool must be configured to communicate with your cluster. It is recommended to run this tutorial on a cluster with at least two nodes that are not acting as control plane hosts. If you do not already have a cluster, you can create one by using minikube or you can use one of these Kubernetes playgrounds:

- Katacoda
- Play with Kubernetes

Your Kubernetes server must be at or later than version 1.17. To check the version, enter `kubectl version`.

The following prerequisites are needed in order to enable topology aware service routing:

- Kubernetes v1.17 or later
- Configure kube-proxy to run in iptables mode or IPVS mode

## Enable Service Topology

**FEATURE STATE:** `Kubernetes v1.21 [deprecated]`

To enable service topology, enable the `ServiceTopology` feature gate for all Kubernetes components:

```
--feature-gates="ServiceTopology=true`
```

## What's next

- Read about Topology Aware Hints, the replacement for the `topologyKeys` field.
- Read about EndpointSlices
- Read about the Service Topology concept
- Read Connecting Applications with Services

# 22 - Enabling Topology Aware Hints

**FEATURE STATE:** `Kubernetes v1.21 [alpha]`

*Topology Aware Hints* enable topology aware routing with topology hints included in EndpointSlices. This approach tries to keep traffic close to where it originated from; you might do this to reduce costs, or to improve network performance.

## Before you begin

You need to have a Kubernetes cluster, and the kubectl command-line tool must be configured to communicate with your cluster. It is recommended to run this tutorial on a cluster with at least two nodes that are not acting as control plane hosts. If you do not already have a cluster, you can create one by using minikube or you can use one of these Kubernetes playgrounds:

- Katacoda
- Play with Kubernetes

Your Kubernetes server must be at or later than version 1.21. To check the version, enter `kubectl version`.

The following prerequisite is needed in order to enable topology aware hints:

- Configure the kube-proxy to run in iptables mode or IPVS mode
- Ensure that you have not disabled EndpointSlices

## Enable Topology Aware Hints

To enable service topology hints, enable the `TopologyAwareHints` feature gate for the kube-apiserver, kube-controller-manager, and kube-proxy:

```
--feature-gates="TopologyAwareHints=true"
```

## What's next

- Read about Topology Aware Hints for Services
- Read Connecting Applications with Services

# 23 - Encrypting Secret Data at Rest

This page shows how to enable and configure encryption of secret data at rest.

## Before you begin

- You need to have a Kubernetes cluster, and the kubectl command-line tool must be configured to communicate with your cluster. It is recommended to run this tutorial on a cluster with at least two nodes that are not acting as control plane hosts. If you do not already have a cluster, you can create one by using minikube or you can use one of these Kubernetes playgrounds:
  - Katacoda
  - Play with Kubernetes

  Your Kubernetes server must be at or later than version 1.13. To check the version, enter `kubectl version`.
- etcd v3.0 or later is required

## Configuration and determining whether encryption at rest is already enabled

The `kube-apiserver` process accepts an argument `--encryption-provider-config` that controls how API data is encrypted in etcd. An example configuration is provided below.

## Understanding the encryption at rest configuration.

```yaml
apiVersion: apiserver.config.k8s.io/v1
kind: EncryptionConfiguration
resources:
  - resources:
    - secrets
    providers:
    - identity: {}
    - aesgcm:
        keys:
        - name: key1
          secret: c2VjcmV0IGlzIHNlY3VyZQ==
        - name: key2
          secret: dGhpcyBpcyBwYXNzd29yZA==
    - aescbc:
        keys:
        - name: key1
          secret: c2VjcmV0IGlzIHNlY3VyZQ==
        - name: key2
          secret: dGhpcyBpcyBwYXNzd29yZA==
    - secretbox:
        keys:
        - name: key1
          secret: YWJjZGVmZ2hpamtsbW5vcHFyc3R1dnd4eXoxMjM0NTY=
```

Each `resources` array item is a separate config and contains a complete configuration. The `resources.resources` field is an array of Kubernetes resource names ( `resource` or `resource.group` ) that should be encrypted. The `providers` array is an ordered list of the possible encryption providers. Only one provider type may be specified per entry ( `identity` or `aescbc` may be provided, but not both in the same item).

The first provider in the list is used to encrypt resources going into storage. When reading resources from storage each provider that matches the stored data attempts to decrypt the data in order. If no provider can read the stored data due to a mismatch in format or secret key, an error is returned which prevents clients from accessing that resource.

> **Caution: IMPORTANT:** If any resource is not readable via the encryption config (because keys were changed), the only recourse is to delete that key from the underlying etcd directly. Calls that attempt to read that resource will fail until it is deleted or a valid decryption key is provided.

### Providers:

| Name | Encryption | Strength | Speed | Key Length | Other Considerations |
|---|---|---|---|---|---|

| Name | Encryption | Strength | Speed | Key Length | Other Considerations |
|---|---|---|---|---|---|
| `identity` | None | N/A | N/A | N/A | Resources written as-is without encryption. When set as the first provider, the resource will be decrypted as new values are written. |
| `aescbc` | AES-CBC with PKCS#7 padding | Strongest | Fast | 32-byte | The recommended choice for encryption at rest but may be slightly slower than `secretbox`. |
| `secretbox` | XSalsa20 and Poly1305 | Strong | Faster | 32-byte | A newer standard and may not be considered acceptable in environments that require high levels of review. |
| `aesgcm` | AES-GCM with random nonce | Must be rotated every 200k writes | Fastest | 16, 24, or 32-byte | Is not recommended for use except when an automated key rotation scheme is implemented. |
| `kms` | Uses envelope encryption scheme: Data is encrypted by data encryption keys (DEKs) using AES-CBC with PKCS#7 padding, DEKs are encrypted by key encryption keys (KEKs) according to configuration in Key Management Service (KMS) | Strongest | Fast | 32-bytes | The recommended choice for using a third party tool for key management. Simplifies key rotation, with a new DEK generated for each encryption, and KEK rotation controlled by the user. Configure the KMS provider |

Each provider supports multiple keys - the keys are tried in order for decryption, and if the provider is the first provider, the first key is used for encryption.

**Storing the raw encryption key in the EncryptionConfig only moderately improves your security posture, compared to no encryption. Please use `kms` provider for additional security.** By default, the `identity` provider is used to protect secrets in etcd, which provides no encryption. `EncryptionConfiguration` was introduced to encrypt secrets locally, with a locally managed key.

Encrypting secrets with a locally managed key protects against an etcd compromise, but it fails to protect against a host compromise. Since the encryption keys are stored on the host in the EncryptionConfig YAML file, a skilled attacker can access that file and extract the encryption keys.

Envelope encryption creates dependence on a separate key, not stored in Kubernetes. In this case, an attacker would need to compromise etcd, the kubeapi-server, and the third-party KMS provider to retrieve the plaintext values, providing a higher level of security than locally-stored encryption keys.

## Encrypting your data

Create a new encryption config file:

```
apiVersion: apiserver.config.k8s.io/v1
kind: EncryptionConfiguration
resources:
  - resources:
    - secrets
    providers:
    - aescbc:
        keys:
```

```
        - name: key1
          secret: <BASE 64 ENCODED SECRET>
    - identity: {}
```

To create a new secret perform the following steps:

1. Generate a 32 byte random key and base64 encode it. If you're on Linux or macOS, run the following command:

   ```
   head -c 32 /dev/urandom | base64
   ```

2. Place that value in the secret field.
3. Set the `--encryption-provider-config` flag on the `kube-apiserver` to point to the location of the config file.
4. Restart your API server.

**Caution:** Your config file contains keys that can decrypt content in etcd, so you must properly restrict permissions on your masters so only the user who runs the kube-apiserver can read it.

## Verifying that data is encrypted

Data is encrypted when written to etcd. After restarting your `kube-apiserver`, any newly created or updated secret should be encrypted when stored. To check, you can use the `etcdctl` command line program to retrieve the contents of your secret.

1. Create a new secret called `secret1` in the `default` namespace:

   ```
   kubectl create secret generic secret1 -n default --from-literal=mykey=mydata
   ```

2. Using the etcdctl commandline, read that secret out of etcd:

   `ETCDCTL_API=3 etcdctl get /registry/secrets/default/secret1 [...] | hexdump -C`

   where `[...]` must be the additional arguments for connecting to the etcd server.

3. Verify the stored secret is prefixed with `k8s:enc:aescbc:v1:` which indicates the `aescbc` provider has encrypted the resulting data.

4. Verify the secret is correctly decrypted when retrieved via the API:

   ```
   kubectl describe secret secret1 -n default
   ```

   should match `mykey: bXlkYXRh`, mydata is encoded, check decoding a secret to completely decode the secret.

## Ensure all secrets are encrypted

Since secrets are encrypted on write, performing an update on a secret will encrypt that content.

```
kubectl get secrets --all-namespaces -o json | kubectl replace -f -
```

The command above reads all secrets and then updates them to apply server side encryption.

**Note:** If an error occurs due to a conflicting write, retry the command. For larger clusters, you may wish to subdivide the secrets by namespace or script an update.

## Rotating a decryption key

Changing the secret without incurring downtime requires a multi step operation, especially in the presence of a highly available deployment where multiple `kube-apiserver` processes are running.

1. Generate a new key and add it as the second key entry for the current provider on all servers
2. Restart all `kube-apiserver` processes to ensure each server can decrypt using the new key
3. Make the new key the first entry in the `keys` array so that it is used for encryption in the config
4. Restart all `kube-apiserver` processes to ensure each server now encrypts using the new key
5. Run `kubectl get secrets --all-namespaces -o json | kubectl replace -f -` to encrypt all existing secrets with the new key
6. Remove the old decryption key from the config after you back up etcd with the new key in use and update all secrets

With a single `kube-apiserver`, step 2 may be skipped.

# Decrypting all data

To disable encryption at rest place the `identity` provider as the first entry in the config:

```
apiVersion: apiserver.config.k8s.io/v1
kind: EncryptionConfiguration
resources:
  - resources:
    - secrets
    providers:
    - identity: {}
    - aescbc:
        keys:
        - name: key1
          secret: <BASE 64 ENCODED SECRET>
```

and restart all `kube-apiserver` processes. Then run:

```
kubectl get secrets --all-namespaces -o json | kubectl replace -f -
```

to force all secrets to be decrypted.

# 24 - Guaranteed Scheduling For Critical Add-On Pods

Kubernetes core components such as the API server, scheduler, and controller-manager run on a control plane node. However, add-ons must run on a regular cluster node. Some of these add-ons are critical to a fully functional cluster, such as metrics-server, DNS, and UI. A cluster may stop working properly if a critical add-on is evicted (either manually or as a side effect of another operation like upgrade) and becomes pending (for example when the cluster is highly utilized and either there are other pending pods that schedule into the space vacated by the evicted critical add-on pod or the amount of resources available on the node changed for some other reason).

Note that marking a pod as critical is not meant to prevent evictions entirely; it only prevents the pod from becoming permanently unavailable. A static pod marked as critical, can't be evicted. However, a non-static pods marked as critical are always rescheduled.

## Marking pod as critical

To mark a Pod as critical, set priorityClassName for that Pod to `system-cluster-critical` or `system-node-critical`. `system-node-critical` is the highest available priority, even higher than `system-cluster-critical`.

# 25 - IP Masquerade Agent User Guide

This page shows how to configure and enable the ip-masq-agent.

## Before you begin

You need to have a Kubernetes cluster, and the kubectl command-line tool must be configured to communicate with your cluster. It is recommended to run this tutorial on a cluster with at least two nodes that are not acting as control plane hosts. If you do not already have a cluster, you can create one by using minikube or you can use one of these Kubernetes playgrounds:

- Katacoda
- Play with Kubernetes

To check the version, enter `kubectl version`.

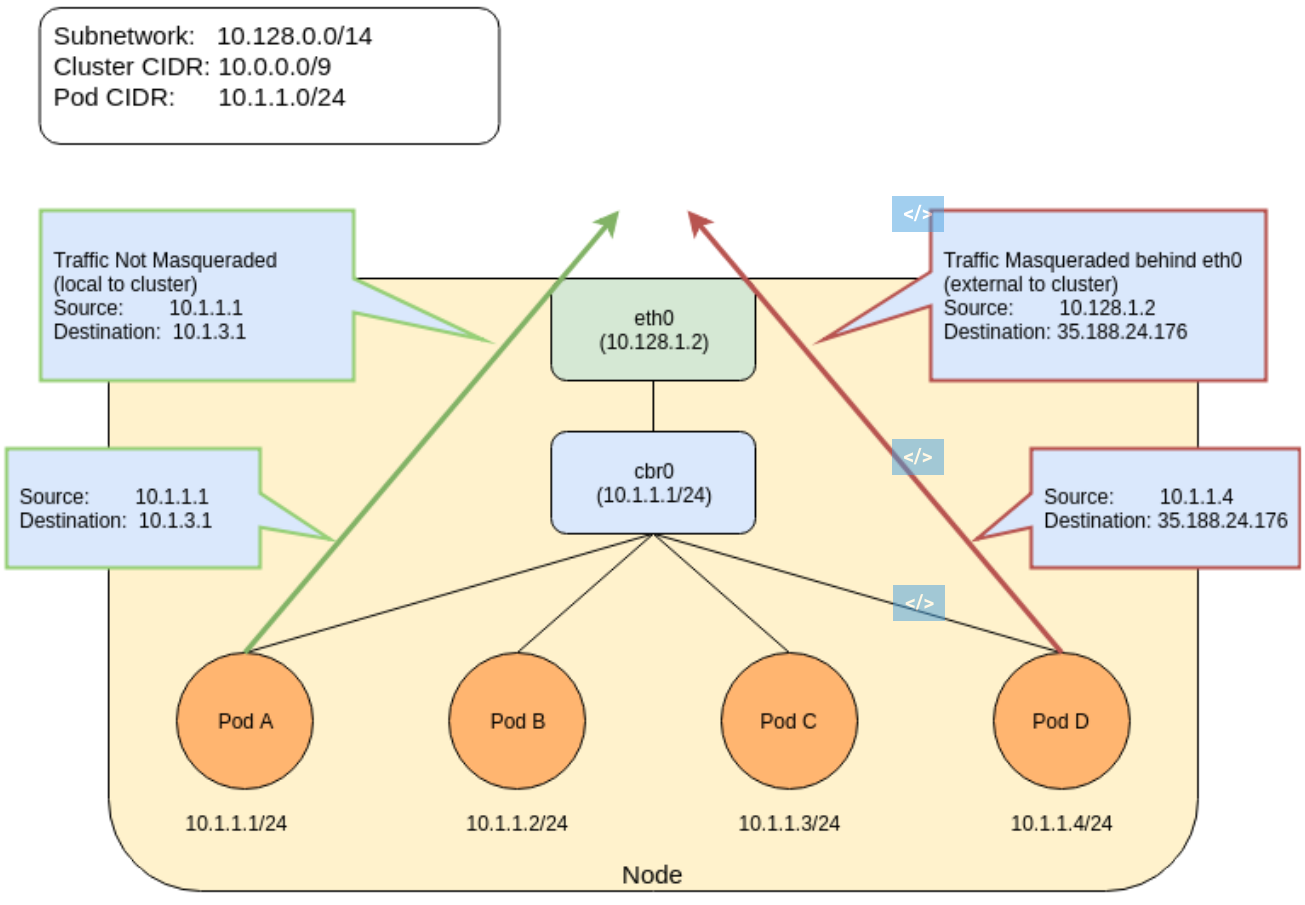
## IP Masquerade Agent User Guide

The ip-masq-agent configures iptables rules to hide a pod's IP address behind the cluster node's IP address. This is typically done when sending traffic to destinations outside the cluster's pod CIDR range.

### Key Terms

- **NAT (Network Address Translation)** Is a method of remapping one IP address to another by modifying either the source and/or destination address information in the IP header. Typically performed by a device doing IP routing.
- **Masquerading** A form of NAT that is typically used to perform a many to one address translation, where multiple source IP addresses are masked behind a single address, which is typically the device doing the IP routing. In Kubernetes this is the Node's IP address.
- **CIDR (Classless Inter-Domain Routing)** Based on the variable-length subnet masking, allows specifying arbitrary-length prefixes. CIDR introduced a new method of representation for IP addresses, now commonly known as **CIDR notation**, in which an address or routing prefix is written with a suffix indicating the number of bits of the prefix, such as 192.168.2.0/24.
- **Link Local** A link-local address is a network address that is valid only for communications within the network segment or the broadcast domain that the host is connected to. Link-local addresses for IPv4 are defined in the address block 169.254.0.0/16 in CIDR notation.

The ip-masq-agent configures iptables rules to handle masquerading node/pod IP addresses when sending traffic to destinations outside the cluster node's IP and the Cluster IP range. This essentially hides pod IP addresses behind the cluster node's IP address. In some environments, traffic to "external" addresses must come from a known machine address. For example, in Google Cloud, any traffic to the internet must come from a VM's IP. When containers are used, as in Google Kubernetes Engine, the Pod IP will be rejected for egress. To avoid this, we must hide the Pod IP behind the VM's own IP address - generally known as "masquerade". By default, the agent is configured to treat the three private IP ranges specified by RFC 1918 as non-masquerade CIDR. These ranges are 10.0.0.0/8, 172.16.0.0/12, and 192.168.0.0/16. The agent will also treat link-local (169.254.0.0/16) as a non-masquerade CIDR by default. The agent is configured to reload its configuration from the location */etc/config/ip-masq-agent* every 60 seconds, which is also configurable.



The agent configuration file must be written in YAML or JSON syntax, and may contain three optional keys:

- **nonMasqueradeCIDRs:** A list of strings in CIDR notation that specify the non-masquerade ranges.
- **masqLinkLocal:** A Boolean (true / false) which indicates whether to masquerade traffic to the link local prefix 169.254.0.0/16. False by default.
- **resyncInterval:** A time interval at which the agent attempts to reload config from disk. For example: '30s', where 's' means seconds, 'ms' means milliseconds, etc...

Traffic to 10.0.0.0/8, 172.16.0.0/12 and 192.168.0.0/16) ranges will NOT be masqueraded. Any other traffic (assumed to be internet) will be masqueraded. An example of a local destination from a pod could be its Node's IP address as well as another node's address or one of the IP addresses in Cluster's IP range. Any other traffic will be masqueraded by default. The below entries show the default set of rules that are applied by the ip-masq-agent:

```
iptables -t nat -L IP-MASQ-AGENT
RETURN     all  --  anywhere             169.254.0.0/16       /* ip-masq-agent: cluster-local traf
RETURN     all  --  anywhere             10.0.0.0/8           /* ip-masq-agent: cluster-local traf
RETURN     all  --  anywhere             172.16.0.0/12        /* ip-masq-agent: cluster-local traf
RETURN     all  --  anywhere             192.168.0.0/16       /* ip-masq-agent: cluster-local traf
MASQUERADE  all  --  anywhere              anywhere             /* ip-masq-agent: outbound traffic
```

By default, in GCE/Google Kubernetes Engine starting with Kubernetes version 1.7.0, if network policy is enabled or you are using a cluster CIDR not in the 10.0.0.0/8 range, the ip-masq-agent will run in your cluster. If you are running in another environment, you can add the ip-masq-agent DaemonSet to your cluster:

## Create an ip-masq-agent

To create an ip-masq-agent, run the following kubectl command:

`kubectl apply -f https://raw.githubusercontent.com/kubernetes-sigs/ip-masq-agent/master/ip-masq-agent.yaml`

You must also apply the appropriate node label to any nodes in your cluster that you want the agent to run on.

`kubectl label nodes my-node beta.kubernetes.io/masq-agent-ds-ready=true`

More information can be found in the ip-masq-agent documentation here

In most cases, the default set of rules should be sufficient; however, if this is not the case for your cluster, you can create and apply a ConfigMap to customize the IP ranges that are affected. For example, to allow only 10.0.0.0/8 to be considered by the ip-masq-agent, you can create the following ConfigMap in a file called "config".

> **Note:**
> It is important that the file is called config since, by default, that will be used as the key for lookup by the ip-masq-agent:
>
> ```
> nonMasqueradeCIDRs:
>   - 10.0.0.0/8
> resyncInterval: 60s
> ```

Run the following command to add the config map to your cluster:

```
kubectl create configmap ip-masq-agent --from-file=config --namespace=kube-system
```

This will update a file located at */etc/config/ip-masq-agent* which is periodically checked every *resyncInterval* and applied to the cluster node. After the resync interval has expired, you should see the iptables rules reflect your changes:

```
iptables -t nat -L IP-MASQ-AGENT
Chain IP-MASQ-AGENT (1 references)
target     prot opt source               destination
RETURN     all  --  anywhere             169.254.0.0/16       /* ip-masq-agent: cluster-local traf
RETURN     all  --  anywhere             10.0.0.0/8           /* ip-masq-agent: cluster-local
MASQUERADE  all  --  anywhere              anywhere             /* ip-masq-agent: outbound traffic
```

By default, the link local range (169.254.0.0/16) is also handled by the ip-masq agent, which sets up the appropriate iptables rules. To have the ip-masq-agent ignore link local, you can set *masqLinkLocal* to true in the config map.

```
nonMasqueradeCIDRs:
  - 10.0.0.0/8
resyncInterval: 60s
masqLinkLocal: true
```

# 26 - Limit Storage Consumption

This example demonstrates how to limit the amount of storage consumed in a namespace.

The following resources are used in the demonstration: ResourceQuota, LimitRange, and PersistentVolumeClaim.

## Before you begin

- You need to have a Kubernetes cluster, and the kubectl command-line tool must be configured to communicate with your cluster. It is recommended to run this tutorial on a cluster with at least two nodes that are not acting as control plane hosts. If you do not already have a cluster, you can create one by using minikube or you can use one of these Kubernetes playgrounds:
  - Katacoda
  - Play with Kubernetes

  To check the version, enter `kubectl version`.

## Scenario: Limiting Storage Consumption

The cluster-admin is operating a cluster on behalf of a user population and the admin wants to control how much storage a single namespace can consume in order to control cost.

The admin would like to limit:

1. The number of persistent volume claims in a namespace
2. The amount of storage each claim can request
3. The amount of cumulative storage the namespace can have

## LimitRange to limit requests for storage

Adding a `LimitRange` to a namespace enforces storage request sizes to a minimum and maximum. Storage is requested via `PersistentVolumeClaim`. The admission controller that enforces limit ranges will reject any PVC that is above or below the values set by the admin.

In this example, a PVC requesting 10Gi of storage would be rejected because it exceeds the 2Gi max.

```
apiVersion: v1
kind: LimitRange
metadata:
  name: storagelimits
spec:
  limits:
  - type: PersistentVolumeClaim
    max:
      storage: 2Gi
    min:
      storage: 1Gi
```

Minimum storage requests are used when the underlying storage provider requires certain minimums. For example, AWS EBS volumes have a 1Gi minimum requirement.

## StorageQuota to limit PVC count and cumulative storage capacity

Admins can limit the number of PVCs in a namespace as well as the cumulative capacity of those PVCs. New PVCs that exceed either maximum value will be rejected.

In this example, a 6th PVC in the namespace would be rejected because it exceeds the maximum count of 5. Alternatively, a 5Gi maximum quota when combined with the 2Gi max limit above, cannot have 3 PVCs where each has 2Gi. That would be 6Gi requested for a namespace capped at 5Gi.

```
apiVersion: v1
kind: ResourceQuota
metadata:
  name: storagequota
spec:
  hard:
```

```
    persistentvolumeclaims: "5"
    requests.storage: "5Gi"
```

## Summary

A limit range can put a ceiling on how much storage is requested while a resource quota can effectively cap the storage consumed by a namespace through claim counts and cumulative storage capacity. The allows a cluster-admin to plan their cluster's storage budget without risk of any one project going over their allotment.

# 27 - Memory Manager

**FEATURE STATE:** `Kubernetes v1.21 [alpha]`

The Kubernetes *Memory Manager* enables the feature of guaranteed memory (and hugepages) allocation for pods in the `Guaranteed` QoS class.

The Memory Manager employs hint generation protocol to yield the most suitable NUMA affinity for a pod. The Memory Manager feeds the central manager (*Topology Manager*) with these affinity hints. Based on both the hints and Topology Manager policy, the pod is rejected or admitted to the node.

Moreover, the Memory Manager ensures that the memory which a pod requests is allocated from a minimum number of NUMA nodes.

The Memory Manager is only pertinent to Linux based hosts.

## Before you begin

You need to have a Kubernetes cluster, and the kubectl command-line tool must be configured to communicate with your cluster. It is recommended to run this tutorial on a cluster with at least two nodes that are not acting as control plane hosts. If you do not already have a cluster, you can create one by using minikube or you can use one of these Kubernetes playgrounds:

- Katacoda
- Play with Kubernetes

Your Kubernetes server must be version v1.21. To check the version, enter `kubectl version`.

To align memory resources with other requested resources in a Pod Spec:

- the CPU Manager should be enabled and proper CPU Manager policy should be configured on a Node. See control CPU Management Policies;
- the Topology Manager should be enabled and proper Topology Manager policy should be configured on a Node. See control Topology Management Policies.

Support for the Memory Manager requires `MemoryManager` feature gate to be enabled.

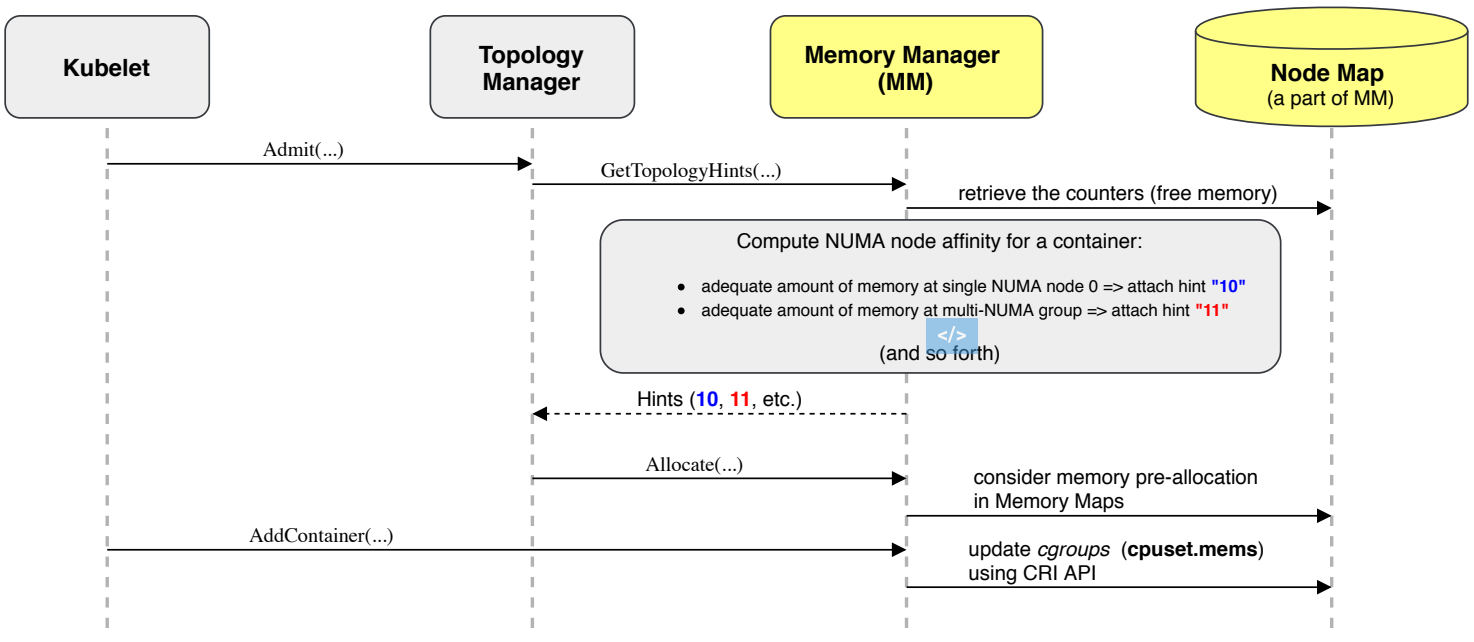
That is, the `kubelet` must be started with the following flag:

```
--feature-gates=MemoryManager=true
```

## How Memory Manager Operates?

The Memory Manager currently offers the guaranteed memory (and hugepages) allocation for Pods in Guaranteed QoS class. To immediately put the Memory Manager into operation follow the guidelines in the section Memory Manager configuration, and subsequently, prepare and deploy a `Guaranteed` pod as illustrated in the section Placing a Pod in the Guaranteed QoS class.

The Memory Manager is a Hint Provider, and it provides topology hints for the Topology Manager which then aligns the requested resources according to these topology hints. It also enforces `cgroups` (i.e. `cpuset.mems`) for pods. The complete flow diagram concerning pod admission and deployment process is illustrated in Memory Manager KEP: Design Overview and below:




During this process, the Memory Manager updates its internal counters stored in Node Map and Memory Maps to manage guaranteed memory allocation.

The Memory Manager updates the Node Map during the startup and runtime as follows.

### Startup

This occurs once a node administrator employs `--reserved-memory` (section Reserved memory flag). In this case, the Node Map becomes updated to reflect this reservation as illustrated in Memory Manager KEP: Memory Maps at start-up (with examples).

The administrator must provide `--reserved-memory` flag when `Static` policy is configured.

### Runtime

Reference Memory Manager KEP: Memory Maps at runtime (with examples) illustrates how a successful pod deployment affects the Node Map, and it also relates to how potential Out-of-Memory (OOM) situations are handled further by Kubernetes or operating system.

Important topic in the context of Memory Manager operation is the management of NUMA groups. Each time pod's memory request is in excess of single NUMA node capacity, the Memory Manager attempts to create a group that comprises several NUMA nodes and features extend memory capacity. The problem has been solved as elaborated in Memory Manager KEP: How to enable the guaranteed memory allocation over many NUMA nodes?. Also, reference Memory Manager KEP: Simulation - how the Memory Manager works? (by examples) illustrates how the management of groups occurs.

## Memory Manager configuration

Other Managers should be first pre-configured (section Pre-configuration). Next, the Memory Manger feature should be enabled (section Enable the Memory Manager feature) and be run with `Static` policy (section Static policy). Optionally, some amount of memory can be reserved for system or kubelet processes to increase node stability (section Reserved memory flag).

### Policies

Memory Manager supports two policies. You can select a policy via a `kubelet` flag `--memory-manager-policy`.

Two policies can be selected:

- `None` (default)
- `Static`

### None policy

This is the default policy and does not affect the memory allocation in any way. It acts the same as if the Memory Manager is not present at all.

The `None` policy returns default topology hint. This special hint denotes that Hint Provider (Memory Manger in this case) has no preference for NUMA affinity with any resource.

### Static policy

In the case of the `Guaranteed` pod, the `Static` Memory Manger policy returns topology hints relating to the set of NUMA nodes where the memory can be guaranteed, and reserves the memory through updating the internal NodeMap object.

In the case of the `BestEffort` or `Burstable` pod, the `Static` Memory Manager policy sends back the default topology hint as there is no request for the guaranteed memory, and does not reserve the memory in the internal NodeMap object.

### Reserved memory flag

The Node Allocatable mechanism is commonly used by node administrators to reserve K8S node system resources for the kubelet or operating system processes in order to enhance the node stability. A dedicated set of flags can be used for this purpose to set the total amount of reserved memory for a node. This pre-configured value is subsequently utilized to calculate the real amount of node's "allocatable" memory available to pods.

The Kubernetes scheduler incorporates "allocatable" to optimise pod scheduling process. The foregoing flags include `--kube-reserved`, `--system-reserved` and `--eviction-threshold`. The sum of their values will account for the total amount of reserved memory.

A new `--reserved-memory` flag was added to Memory Manager to allow for this total reserved memory to be split (by a node administrator) and accordingly reserved across many NUMA nodes.

The flag specifies a comma-separated list of memory reservations per NUMA node. This parameter is only useful in the context of the Memory Manager feature. The Memory Manager will not use this reserved memory for the allocation of container workloads.

For example, if you have a NUMA node "NUMA0" with `10Gi` of memory available, and the `--reserved-memory` was specified to reserve `1Gi` of memory at "NUMA0", the Memory Manager assumes that only `9Gi` is available for containers.

You can omit this parameter, however, you should be aware that the quantity of reserved memory from all NUMA nodes should be equal to the quantity of memory specified by the Node Allocatable feature. If at least one node allocatable parameter is non-zero, you will need to specify `--reserved-memory` for at least one NUMA node. In fact, `eviction-hard` threshold value is equal to `100Mi` by default, so if `Static` policy is used, `--reserved-memory` is obligatory.

Also, avoid the following configurations:

1. duplicates, i.e. the same NUMA node or memory type, but with a different value;
2. setting zero limit for any of memory types;
3. NUMA node IDs that do not exist in the machine hardware;
4. memory type names different than `memory` or `hugepages-<size>` (hugepages of particular `<size>` should also exist).

Syntax:

```
--reserved-memory N:memory-type1=value1,memory-type2=value2,...
```

- `N` (integer) - NUMA node index, e.g. `0`
- `memory-type` (string) - represents memory type:
  - `memory` - conventional memory
  - `hugepages-2Mi` or `hugepages-1Gi` - hugepages
- `value` (string) - the quantity of reserved memory, e.g. `1Gi`

Example usage:

```
--reserved-memory 0:memory=1Gi,hugepages-1Gi=2Gi
```

or

```
--reserved-memory 0:memory=1Gi --reserved-memory 1:memory=2Gi
```

When you specify values for `--reserved-memory` flag, you must comply with the setting that you prior provided via Node Allocatable Feature flags. That is, the following rule must be obeyed for each memory type:

```
sum(reserved-memory(i)) = kube-reserved + system-reserved + eviction-threshold,
```

where `i` is an index of a NUMA node.

If you do not follow the formula above, the Memory Manager will show an error on startup.

In other words, the example above illustrates that for the conventional memory (`type=memory`), we reserve `3Gi` in total, i.e.:

```
sum(reserved-memory(i)) = reserved-memory(0) + reserved-memory(1) = 1Gi + 2Gi = 3Gi
```

An example of kubelet command-line arguments relevant to the node Allocatable configuration:

- `--kube-reserved=cpu=500m,memory=50Mi`
- `--system-reserved=cpu=123m,memory=333Mi`
- `--eviction-hard=memory.available<500Mi`

> **Note:** The default hard eviction threshold is 100MiB, and **not** zero. Remember to increase the quantity of memory that you reserve by setting `--reserved-memory` by that hard eviction threshold. Otherwise, the kubelet will not start Memory Manager and display an error.

Here is an example of a correct configuration:

```
--feature-gates=MemoryManager=true
--kube-reserved=cpu=4,memory=4Gi
--system-reserved=cpu=1,memory=1Gi
--memory-manager-policy=Static
--reserved-memory 0:memory=3Gi --reserved-memory 1:memory=2148Mi
```

Let us validate the configuration above:

1. `kube-reserved + system-reserved + eviction-hard(default) = reserved-memory(0) + reserved-memory(1)`
2. `4GiB + 1GiB + 100MiB = 3GiB + 2148MiB`
3. `5120MiB + 100MiB = 3072MiB + 2148MiB`
4. `5220MiB = 5220MiB` (which is correct)

# Placing a Pod in the Guaranteed QoS class

If the selected policy is anything other than `None`, the Memory Manager identifies pods that are in the `Guaranteed` QoS class. The Memory Manager provides specific topology hints to the Topology Manager for each `Guaranteed` pod. For pods in a QoS class other than `Guaranteed`, the Memory Manager provides default topology hints to the Topology Manager.

The following excerpts from pod manifests assign a pod to the `Guaranteed` QoS class.

Pod with integer CPU(s) runs in the `Guaranteed` QoS class, when `requests` are equal to `limits`:

```
spec:
  containers:
  - name: nginx
    image: nginx
    resources:
      limits:
        memory: "200Mi"
        cpu: "2"
        example.com/device: "1"
      requests:
        memory: "200Mi"
        cpu: "2"
        example.com/device: "1"
```

Also, a pod sharing CPU(s) runs in the `Guaranteed` QoS class, when `requests` are equal to `limits`.

```
spec:
  containers:
  - name: nginx
    image: nginx
    resources:
      limits:
        memory: "200Mi"
        cpu: "300m"
        example.com/device: "1"
      requests:
        memory: "200Mi"
        cpu: "300m"
        example.com/device: "1"
```

Notice that both CPU and memory requests must be specified for a Pod to lend it to Guaranteed QoS class.

# Troubleshooting

The following means can be used to troubleshoot the reason why a pod could not be deployed or became rejected at a node:

- pod status - indicates topology affinity errors
- system logs - include valuable information for debugging, e.g., about generated hints
- state file - the dump of internal state of the Memory Manager (includes Node Map and Memory Maps)

## Pod status (TopologyAffinityError)

This error typically occurs in the following situations:

- a node has not enough resources available to satisfy the pod's request
- the pod's request is rejected due to particular Topology Manager policy constraints

The error appears in the status of a pod:

```
# kubectl get pods
NAME         READY   STATUS                    RESTARTS   AGE
guaranteed   0/1     TopologyAffinityError     0          113s
```

Use `kubectl describe pod <id>` or `kubectl get events` to obtain detailed error message:

```
Warning  TopologyAffinityError  10m   kubelet, dell8  Resources cannot be allocated with Topology
```

## System logs

Search system logs with respect to a particular pod.

The set of hints that Memory Manager generated for the pod can be found in the logs. Also, the set of hints generated by CPU Manager should be present in the logs.

Topology Manager merges these hints to calculate a single best hint. The best hint should be also present in the logs.

The best hint indicates where to allocate all the resources. Topology Manager tests this hint against its current policy, and based on the verdict, it either admits the pod to the node or rejects it.

Also, search the logs for occurrences associated with the Memory Manager, e.g. to find out information about `cgroups` and `cpuset.mems` updates.

## Examine the memory manager state on a node

Let us first deploy a sample `Guaranteed` pod whose specification is as follows:

```
apiVersion: v1
kind: Pod
metadata:
  name: guaranteed
spec:
  containers:
  - name: guaranteed
    image: consumer
    imagePullPolicy: Never
    resources:
      limits:
        cpu: "2"
        memory: 150Gi
      requests:
        cpu: "2"
        memory: 150Gi
    command: ["sleep","infinity"]
```

Next, let us log into the node where it was deployed and examine the state file in `/var/lib/kubelet/memory_manager_state`:

```
{
   "policyName":"Static",
   "machineState":{
      "0":{
         "numberOfAssignments":1,
         "memoryMap":{
            "hugepages-1Gi":{
               "total":0,
               "systemReserved":0,
               "allocatable":0,
               "reserved":0,
               "free":0
            },
            "memory":{
               "total":134987354112,
               "systemReserved":3221225472,
               "allocatable":131766128640,
               "reserved":131766128640,
               "free":0
            }
         },
         "nodes":[
            0,
            1
         ]
      },
      "1":{
         "numberOfAssignments":1,
         "memoryMap":{
            "hugepages-1Gi":{
               "total":0,
               "systemReserved":0,
               "allocatable":0,
               "reserved":0,
               "free":0
            },
            "memory":{
               "total":135286722560,
               "systemReserved":2252341248,
               "allocatable":133034381312,
```

```
                "reserved":29295144960,
                "free":103739236352
             }
          },
          "nodes":[
             0,
             1
          ]
       }
    },
    "entries":{
       "fa9bdd38-6df9-4cf9-aa67-8c4814da37a8":{
          "guaranteed":[
             {
                "numaAffinity":[
                   0,
                   1
                ],
                "type":"memory",
                "size":161061273600
             }
          ]
       }
    },
    "checksum":4142013182
}
```


It can be deduced from the state file that the pod was pinned to both NUMA nodes, i.e.:


```
"numaAffinity":[
   0,
   1
],
```


Pinned term means that pod's memory consumption is constrained (through `cgroups` configuration) to these NUMA nodes.

This automatically implies that Memory Manager instantiated a new group that comprises these two NUMA nodes, i.e. `0` and `1` indexed NUMA nodes.

Notice that the management of groups is handled in a relatively complex manner, and further elaboration is provided in Memory Manager KEP in this and this sections.

In order to analyse memory resources available in a group, the corresponding entries from NUMA nodes belonging to the group must be added up.

For example, the total amount of free "conventional" memory in the group can be computed by adding up the free memory available at every NUMA node in the group, i.e., in the `"memory"` section of NUMA node `0` ( `"free":0` ) and NUMA node `1` ( `"free":103739236352` ). So, the total amount of free "conventional" memory in this group is equal to `0 + 103739236352` bytes.

The line `"systemReserved":3221225472` indicates that the administrator of this node reserved `3221225472` bytes (i.e. `3Gi` ) to serve kubelet and system processes at NUMA node `0` , by using `--reserved-memory` flag.

## What's next

- Memory Manager KEP: Design Overview
- Memory Manager KEP: Memory Maps at start-up (with examples)
- Memory Manager KEP: Memory Maps at runtime (with examples)
- Memory Manager KEP: Simulation - how the Memory Manager works? (by examples)
- Memory Manager KEP: The Concept of Node Map and Memory Maps
- Memory Manager KEP: How to enable the guaranteed memory allocation over many NUMA nodes?

# 28 - Migrate Replicated Control Plane To Use Cloud Controller Manager

**FEATURE STATE:** `Kubernetes v1.21 [alpha]`

The cloud-controller-manager is a Kubernetes control plane component that embeds cloud-specific control logic. The cloud controller manager lets you link your cluster into your cloud provider's API, and separates out the components that interact with that cloud platform from components that only interact with your cluster.

By decoupling the interoperability logic between Kubernetes and the underlying cloud infrastructure, the cloud-controller-manager component enables cloud providers to release features at a different pace compared to the main Kubernetes project.

## Background

As part of the cloud provider extraction effort, all cloud specific controllers must be moved out of the `kube-controller-manager`. All existing clusters that run cloud controllers in the `kube-controller-manager` must migrate to instead run the controllers in a cloud provider specific `cloud-controller-manager`.

Leader Migration provides a mechanism in which HA clusters can safely migrate "cloud specific" controllers between the `kube-controller-manager` and the `cloud-controller-manager` via a shared resource lock between the two components while upgrading the replicated control plane. For a single-node control plane, or if unavailability of controller managers can be tolerated during the upgrade, Leader Migration is not needed and this guide can be ignored.

Leader Migration is an alpha feature that is disabled by default and it requires `--enable-leader-migration` to be set on controller managers. It can be enabled by setting the feature gate `ControllerManagerLeaderMigration` plus `--enable-leader-migration` on `kube-controller-manager` or `cloud-controller-manager`. Leader Migration only applies during the upgrade and can be safely disabled or left enabled after the upgrade is complete.

This guide walks you through the manual process of upgrading the control plane from `kube-controller-manager` with built-in cloud provider to running both `kube-controller-manager` and `cloud-controller-manager`. If you use a tool to administrator the cluster, please refer to the documentation of the tool and the cloud provider for more details.

## Before you begin

It is assumed that the control plane is running Kubernetes version N and to be upgraded to version N + 1. Although it is possible to migrate within the same version, ideally the migration should be performed as part of a upgrade so that changes of configuration can be aligned to releases. The exact versions of N and N + 1 depend on each cloud provider. For example, if a cloud provider builds a `cloud-controller-manager` to work with Kubernetes 1.22, then N can be 1.21 and N + 1 can be 1.22.

The control plane nodes should run `kube-controller-manager` with Leader Election enabled through `--leader-elect=true`. As of version N, an in-tree cloud privider must be set with `--cloud-provider` flag and `cloud-controller-manager` should not yet be deployed.

The out-of-tree cloud provider must have built a `cloud-controller-manager` with Leader Migration implmentation. If the cloud provider imports `k8s.io/cloud-provider` and `k8s.io/controller-manager` of version v0.21.0 or later, Leader Migration will be avaliable.

This guide assumes that kubelet of each control plane node starts `kube-controller-manager` and `cloud-controller-manager` as static pods defined by their manifests. If the components run in a different setting, please adjust the steps accordingly.

For authorization, this guide assumes that the cluser uses RBAC. If another authorization mode grants permissions to `kube-controller-manager` and `cloud-controller-manager` components, please grant the needed access in a way that matches the mode.

### Grant access to Migration Lease

The default permissions of the controller manager allow only accesses to their main Lease. In order for the migration to work, accesses to another Lease are required.

You can grant `kube-controller-manager` full access to the leases API by modifying the `system::leader-locking-kube-controller-manager` role. This task guide assumes that the name of the migration lease is `cloud-provider-extraction-migration`.

```
kubectl patch -n kube-system role 'system::leader-locking-kube-controller-manager' -p '{"rules": [ {"apiGroups":[ "coordination.k8s.io"], "resources": ["leases"], "resourceNames": ["cloud-provider-extraction-migration"], "verbs": ["create", "list", "get", "update"] } ]}' --type=merge
```

Do the same to the `system::leader-locking-cloud-controller-manager` role.

```
kubectl patch -n kube-system role 'system::leader-locking-cloud-controller-manager' -p '{"rules": [ {"apiGroups":[ "coordination.k8s.io"], "resources": ["leases"], "resourceNames": ["cloud-provider-extraction-migration"], "verbs": ["create", "list", "get", "update"] } ]}' --type=merge
```

## Initial Leader Migration configuration

Leader Migration requires a configuration file representing the state of controller-to-manager assignment. At this moment, with in-tree cloud provider, `kube-controller-manager` runs `route`, `service`, and `cloud-node-lifecycle`. The following example configuration shows the assignment.

```
kind: LeaderMigrationConfiguration
apiVersion: controllermanager.config.k8s.io/v1alpha1
leaderName: cloud-provider-extraction-migration
resourceLock: leases
controllerLeaders:
  - name: route
    component: kube-controller-manager
  - name: service
    component: kube-controller-manager
  - name: cloud-node-lifecycle
    component: kube-controller-manager
```

On each control plane node, save the content to `/etc/leadermigration.conf`, and update the manifest of `kube-controller-manager` so that the file is mounted inside the container at the same location. Also, update the same manifest to add the following arguments:

- `--feature-gates=ControllerManagerLeaderMigration=true` to enable Leader Migration which is an alpha feature
- `--enable-leader-migration` to enable Leader Migration on the controller manager
- `--leader-migration-config=/etc/leadermigration.conf` to set configuration file

Restart `kube-controller-manager` on each node. At this moment, `kube-controller-manager` has leader migration enabled and is ready for the migration.

## Deploy Cloud Controller Manager

In version N + 1, the desired state of controller-to-manager assignment can be represented by a new configuration file, shown as follows. Please note `component` field of each `controllerLeaders` changing from `kube-controller-manager` to `cloud-controller-manager`.

```
kind: LeaderMigrationConfiguration
apiVersion: controllermanager.config.k8s.io/v1alpha1
leaderName: cloud-provider-extraction-migration
resourceLock: leases
controllerLeaders:
  - name: route
    component: cloud-controller-manager
  - name: service
    component: cloud-controller-manager
  - name: cloud-node-lifecycle
    component: cloud-controller-manager
```

When creating control plane nodes of version N + 1, the content should be deploy to `/etc/leadermigration.conf`. The manifest of `cloud-controller-manager` should be updated to mount the configuration file in the same manner as `kube-controller-manager` of version N. Similarly, add `--feature-gates=ControllerManagerLeaderMigration=true`, `--enable-leader-migration`, and `--leader-migration-config=/etc/leadermigration.conf` to the arguments of `cloud-controller-manager`.

Create a new control plane node of version N + 1 with the updated `cloud-controller-manager` manifest, and with the `--cloud-provider` flag unset for `kube-controller-manager`. `kube-controller-manager` of version N + 1 MUST NOT have Leader Migration enabled because, with an external cloud provider, it does not run the migrated controllers anymore and thus it is not involved in the migration.

Please refer to Cloud Controller Manager Administration for more detail on how to deploy `cloud-controller-manager`.

## Upgrade Control Plane

The control plane now contains nodes of both version N and N + 1. The nodes of version N run `kube-controller-manager` only, and these of version N + 1 run both `kube-controller-manager` and `cloud-controller-manager`. The migrated controllers, as specified in the configuration, are running under either `kube-controller-manager` of version N or `cloud-controller-manager` of version N + 1 depending on which controller manager holds the migration lease. No controller will ever be running under both controller managers at any time.

In a rolling manner, create a new control plane node of version N + 1 and bring down one of version N + 1 until the control plane contains only nodes of version N + 1. If a rollback from version N + 1 to N is required, add nodes of version N with Leader Migration enabled for `kube-controller-manager` back to the control plane, replacing one of version N + 1 each time until there are only nodes of version N.

### (Optional) Disable Leader Migration

Now that the control plane has been upgraded to run both `kube-controller-manager` and `cloud-controller-manager` of version N + 1, Leader Migration has finished its job and can be safely disabled to save one Lease resource. It is safe to re-enable Leader Migration for the rollback in the future.

In a rolling manager, update manifest of `cloud-controller-manager` to unset both `--enable-leader-migration` and `--leader-migration-config=` flag, also remove the mount of `/etc/leadermigration.conf`, and finally remove `/etc/leadermigration.conf`. To re-enable Leader Migration, recreate the configuration file and add its mount and the flags that enable Leader Migration back to `cloud-controller-manager`.

## What's next

- Read the Controller Manager Leader Migration enhancement proposal

# 29 - Namespaces Walkthrough

Kubernetes namespaces help different projects, teams, or customers to share a Kubernetes cluster.

It does this by providing the following:

1. A scope for Names.
2. A mechanism to attach authorization and policy to a subsection of the cluster.

Use of multiple namespaces is optional.

This example demonstrates how to use Kubernetes namespaces to subdivide your cluster.

## Before you begin

You need to have a Kubernetes cluster, and the kubectl command-line tool must be configured to communicate with your cluster. It is recommended to run this tutorial on a cluster with at least two nodes that are not acting as control plane hosts. If you do not already have a cluster, you can create one by using minikube or you can use one of these Kubernetes playgrounds:

- Katacoda
- Play with Kubernetes

To check the version, enter `kubectl version`.

## Prerequisites

This example assumes the following:

1. You have an existing Kubernetes cluster.
2. You have a basic understanding of Kubernetes Pods, Services, and Deployments.

## Understand the default namespace

By default, a Kubernetes cluster will instantiate a default namespace when provisioning the cluster to hold the default set of Pods, Services, and Deployments used by the cluster.

Assuming you have a fresh cluster, you can inspect the available namespaces by doing the following:

```
kubectl get namespaces
```

```
NAME      STATUS    AGE
default   Active    13m
```

## Create new namespaces

For this exercise, we will create two additional Kubernetes namespaces to hold our content.

Let's imagine a scenario where an organization is using a shared Kubernetes cluster for development and production use cases.

The development team would like to maintain a space in the cluster where they can get a view on the list of Pods, Services, and Deployments they use to build and run their application. In this space, Kubernetes resources come and go, and the restrictions on who can or cannot modify resources are relaxed to enable agile development.

The operations team would like to maintain a space in the cluster where they can enforce strict procedures on who can or cannot manipulate the set of Pods, Services, and Deployments that run the production site.

One pattern this organization could follow is to partition the Kubernetes cluster into two namespaces: `development` and `production`.

Let's create two new namespaces to hold our work.

Use the file `namespace-dev.json` which describes a `development` namespace:

admin/namespace-dev.json

```
{
```

```
  "apiVersion": "v1",
  "kind": "Namespace",
  "metadata": {
    "name": "development",
    "labels": {
      "name": "development"
    }
  }
}
```

Create the `development` namespace using kubectl.

```
kubectl create -f https://k8s.io/examples/admin/namespace-dev.json
```

Save the following contents into file `namespace-prod.json` which describes a `production` namespace:

admin/namespace-prod.json

```
{
  "apiVersion": "v1",
  "kind": "Namespace",
  "metadata": {
    "name": "production",
    "labels": {
      "name": "production"
    }
  }
}
```

And then let's create the `production` namespace using kubectl.

```
kubectl create -f https://k8s.io/examples/admin/namespace-prod.json
```

To be sure things are right, let's list all of the namespaces in our cluster.

```
kubectl get namespaces --show-labels
```

```
NAME          STATUS    AGE       LABELS
default       Active    32m       <none>
development   Active    29s       name=development
production    Active    23s       name=production
```

# Create pods in each namespace

A Kubernetes namespace provides the scope for Pods, Services, and Deployments in the cluster.

Users interacting with one namespace do not see the content in another namespace.

To demonstrate this, let's spin up a simple Deployment and Pods in the `development` namespace.

We first check what is the current context:

```
kubectl config view
```

```
apiVersion: v1
clusters:
- cluster:
    certificate-authority-data: REDACTED
    server: https://130.211.122.180
  name: lithe-cocoa-92103_kubernetes
```

```
contexts:
- context:
    cluster: lithe-cocoa-92103_kubernetes
    user: lithe-cocoa-92103_kubernetes
  name: lithe-cocoa-92103_kubernetes
current-context: lithe-cocoa-92103_kubernetes
kind: Config
preferences: {}
users:
- name: lithe-cocoa-92103_kubernetes
  user:
    client-certificate-data: REDACTED
    client-key-data: REDACTED
    token: 65rZW78y8HbwXXtSXuUw9DbP4FLjHi4b
- name: lithe-cocoa-92103_kubernetes-basic-auth
  user:
    password: h5M0FtUUIflBSdI7
    username: admin
```

```
kubectl config current-context
```

```
lithe-cocoa-92103_kubernetes
```

The next step is to define a context for the kubectl client to work in each namespace. The value of "cluster" and "user" fields are copied from the current context.

```
kubectl config set-context dev --namespace=development \
  --cluster=lithe-cocoa-92103_kubernetes \
  --user=lithe-cocoa-92103_kubernetes

kubectl config set-context prod --namespace=production \
  --cluster=lithe-cocoa-92103_kubernetes \
  --user=lithe-cocoa-92103_kubernetes
```

By default, the above commands adds two contexts that are saved into file `.kube/config`. You can now view the contexts and alternate against the two new request contexts depending on which namespace you wish to work against.

To view the new contexts:

```
kubectl config view
```

```
apiVersion: v1
clusters:
- cluster:
    certificate-authority-data: REDACTED
    server: https://130.211.122.180
  name: lithe-cocoa-92103_kubernetes
contexts:
- context:
    cluster: lithe-cocoa-92103_kubernetes
    user: lithe-cocoa-92103_kubernetes
  name: lithe-cocoa-92103_kubernetes
- context:
    cluster: lithe-cocoa-92103_kubernetes
    namespace: development
    user: lithe-cocoa-92103_kubernetes
  name: dev
- context:
    cluster: lithe-cocoa-92103_kubernetes
    namespace: production
    user: lithe-cocoa-92103_kubernetes
  name: prod
current-context: lithe-cocoa-92103_kubernetes
kind: Config
preferences: {}
users:
- name: lithe-cocoa-92103_kubernetes
  user:
    client-certificate-data: REDACTED
    client-key-data: REDACTED
    token: 65rZW78y8HbwXXtSXuUw9DbP4FLjHi4b
```

```
- name: lithe-cocoa-92103_kubernetes-basic-auth
  user:
    password: h5M0FtUUIflBSdI7
    username: admin
```

Let's switch to operate in the `development` namespace.

```
kubectl config use-context dev
```

You can verify your current context by doing the following:

```
kubectl config current-context
```

```
dev
```

At this point, all requests we make to the Kubernetes cluster from the command line are scoped to the `development` namespace.

Let's create some contents.

admin/snowflake-deployment.yaml

```
apiVersion: apps/v1
kind: Deployment
metadata:
  labels:
    app: snowflake
  name: snowflake
spec:
  replicas: 2
  selector:
    matchLabels:
      app: snowflake
  template:
    metadata:
      labels:
        app: snowflake
    spec:
      containers:
      - image: k8s.gcr.io/serve_hostname
        imagePullPolicy: Always
        name: snowflake
```

Apply the manifest to create a Deployment

```
kubectl apply -f https://k8s.io/examples/admin/snowflake-deployment.yaml
```

We have created a deployment whose replica size is 2 that is running the pod called `snowflake` with a basic container that serves the hostname.

```
kubectl get deployment
```

```
NAME         READY   UP-TO-DATE   AVAILABLE   AGE
snowflake    2/2     2            2           2m
```

```
kubectl get pods -l app=snowflake
```

```
NAME                          READY     STATUS    RESTARTS   AGE
snowflake-3968820950-9dgr8    1/1       Running   0          2m
snowflake-3968820950-vgc4n    1/1       Running   0          2m
```

And this is great, developers are able to do what they want, and they do not have to worry about affecting content in the `production` namespace.

Let's switch to the `production` namespace and show how resources in one namespace are hidden from the other.

```
kubectl config use-context prod
```

The `production` namespace should be empty, and the following commands should return nothing.

```
kubectl get deployment
kubectl get pods
```

Production likes to run cattle, so let's create some cattle pods.

```
kubectl create deployment cattle --image=k8s.gcr.io/serve_hostname --replicas=5

kubectl get deployment
```

```
NAME         READY   UP-TO-DATE   AVAILABLE   AGE
cattle       5/5     5            5           10s
```

```
kubectl get pods -l app=cattle
```

```
NAME                      READY     STATUS    RESTARTS   AGE
cattle-2263376956-41xy6   1/1       Running   0          34s
cattle-2263376956-kw466   1/1       Running   0          34s
cattle-2263376956-n4v97   1/1       Running   0          34s
cattle-2263376956-p5p3i   1/1       Running   0          34s
cattle-2263376956-sxpth   1/1       Running   0          34s
```

At this point, it should be clear that the resources users create in one namespace are hidden from the other namespace.

As the policy support in Kubernetes evolves, we will extend this scenario to show how you can provide different authorization rules for each namespace.

</>

</>

# 30 - Operating etcd clusters for Kubernetes

etcd is a consistent and highly-available key value store used as Kubernetes' backing store for all cluster data.

If your Kubernetes cluster uses etcd as its backing store, make sure you have a back up plan for those data.

You can find in-depth information about etcd in the official documentation.

## Before you begin

You need to have a Kubernetes cluster, and the kubectl command-line tool must be configured to communicate with your cluster. It is recommended to run this tutorial on a cluster with at least two nodes that are not acting as control plane hosts. If you do not already have a cluster, you can create one by using minikube or you can use one of these Kubernetes playgrounds:

- Katacoda
- Play with Kubernetes

To check the version, enter `kubectl version`.

## Prerequisites

- Run etcd as a cluster of odd members.
- etcd is a leader-based distributed system. Ensure that the leader periodically send heartbeats on time to all followers to keep the cluster stable.
- Ensure that no resource starvation occurs.

  Performance and stability of the cluster is sensitive to network and disk I/O. Any resource starvation can lead to heartbeat timeout, causing instability of the cluster. An unstable etcd indicates that no leader is elected. Under such circumstances, a cluster cannot make any changes to its current state, which implies no new pods can be scheduled.
- Keeping etcd clusters stable is critical to the stability of Kubernetes clusters. Therefore, run etcd clusters on dedicated machines or isolated environments for guaranteed resource requirements.
- The minimum recommended version of etcd to run in production is `3.2.10+`.

## Resource requirements

Operating etcd with limited resources is suitable only for testing purposes. For deploying in production, advanced hardware configuration is required. Before deploying etcd in production, see resource requirement reference.

## Starting etcd clusters

This section covers starting a single-node and multi-node etcd cluster.

### Single-node etcd cluster

Use a single-node etcd cluster only for testing purpose.

1. Run the following:

```
etcd --listen-client-urls=http://$PRIVATE_IP:2379 \
   --advertise-client-urls=http://$PRIVATE_IP:2379
```

2. Start the Kubernetes API server with the flag `--etcd-servers=$PRIVATE_IP:2379`.

   Make sure `PRIVATE_IP` is set to your etcd client IP.

### Multi-node etcd cluster

For durability and high availability, run etcd as a multi-node cluster in production and back it up periodically. A five-member cluster is recommended in production. For more information, see FAQ documentation.

Configure an etcd cluster either by static member information or by dynamic discovery. For more information on clustering, see etcd clustering documentation.

For an example, consider a five-member etcd cluster running with the following client URLs: `http://$IP1:2379`, `http://$IP2:2379`, `http://$IP3:2379`, `http://$IP4:2379`, and `http://$IP5:2379`. To start a Kubernetes API server:

1. Run the following:

```
etcd --listen-client-urls=http://$IP1:2379,http://$IP2:2379,http://$IP3:2379,http://$IP4:2379
```

2. Start the Kubernetes API servers with the flag `--etcd-servers=$IP1:2379,$IP2:2379,$IP3:2379,$IP4:2379,$IP5:2379`.

   Make sure the `IP<n>` variables are set to your client IP addresses.

## Multi-node etcd cluster with load balancer

To run a load balancing etcd cluster:

1. Set up an etcd cluster.
2. Configure a load balancer in front of the etcd cluster. For example, let the address of the load balancer be `$LB`.
3. Start Kubernetes API Servers with the flag `--etcd-servers=$LB:2379`.

# Securing etcd clusters

Access to etcd is equivalent to root permission in the cluster so ideally only the API server should have access to it. Considering the sensitivity of the data, it is recommended to grant permission to only those nodes that require access to etcd clusters.

To secure etcd, either set up firewall rules or use the security features provided by etcd. etcd security features depend on x509 Public Key Infrastructure (PKI). To begin, establish secure communication channels by generating a key and certificate pair. For example, use key pairs `peer.key` and `peer.cert` for securing communication between etcd members, and `client.key` and `client.cert` for securing communication between etcd and its clients. See the example scripts provided by the etcd project to generate key pairs and CA files for client authentication.

## Securing communication

To configure etcd with secure peer communication, specify flags `--peer-key-file=peer.key` and `--peer-cert-file=peer.cert`, and use HTTPS as the URL schema.

Similarly, to configure etcd with secure client communication, specify flags `--key-file=k8sclient.key` and `--cert-file=k8sclient.cert`, and use HTTPS as the URL schema. Here is an example on a client command that uses secure communication:

```
ETCDCTL_API=3 etcdctl --endpoints 10.2.0.9:2379 \
  --cert=/etc/kubernetes/pki/etcd/server.crt \
  --key=/etc/kubernetes/pki/etcd/server.key \
  --cacert=/etc/kubernetes/pki/etcd/ca.crt \
  member list
```

## Limiting access of etcd clusters

After configuring secure communication, restrict the access of etcd cluster to only the Kubernetes API servers. Use TLS authentication to do so.

For example, consider key pairs `k8sclient.key` and `k8sclient.cert` that are trusted by the CA `etcd.ca`. When etcd is configured with `--client-cert-auth` along with TLS, it verifies the certificates from clients by using system CAs or the CA passed in by `--trusted-ca-file` flag. Specifying flags `--client-cert-auth=true` and `--trusted-ca-file=etcd.ca` will restrict the access to clients with the certificate `k8sclient.cert`.

Once etcd is configured correctly, only clients with valid certificates can access it. To give Kubernetes API servers the access, configure them with the flags `--etcd-certfile=k8sclient.cert`, `--etcd-keyfile=k8sclient.key` and `--etcd-cafile=ca.cert`.

> **Note:** etcd authentication is not currently supported by Kubernetes. For more information, see the related issue Support Basic Auth for Etcd v2.

# Replacing a failed etcd member

etcd cluster achieves high availability by tolerating minor member failures. However, to improve the overall health of the cluster, replace failed members immediately. When multiple members fail, replace them one by one. Replacing a failed member involves two steps: removing the failed member and adding a new member.

Though etcd keeps unique member IDs internally, it is recommended to use a unique name for each member to avoid human errors. For example, consider a three-member etcd cluster. Let the URLs be, `member1=http://10.0.0.1`, `member2=http://10.0.0.2`, and `member3=http://10.0.0.3`. When `member1` fails, replace it with `member4=http://10.0.0.4`.

1. Get the member ID of the failed `member1`:

   ```
   etcdctl --endpoints=http://10.0.0.2,http://10.0.0.3 member list
   ```

   The following message is displayed:

   ```
   8211f1d0f64f3269, started, member1, http://10.0.0.1:2380, http://10.0.0.1:2379
   91bc3c398fb3c146, started, member2, http://10.0.0.2:2380, http://10.0.0.2:2379
   fd422379fda50e48, started, member3, http://10.0.0.3:2380, http://10.0.0.3:2379
   ```

2. Remove the failed member:

   ```
   etcdctl member remove 8211f1d0f64f3269
   ```

   The following message is displayed:

   ```
   Removed member 8211f1d0f64f3269 from cluster
   ```

3. Add the new member:

   ```
   etcdctl member add member4 --peer-urls=http://10.0.0.4:2380
   ```

   The following message is displayed:

   ```
   Member 2be1eb8f84b7f63e added to cluster ef37ad9dc622a7c4
   ```

4. Start the newly added member on a machine with the IP `10.0.0.4`:

   ```
   export ETCD_NAME="member4"
   export ETCD_INITIAL_CLUSTER="member2=http://10.0.0.2:2380,member3=http://10.0.0.3:2380,member
   export ETCD_INITIAL_CLUSTER_STATE=existing
   etcd [flags]
   ```

5. Do either of the following:

   1. Update the `--etcd-servers` flag for the Kubernetes API servers to make Kubernetes aware of the configuration changes, then restart the Kubernetes API servers.
   2. Update the load balancer configuration if a load balancer is used in the deployment.

For more information on cluster reconfiguration, see etcd reconfiguration documentation.

# Backing up an etcd cluster

All Kubernetes objects are stored on etcd. Periodically backing up the etcd cluster data is important to recover Kubernetes clusters under disaster scenarios, such as losing all control plane nodes. The snapshot file contains all the Kubernetes states and critical information. In order to keep the sensitive Kubernetes data safe, encrypt the snapshot files.

Backing up an etcd cluster can be accomplished in two ways: etcd built-in snapshot and volume snapshot.

## Built-in snapshot

etcd supports built-in snapshot. A snapshot may either be taken from a live member with the `etcdctl snapshot save` command or by copying the `member/snap/db` file from an etcd data directory that is not currently used by an etcd process. Taking the snapshot will not affect the performance of the member.

Below is an example for taking a snapshot of the keyspace served by `$ENDPOINT` to the file `snapshotdb`:

```
ETCDCTL_API=3 etcdctl --endpoints $ENDPOINT snapshot save snapshotdb
```

Verify the snapshot:

```
ETCDCTL_API=3 etcdctl --write-out=table snapshot status snapshotdb
```

```
+----------+----------+------------+------------+
|   HASH   | REVISION | TOTAL KEYS | TOTAL SIZE |
+----------+----------+------------+------------+
| fe01cf57 |       10 |          7 | 2.1 MB     |
+----------+----------+------------+------------+
```

</>

## Volume snapshot

If etcd is running on a storage volume that supports backup, such as Amazon Elastic Block Store, back up etcd data by taking a snapshot of the storage volume.

### Snapshot using etcdctl options

We can also take the snapshot using various options given by etcdctl. For example

```
ETCDCTL_API=3 etcdctl -h
```

will list various options available from etcdctl. For example, you can take a snapshot by specifying the endpoint, certificates etc as shown below:

```
ETCDCTL_API=3 etcdctl --endpoints=https://127.0.0.1:2379 \
  --cacert=<trusted-ca-file> --cert=<cert-file> --key=<key-file> \
  snapshot save <backup-file-location>
```

where `trusted-ca-file`, `cert-file` and `key-file` can be obtained from the description of the etcd Pod.

# Scaling up etcd clusters

Scaling up etcd clusters increases availability by trading off performance. Scaling does not increase cluster performance nor capability. A general rule is not to scale up or down etcd clusters. Do not configure any auto scaling groups for etcd clusters. It is highly recommended to always run a static five-member etcd cluster for production Kubernetes clusters at any officially supported scale.

A reasonable scaling is to upgrade a three-member cluster to a five-member one, when more reliability is desired. See etcd reconfiguration documentation for information on how to add members into an existing cluster.

# Restoring an etcd cluster

etcd supports restoring from snapshots that are taken from an etcd process of the major.minor version. Restoring a version from a different patch version of etcd also is supported. A restore operation is employed to recover the data of a failed cluster.

Before starting the restore operation, a snapshot file must be present. It can either be a snapshot file from a previous backup operation, or from a remaining data directory. Here is an example:

```
ETCDCTL_API=3 etcdctl --endpoints 10.2.0.9:2379 snapshot restore snapshotdb
```

For more information and examples on restoring a cluster from a snapshot file, see etcd disaster recovery documentation.

If the access URLs of the restored cluster is changed from the previous cluster, the Kubernetes API server must be reconfigured accordingly. In this case, restart Kubernetes API servers with the flag `--etcd-servers=$NEW_ETCD_CLUSTER` instead of the flag `--etcd-servers=$OLD_ETCD_CLUSTER`. Replace

`$NEW_ETCD_CLUSTER` and `$OLD_ETCD_CLUSTER` with the respective IP addresses. If a load balancer is used in front of an etcd cluster, you might need to update the load balancer instead.

If the majority of etcd members have permanently failed, the etcd cluster is considered failed. In this scenario, Kubernetes cannot make any changes to its current state. Although the scheduled pods might continue to run, no new pods can be scheduled. In such cases, recover the etcd cluster and potentially reconfigure Kubernetes API servers to fix the issue.

> **Note:**
> If any API servers are running in your cluster, you should not attempt to restore instances of etcd. Instead, follow these steps to restore etcd:
>
> - stop *all* API server instances
> - restore state in all etcd instances
> - restart all API server instances
>
> We also recommend restarting any components (e.g. `kube-scheduler`, `kube-controller-manager`, `kubelet`) to ensure that they don't rely on some stale data. Note that in practice, the restore takes a bit of time. During the restoration, critical components will lose leader lock and restart themselves.

# 31 - Reconfigure a Node's Kubelet in a Live Cluster

**FEATURE STATE:** `Kubernetes v1.11 [beta]`

Dynamic Kubelet Configuration allows you to change the configuration of each kubelet in a running Kubernetes cluster, by deploying a ConfigMap and configuring each Node to use it.

> **Warning:** All kubelet configuration parameters can be changed dynamically, but this is unsafe for some parameters. Before deciding to change a parameter dynamically, you need a strong understanding of how that change will affect your cluster's behavior. Always carefully test configuration changes on a small set of nodes before rolling them out cluster-wide. Advice on configuring specific fields is available in the inline `KubeletConfiguration`.

## Before you begin

You need to have a Kubernetes cluster. You also need kubectl v1.11 or higher, configured to communicate with your cluster. Your Kubernetes server must be at or later than version v1.11. To check the version, enter `kubectl version`. Your cluster API server version (eg v1.12) must be no more than one minor version away from the version of kubectl that you are using. For example, if your cluster is running v1.16 then you can use kubectl v1.15, v1.16 or v1.17; other combinations aren't supported.

Some of the examples use the command line tool jq. You do not need `jq` to complete the task, because there are manual alternatives.

For each node that you're reconfiguring, you must set the kubelet `--dynamic-config-dir` flag to a writable directory.

## Reconfiguring the kubelet on a running node in your cluster

### Basic workflow overview

The basic workflow for configuring a kubelet in a live cluster is as follows:

1. Write a YAML or JSON configuration file containing the kubelet's configuration.
2. Wrap this file in a ConfigMap and save it to the Kubernetes control plane.
3. Update the kubelet's corresponding Node object to use this ConfigMap.

Each kubelet watches a configuration reference on its respective Node object. When this reference changes, the kubelet downloads the new configuration, updates a local reference to refer to the file, and exits. For the feature to work correctly, you must be running an OS-level service manager (such as systemd), which will restart the kubelet if it exits. When the kubelet is restarted, it will begin using the new configuration.

The new configuration completely overrides configuration provided by `--config`, and is overridden by command-line flags. Unspecified values in the new configuration will receive default values appropriate to the configuration version (e.g. `kubelet.config.k8s.io/v1beta1`), unless overridden by flags.

The status of the Node's kubelet configuration is reported via `Node.Status.Config`. Once you have updated a Node to use the new ConfigMap, you can observe this status to confirm that the Node is using the intended configuration.

This document describes editing Nodes using `kubectl edit`. There are other ways to modify a Node's spec, including `kubectl patch`, for example, which facilitate scripted workflows.

This document only describes a single Node consuming each ConfigMap. Keep in mind that it is also valid for multiple Nodes to consume the same ConfigMap.

> **Warning:** While it is *possible* to change the configuration by updating the ConfigMap in-place, this causes all kubelets configured with that ConfigMap to update simultaneously. It is much safer to treat ConfigMaps as immutable by convention, aided by `kubectl`'s `--append-hash` option, and incrementally roll out updates to `Node.Spec.ConfigSource`.

### Automatic RBAC rules for Node Authorizer

Previously, you were required to manually create RBAC rules to allow Nodes to access their assigned ConfigMaps. The Node Authorizer now automatically configures these rules.

### Generating a file that contains the current configuration

The Dynamic Kubelet Configuration feature allows you to provide an override for the entire configuration object, rather than a per-field overlay. This is a simpler model that makes it easier to trace the source of configuration values and debug issues. The compromise, however, is that you must start with knowledge of the existing configuration to ensure that you only change the fields you intend to change.

The kubelet loads settings from its configuration file, but you can set command line flags to override the configuration in the file. This means that if you only know the contents of the configuration file, and you don't know the command line overrides, then you do not know the running configuration either.

Because you need to know the running configuration in order to override it, you can fetch the running configuration from the kubelet. You can generate a config file containing a Node's current configuration by accessing the kubelet's `configz` endpoint, through `kubectl proxy`. The next section explains how to do this.

> **Caution:** The kubelet's `configz` endpoint is there to help with debugging, and is not a stable part of kubelet behavior. Do not rely on the behavior of this endpoint for production scenarios or for use with automated tools.

For more information on configuring the kubelet via a configuration file, see Set kubelet parameters via a config file).

## Generate the configuration file

> **Note:** The steps below use the `jq` command to streamline working with JSON. To follow the tasks as written, you need to have `jq` installed. You can adapt the steps if you prefer to extract the `kubeletconfig` subobject manually.

1. Choose a Node to reconfigure. In this example, the name of this Node is referred to as `NODE_NAME`.
2. Start the kubectl proxy in the background using the following command:

```
kubectl proxy --port=8001 &
```

3. Run the following command to download and unpack the configuration from the `configz` endpoint. The command is long, so be careful when copying and pasting. **If you use zsh**, note that common zsh configurations add backslashes to escape the opening and closing curly braces around the variable name in the URL. For example: `${NODE_NAME}` will be rewritten as `$\{NODE_NAME\}` during the paste. You must remove the backslashes before running the command, or the command will fail.

```
NODE_NAME="the-name-of-the-node-you-are-reconfiguring"; curl -sSL "http://localhost:8001/api/
```

> **Note:** You need to manually add the `kind` and `apiVersion` to the downloaded object, because those fields are not reported by the `configz` endpoint.

## Edit the configuration file

Using a text editor, change one of the parameters in the file generated by the previous procedure. For example, you might edit the parameter `eventRecordQPS`, that controls rate limiting for event recording.

## Push the configuration file to the control plane

Push the edited configuration file to the control plane with the following command:

```
kubectl -n kube-system create configmap my-node-config --from-file=kubelet=kubelet_configz_${NODE
```

This is an example of a valid response:

```
apiVersion: v1
kind: ConfigMap
metadata:
  creationTimestamp: 2017-09-14T20:23:33Z
  name: my-node-config-gkt4c2m4b2
  namespace: kube-system
  resourceVersion: "119980"
  uid: 946d785e-998a-11e7-a8dd-42010a800006
data:
  kubelet: |
    {...}
```

You created that ConfigMap inside the `kube-system` namespace because the kubelet is a Kubernetes system component.

The `--append-hash` option appends a short checksum of the ConfigMap contents to the name. This is convenient for an edit-then-push workflow, because it automatically, yet deterministically, generates new names for new resources. The name that includes this generated hash is referred to as `CONFIG_MAP_NAME` in the following examples.

## Set the Node to use the new configuration

Edit the Node's reference to point to the new ConfigMap with the following command:

```
kubectl edit node ${NODE_NAME}
```

In your text editor, add the following YAML under `spec`:

```
configSource:
    configMap:
        name: CONFIG_MAP_NAME # replace CONFIG_MAP_NAME with the name of the ConfigMap
        namespace: kube-system
        kubeletConfigKey: kubelet
```

You must specify all three of `name`, `namespace`, and `kubeletConfigKey`. The `kubeletConfigKey` parameter shows the kubelet which key of the ConfigMap contains its config.

## Observe that the Node begins using the new configuration

Retrieve the Node using the `kubectl get node ${NODE_NAME} -o yaml` command and inspect `Node.Status.Config`. The config sources corresponding to the `active`, `assigned`, and `lastKnownGood` configurations are reported in the status.

- The `active` configuration is the version the kubelet is currently running with.
- The `assigned` configuration is the latest version the kubelet has resolved based on `Node.Spec.ConfigSource`.
- The `lastKnownGood` configuration is the version the kubelet will fall back to if an invalid config is assigned in `Node.Spec.ConfigSource`.

The `lastKnownGood` configuration might not be present if it is set to its default value, the local config deployed with the node. The status will update `lastKnownGood` to match a valid `assigned` config after the kubelet becomes comfortable with the config. The details of how the kubelet determines a config should become the `lastKnownGood` are not guaranteed by the API, but is currently implemented as a 10-minute grace period.

You can use the following command (using `jq`) to filter down to the config status:

```
kubectl get no ${NODE_NAME} -o json | jq '.status.config'
```

The following is an example response:

```
{
  "active": {
    "configMap": {
      "kubeletConfigKey": "kubelet",
      "name": "my-node-config-9mbkccg2cc",
      "namespace": "kube-system",
      "resourceVersion": "1326",
      "uid": "705ab4f5-6393-11e8-b7cc-42010a800002"
    }
  },
  "assigned": {
    "configMap": {
      "kubeletConfigKey": "kubelet",
      "name": "my-node-config-9mbkccg2cc",
      "namespace": "kube-system",
      "resourceVersion": "1326",
      "uid": "705ab4f5-6393-11e8-b7cc-42010a800002"
    }
  },
  "lastKnownGood": {
    "configMap": {
      "kubeletConfigKey": "kubelet",
      "name": "my-node-config-9mbkccg2cc",
```

```
      "namespace": "kube-system",
      "resourceVersion": "1326",
      "uid": "705ab4f5-6393-11e8-b7cc-42010a800002"
    }
  }
}
```

(if you do not have `jq`, you can look at the whole response and find `Node.Status.Config` by eye).

If an error occurs, the kubelet reports it in the `Node.Status.Config.Error` structure. Possible errors are listed in Understanding Node.Status.Config.Error messages. You can search for the identical text in the kubelet log for additional details and context about the error.

### Make more changes

Follow the workflow above to make more changes and push them again. Each time you push a ConfigMap with new contents, the `--append-hash` kubectl option creates the ConfigMap with a new name. The safest rollout strategy is to first create a new ConfigMap, and then update the Node to use the new ConfigMap.

### Reset the Node to use its local default configuration

To reset the Node to use the configuration it was provisioned with, edit the Node using `kubectl edit node ${NODE_NAME}` and remove the `Node.Spec.ConfigSource` field.

#### Observe that the Node is using its local default configuration

After removing this subfield, `Node.Status.Config` eventually becomes empty, since all config sources have been reset to `nil`, which indicates that the local default config is `assigned`, `active`, and `lastKnownGood`, and no error is reported.

## `kubectl patch` example

You can change a Node's configSource using several different mechanisms. This example uses `kubectl patch`:

```
kubectl patch node ${NODE_NAME} -p "{\"spec\":{\"configSource\":{\"configMap\":{\"name\":\"${CONF
```

# Understanding how the kubelet checkpoints config

When a new config is assigned to the Node, the kubelet downloads and unpacks the config payload as a set of files on the local disk. The kubelet also records metadata that locally tracks the assigned and last-known-good config sources, so that the kubelet knows which config to use across restarts, even if the API server becomes unavailable. After checkpointing a config and the relevant metadata, the kubelet exits if it detects that the assigned config has changed. When the kubelet is restarted by the OS-level service manager (such as `systemd`), it reads the new metadata and uses the new config.

The recorded metadata is fully resolved, meaning that it contains all necessary information to choose a specific config version - typically a `UID` and `ResourceVersion`. This is in contrast to `Node.Spec.ConfigSource`, where the intended config is declared via the idempotent `namespace/name` that identifies the target ConfigMap; the kubelet tries to use the latest version of this ConfigMap.

When you are debugging problems on a node, you can inspect the kubelet's config metadata and checkpoints. The structure of the kubelet's checkpointing directory is:

```
- --dynamic-config-dir (root for managing dynamic config)
| - meta
  | - assigned (encoded kubeletconfig/v1beta1.SerializedNodeConfigSource object, indicating the as
  | - last-known-good (encoded kubeletconfig/v1beta1.SerializedNodeConfigSource object, indicating
| - checkpoints
  | - uid1 (dir for versions of object identified by uid1)
    | - resourceVersion1 (dir for unpacked files from resourceVersion1 of object with uid1)
    | - ...
  | - ...
```

# Understanding `Node.Status.Config.Error` messages

The following table describes error messages that can occur when using Dynamic Kubelet Config. You can search for the identical text in the Kubelet log for additional details and context about the error.

| Error Message | Possible Causes |
| --- | --- |
| failed to load config, see Kubelet log for details | The kubelet likely could not parse the downloaded config payload, or encountered a filesystem error attempting to load the payload from disk. |
| failed to validate config, see Kubelet log for details | The configuration in the payload, combined with any command-line flag overrides, and the sum of feature gates from flags, the config file, and the remote payload, was determined to be invalid by the kubelet. |
| invalid NodeConfigSource, exactly one subfield must be non-nil, but all were nil | Since Node.Spec.ConfigSource is validated by the API server to contain at least one non-nil subfield, this likely means that the kubelet is older than the API server and does not recognize a newer source type. |
| failed to sync: failed to download config, see Kubelet log for details | The kubelet could not download the config. It is possible that Node.Spec.ConfigSource could not be resolved to a concrete API object, or that network errors disrupted the download attempt. The kubelet will retry the download when in this error state. |
| failed to sync: internal failure, see Kubelet log for details | The kubelet encountered some internal problem and failed to update its config as a result. Examples include filesystem errors and reading objects from the internal informer cache. |
| internal failure, see Kubelet log for details | The kubelet encountered some internal problem while manipulating config, outside of the configuration sync loop. |

# What's next

- For more information on configuring the kubelet via a configuration file, see Set kubelet parameters via a config file.
- See the reference documentation for `NodeConfigSource`
- Learn more about kubelet configuration by checking the `KubeletConfiguration` reference.

# 32 - Reserve Compute Resources for System Daemons

Kubernetes nodes can be scheduled to `Capacity`. Pods can consume all the available capacity on a node by default. This is an issue because nodes typically run quite a few system daemons that power the OS and Kubernetes itself. Unless resources are set aside for these system daemons, pods and system daemons compete for resources and lead to resource starvation issues on the node.

The `kubelet` exposes a feature named 'Node Allocatable' that helps to reserve compute resources for system daemons. Kubernetes recommends cluster administrators to configure 'Node Allocatable' based on their workload density on each node.

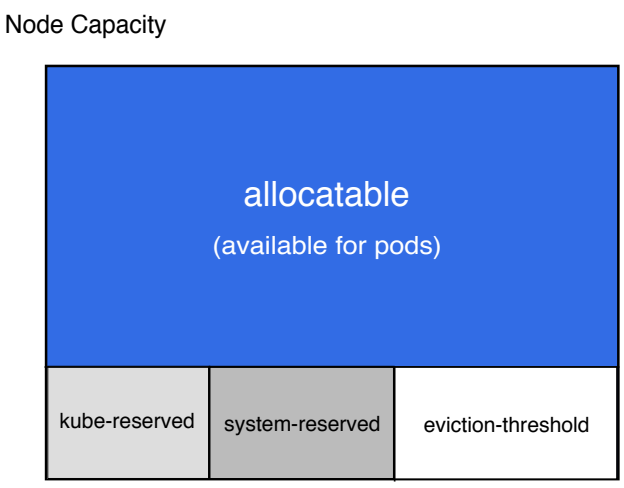
## Before you begin

You need to have a Kubernetes cluster, and the kubectl command-line tool must be configured to communicate with your cluster. It is recommended to run this tutorial on a cluster with at least two nodes that are not acting as control plane hosts. If you do not already have a cluster, you can create one by using minikube or you can use one of these Kubernetes playgrounds:

- Katacoda
- Play with Kubernetes

Your Kubernetes server must be at or later than version 1.8. To check the version, enter `kubectl version`. Your Kubernetes server must be at or later than version 1.17 to use the kubelet command line option `--reserved-cpus` to set an explicitly reserved CPU list.

## Node Allocatable




'Allocatable' on a Kubernetes node is defined as the amount of compute resources that are available for pods. The scheduler does not over-subscribe 'Allocatable'. 'CPU', 'memory' and 'ephemeral-storage' are supported as of now.

Node Allocatable is exposed as part of `v1.Node` object in the API and as part of `kubectl describe node` in the CLI.

Resources can be reserved for two categories of system daemons in the `kubelet`.

### Enabling QoS and Pod level cgroups

To properly enforce node allocatable constraints on the node, you must enable the new cgroup hierarchy via the `--cgroups-per-qos` flag. This flag is enabled by default. When enabled, the `kubelet` will parent all end-user pods under a cgroup hierarchy managed by the `kubelet`.

### Configuring a cgroup driver

The `kubelet` supports manipulation of the cgroup hierarchy on the host using a cgroup driver. The driver is configured via the `--cgroup-driver` flag.

The supported values are the following:

- `cgroupfs` is the default driver that performs direct manipulation of the cgroup filesystem on the host in order to manage cgroup sandboxes.
- `systemd` is an alternative driver that manages cgroup sandboxes using transient slices for resources that are supported by that init system.

Depending on the configuration of the associated container runtime, operators may have to choose a particular cgroup driver to ensure proper system behavior. For example, if operators use the `systemd` cgroup driver provided by the `docker` runtime, the `kubelet` must be configured to use the `systemd` cgroup driver.

## Kube Reserved

- **Kubelet Flag**: `--kube-reserved=[cpu=100m][,][memory=100Mi][,][ephemeral-storage=1Gi][,][pid=1000]`
- **Kubelet Flag**: `--kube-reserved-cgroup=`

`kube-reserved` is meant to capture resource reservation for kubernetes system daemons like the `kubelet`, `container runtime`, `node problem detector`, etc. It is not meant to reserve resources for system daemons that are run as pods. `kube-reserved` is typically a function of `pod density` on the nodes.

In addition to `cpu`, `memory`, and `ephemeral-storage`, `pid` may be specified to reserve the specified number of process IDs for kubernetes system daemons.

To optionally enforce `kube-reserved` on kubernetes system daemons, specify the parent control group for kube daemons as the value for `--kube-reserved-cgroup` kubelet flag.

It is recommended that the kubernetes system daemons are placed under a top level control group (`runtime.slice` on systemd machines for example). Each system daemon should ideally run within its own child control group. Refer to the design proposal for more details on recommended control group hierarchy.

Note that Kubelet **does not** create `--kube-reserved-cgroup` if it doesn't exist. Kubelet will fail if an invalid cgroup is specified.

## System Reserved

- **Kubelet Flag**: `--system-reserved=[cpu=100m][,][memory=100Mi][,][ephemeral-storage=1Gi][,][pid=1000]`
- **Kubelet Flag**: `--system-reserved-cgroup=`

`system-reserved` is meant to capture resource reservation for OS system daemons like `sshd`, `udev`, etc. `system-reserved` should reserve `memory` for the `kernel` too since `kernel` memory is not accounted to pods in Kubernetes at this time. Reserving resources for user login sessions is also recommended (`user.slice` in systemd world).

In addition to `cpu`, `memory`, and `ephemeral-storage`, `pid` may be specified to reserve the specified number of process IDs for OS system daemons.

To optionally enforce `system-reserved` on system daemons, specify the parent control group for OS system daemons as the value for `--system-reserved-cgroup` kubelet flag.

It is recommended that the OS system daemons are placed under a top level control group (`system.slice` on systemd machines for example).

Note that `kubelet` **does not** create `--system-reserved-cgroup` if it doesn't exist. `kubelet` will fail if an invalid cgroup is specified.

## Explicitly Reserved CPU List

**FEATURE STATE:** `Kubernetes v1.17 [stable]`

**Kubelet Flag**: `--reserved-cpus=0-3`

`reserved-cpus` is meant to define an explicit CPU set for OS system daemons and kubernetes system daemons. `reserved-cpus` is for systems that do not intend to define separate top level cgroups for OS system daemons and kubernetes system daemons with regard to cpuset resource. If the Kubelet **does not** have `--system-reserved-cgroup` and `--kube-reserved-cgroup`, the explicit cpuset provided by `reserved-cpus` will take precedence over the CPUs defined by `--kube-reserved` and `--system-reserved` options.

This option is specifically designed for Telco/NFV use cases where uncontrolled interrupts/timers may impact the workload performance. you can use this option to define the explicit cpuset for the system/kubernetes daemons as well as the interrupts/timers, so the rest CPUs on the system can be used exclusively for workloads, with less impact from uncontrolled interrupts/timers. To move the system daemon, kubernetes daemons and interrupts/timers to the explicit cpuset defined by this option, other mechanism outside Kubernetes should be used. For example: in Centos, you can do this using the tuned toolset.

## Eviction Thresholds

**Kubelet Flag**: `--eviction-hard=[memory.available<500Mi]`

Memory pressure at the node level leads to System OOMs which affects the entire node and all pods running on it. Nodes can go offline temporarily until memory has been reclaimed. To avoid (or reduce the probability of) system OOMs kubelet provides out of resource management. Evictions are supported for `memory` and `ephemeral-storage` only. By reserving some memory via `--eviction-hard` flag, the `kubelet` attempts to evict pods

whenever memory availability on the node drops below the reserved value. Hypothetically, if system daemons did not exist on a node, pods cannot use more than `capacity - eviction-hard`. For this reason, resources reserved for evictions are not available for pods.

### Enforcing Node Allocatable

**Kubelet Flag**: `--enforce-node-allocatable=pods[,][system-reserved][,][kube-reserved]`

The scheduler treats 'Allocatable' as the available `capacity` for pods.

`kubelet` enforce 'Allocatable' across pods by default. Enforcement is performed by evicting pods whenever the overall usage across all pods exceeds 'Allocatable'. More details on eviction policy can be found on the node pressure eviction page. This enforcement is controlled by specifying `pods` value to the kubelet flag `--enforce-node-allocatable`.

Optionally, `kubelet` can be made to enforce `kube-reserved` and `system-reserved` by specifying `kube-reserved` & `system-reserved` values in the same flag. Note that to enforce `kube-reserved` or `system-reserved`, `--kube-reserved-cgroup` or `--system-reserved-cgroup` needs to be specified respectively.

## General Guidelines

System daemons are expected to be treated similar to 'Guaranteed' pods. System daemons can burst within their bounding control groups and this behavior needs to be managed as part of kubernetes deployments. For example, `kubelet` should have its own control group and share `kube-reserved` resources with the container runtime. However, Kubelet cannot burst and use up all available Node resources if `kube-reserved` is enforced.

Be extra careful while enforcing `system-reserved` reservation since it can lead to critical system services being CPU starved, OOM killed, or unable to fork on the node. The recommendation is to enforce `system-reserved` only if a user has profiled their nodes exhaustively to come up with precise estimates and is confident in their ability to recover if any process in that group is oom-killed.

- To begin with enforce 'Allocatable' on `pods`.
- Once adequate monitoring and alerting is in place to track kube system daemons, attempt to enforce `kube-reserved` based on usage heuristics.
- If absolutely necessary, enforce `system-reserved` over time.

The resource requirements of kube system daemons may grow over time as more and more features are added. Over time, kubernetes project will attempt to bring down utilization of node system daemons, but that is not a priority as of now. So expect a drop in `Allocatable` capacity in future releases.

## Example Scenario

Here is an example to illustrate Node Allocatable computation:

- Node has `32Gi` of `memory`, `16 CPUs` and `100Gi` of `Storage`
- `--kube-reserved` is set to `cpu=1,memory=2Gi,ephemeral-storage=1Gi`
- `--system-reserved` is set to `cpu=500m,memory=1Gi,ephemeral-storage=1Gi`
- `--eviction-hard` is set to `memory.available<500Mi,nodefs.available<10%`

Under this scenario, 'Allocatable' will be 14.5 CPUs, 28.5Gi of memory and `88Gi` of local storage. Scheduler ensures that the total memory `requests` across all pods on this node does not exceed 28.5Gi and storage doesn't exceed 88Gi. Kubelet evicts pods whenever the overall memory usage across pods exceeds 28.5Gi, or if overall disk usage exceeds 88Gi If all processes on the node consume as much CPU as they can, pods together cannot consume more than 14.5 CPUs.

If `kube-reserved` and/or `system-reserved` is not enforced and system daemons exceed their reservation, `kubelet` evicts pods whenever the overall node memory usage is higher than 31.5Gi or `storage` is greater than 90Gi.

# 33 - Safely Drain a Node

This page shows how to safely drain a node, optionally respecting the PodDisruptionBudget you have defined.

## Before you begin

Your Kubernetes server must be at or later than version 1.5. To check the version, enter `kubectl version`.

This task also assumes that you have met the following prerequisites:

1. You do not require your applications to be highly available during the node drain, or
2. You have read about the PodDisruptionBudget concept, and have configured PodDisruptionBudgets for applications that need them.

## (Optional) Configure a disruption budget

To endure that your workloads remain available during maintenance, you can configure a PodDisruptionBudget.

If availability is important for any applications that run or could run on the node(s) that you are draining, configure a PodDisruptionBudgets first and the continue following this guide.

## Use `kubectl drain` to remove a node from service

You can use `kubectl drain` to safely evict all of your pods from a node before you perform maintenance on the node (e.g. kernel upgrade, hardware maintenance, etc.). Safe evictions allow the pod's containers to gracefully terminate and will respect the PodDisruptionBudgets you have specified.

> **Note:** By default `kubectl drain` ignores certain system pods on the node that cannot be killed; see the kubectl drain documentation for more details.

When `kubectl drain` returns successfully, that indicates that all of the pods (except the ones excluded as described in the previous paragraph) have been safely evicted (respecting the desired graceful termination period, and respecting the PodDisruptionBudget you have defined). It is then safe to bring down the node by powering down its physical machine or, if running on a cloud platform, deleting its virtual machine.

First, identify the name of the node you wish to drain. You can list all of the nodes in your cluster with

```
kubectl get nodes
```

Next, tell Kubernetes to drain the node:

```
kubectl drain <node name>
```

Once it returns (without giving an error), you can power down the node (or equivalently, if on a cloud platform, delete the virtual machine backing the node). If you leave the node in the cluster during the maintenance operation, you need to run

```
kubectl uncordon <node name>
```

afterwards to tell Kubernetes that it can resume scheduling new pods onto the node.

## Draining multiple nodes in parallel

The `kubectl drain` command should only be issued to a single node at a time. However, you can run multiple `kubectl drain` commands for different nodes in parallel, in different terminals or in the background. Multiple drain commands running concurrently will still respect the PodDisruptionBudget you specify.

For example, if you have a StatefulSet with three replicas and have set a PodDisruptionBudget for that set specifying `minAvailable: 2`, `kubectl drain` only evicts a pod from the StatefulSet if all three replicas pods are ready; if then you issue multiple drain commands in parallel, Kubernetes respects the PodDisruptionBudget and ensure that only 1 (calculated as `replicas - minAvailable`) Pod is unavailable at any given time. Any drains that would cause the number of ready replicas to fall below the specified budget are blocked.

# The Eviction API

If you prefer not to use kubectl drain (such as to avoid calling to an external command, or to get finer control over the pod eviction process), you can also programmatically cause evictions using the eviction API.

You should first be familiar with using Kubernetes language clients to access the API.

The eviction subresource of a Pod can be thought of as a kind of policy-controlled DELETE operation on the Pod itself. To attempt an eviction (more precisely: to attempt to *create* an Eviction), you POST an attempted operation. Here's an example:

```
{
  "apiVersion": "policy/v1beta1",
  "kind": "Eviction",
  "metadata": {
    "name": "quux",
    "namespace": "default"
  }
}
```

You can attempt an eviction using `curl`:

```
curl -v -H 'Content-type: application/json' https://your-cluster-api-endpoint.example/api/v1/name
```

The API can respond in one of three ways:

- If the eviction is granted, then the Pod is deleted as if you sent a `DELETE` request to the Pod's URL and received back `200 OK`.
- If the current state of affairs wouldn't allow an eviction by the rules set forth in the budget, you get back `429 Too Many Requests`. This is typically used for generic rate limiting of *any* requests, but here we mean that this request isn't allowed *right now* but it may be allowed later.
- If there is some kind of misconfiguration; for example multiple PodDisruptionBudgets that refer the same Pod, you get a `500 Internal Server Error` response.

For a given eviction request, there are two cases:

- There is no budget that matches this pod. In this case, the server always returns `200 OK`.
- There is at least one budget. In this case, any of the three above responses may apply.

## Stuck evictions

In some cases, an application may reach a broken state, one where unless you intervene the eviction API will never return anything other than 429 or 500.

For example: this can happen if ReplicaSet is creating Pods for your application but the replacement Pods do not become `Ready`. You can also see similar symptoms if the last Pod evicted has a very long termination grace period.

In this case, there are two potential solutions:

- Abort or pause the automated operation. Investigate the reason for the stuck application, and restart the automation.
- After a suitably long wait, `DELETE` the Pod from your cluster's control plane, instead of using the eviction API.

Kubernetes does not specify what the behavior should be in this case; it is up to the application owners and cluster owners to establish an agreement on behavior in these cases.

## What's next

- Follow steps to protect your application by configuring a Pod Disruption Budget.

</>

# 34 - Securing a Cluster

This document covers topics related to protecting a cluster from accidental or malicious access and provides recommendations on overall security.

## Before you begin

- You need to have a Kubernetes cluster, and the kubectl command-line tool must be configured to communicate with your cluster. It is recommended to run this tutorial on a cluster with at least two nodes that are not acting as control plane hosts. If you do not already have a cluster, you can create one by using minikube or you can use one of these Kubernetes playgrounds:
  - Katacoda
  - Play with Kubernetes

  To check the version, enter `kubectl version`.

## Controlling access to the Kubernetes API

As Kubernetes is entirely API driven, controlling and limiting who can access the cluster and what actions they are allowed to perform is the first line of defense.

### Use Transport Layer Security (TLS) for all API traffic

Kubernetes expects that all API communication in the cluster is encrypted by default with TLS, and the majority of installation methods will allow the necessary certificates to be created and distributed to the cluster components. Note that some components and installation methods may enable local ports over HTTP and administrators should familiarize themselves with the settings of each component to identify potentially unsecured traffic.

### API Authentication

Choose an authentication mechanism for the API servers to use that matches the common access patterns when you install a cluster. For instance, small single user clusters may wish to use a simple certificate or static Bearer token approach. Larger clusters may wish to integrate an existing OIDC or LDAP server that allow users to be subdivided into groups.

All API clients must be authenticated, even those that are part of the infrastructure like nodes, proxies, the scheduler, and volume plugins. These clients are typically service accounts or use x509 client certificates, and they are created automatically at cluster startup or are setup as part of the cluster installation.

Consult the authentication reference document for more information.

### API Authorization

Once authenticated, every API call is also expected to pass an authorization check. Kubernetes ships an integrated Role-Based Access Control (RBAC) component that matches an incoming user or group to a set of permissions bundled into roles. These permissions combine verbs (get, create, delete) with resources (pods, services, nodes) and can be namespace or cluster scoped. A set of out of the box roles are provided that offer reasonable default separation of responsibility depending on what actions a client might want to perform. It is recommended that you use the Node and RBAC authorizers together, in combination with the NodeRestriction admission plugin.

As with authentication, simple and broad roles may be appropriate for smaller clusters, but as more users interact with the cluster, it may become necessary to separate teams into separate namespaces with more limited roles.

With authorization, it is important to understand how updates on one object may cause actions in other places. For instance, a user may not be able to create pods directly, but allowing them to create a deployment, which creates pods on their behalf, will let them create those pods indirectly. Likewise, deleting a node from the API will result in the pods scheduled to that node being terminated and recreated on other nodes. The out of the box roles represent a balance between flexibility and the common use cases, but more limited roles should be carefully reviewed to prevent accidental escalation. You can make roles specific to your use case if the out-of-box ones don't meet your needs.

Consult the authorization reference section for more information.

## Controlling access to the Kubelet

Kubelets expose HTTPS endpoints which grant powerful control over the node and containers. By default Kubelets allow unauthenticated access to this API.

Production clusters should enable Kubelet authentication and authorization.

Consult the Kubelet authentication/authorization reference for more information.

# Controlling the capabilities of a workload or user at runtime

Authorization in Kubernetes is intentionally high level, focused on coarse actions on resources. More powerful controls exist as **policies** to limit by use case how those objects act on the cluster, themselves, and other resources.

## Limiting resource usage on a cluster

Resource quota limits the number or capacity of resources granted to a namespace. This is most often used to limit the amount of CPU, memory, or persistent disk a namespace can allocate, but can also control how many pods, services, or volumes exist in each namespace.

Limit ranges restrict the maximum or minimum size of some of the resources above, to prevent users from requesting unreasonably high or low values for commonly reserved resources like memory, or to provide default limits when none are specified.

## Controlling what privileges containers run with

A pod definition contains a security context that allows it to request access to running as a specific Linux user on a node (like root), access to run privileged or access the host network, and other controls that would otherwise allow it to run unfettered on a hosting node. Pod security policies can limit which users or service accounts can provide dangerous security context settings. For example, pod security policies can limit volume mounts, especially `hostPath`, which are aspects of a pod that should be controlled.

Generally, most application workloads need limited access to host resources so they can successfully run as a root process (uid 0) without access to host information. However, considering the privileges associated with the root user, you should write application containers to run as a non-root user. Similarly, administrators who wish to prevent client applications from escaping their containers should use a restrictive pod security policy.

## Preventing containers from loading unwanted kernel modules

The Linux kernel automatically loads kernel modules from disk if needed in certain circumstances, such as when a piece of hardware is attached or a filesystem is mounted. Of particular relevance to Kubernetes, even unprivileged processes can cause certain network-protocol-related kernel modules to be loaded, just by creating a socket of the appropriate type. This may allow an attacker to exploit a security hole in a kernel module that the administrator assumed was not in use.

To prevent specific modules from being automatically loaded, you can uninstall them from the node, or add rules to block them. On most Linux distributions, you can do that by creating a file such as `/etc/modprobe.d/kubernetes-blacklist.conf` with contents like:

```
# DCCP is unlikely to be needed, has had multiple serious
# vulnerabilities, and is not well-maintained.
blacklist dccp

# SCTP is not used in most Kubernetes clusters, and has also had
# vulnerabilities in the past.
blacklist sctp
```

To block module loading more generically, you can use a Linux Security Module (such as SELinux) to completely deny the `module_request` permission to containers, preventing the kernel from loading modules for containers under any circumstances. (Pods would still be able to use modules that had been loaded manually, or modules that were loaded by the kernel on behalf of some more-privileged process.)

## Restricting network access

The network policies for a namespace allows application authors to restrict which pods in other namespaces may access pods and ports within their namespaces. Many of the supported Kubernetes networking providers now respect network policy.

Quota and limit ranges can also be used to control whether users may request node ports or load balanced services, which on many clusters can control whether those users applications are visible outside of the cluster.

Additional protections may be available that control network rules on a per plugin or per environment basis, such as per-node firewalls, physically separating cluster nodes to prevent cross talk, or advanced networking policy.

## Restricting cloud metadata API access

Cloud platforms (AWS, Azure, GCE, etc.) often expose metadata services locally to instances. By default these APIs are accessible by pods running on an instance and can contain cloud credentials for that node, or provisioning data such as kubelet credentials. These credentials can be used to escalate within the cluster or to other cloud services under the same account.

When running Kubernetes on a cloud platform limit permissions given to instance credentials, use network policies to restrict pod access to the metadata API, and avoid using provisioning data to deliver secrets.

### Controlling which nodes pods may access

By default, there are no restrictions on which nodes may run a pod. Kubernetes offers a rich set of policies for controlling placement of pods onto nodes and the taint based pod placement and eviction that are available to end users. For many clusters use of these policies to separate workloads can be a convention that authors adopt or enforce via tooling.

As an administrator, a beta admission plugin `PodNodeSelector` can be used to force pods within a namespace to default or require a specific node selector, and if end users cannot alter namespaces, this can strongly limit the placement of all of the pods in a specific workload.

## Protecting cluster components from compromise

This section describes some common patterns for protecting clusters from compromise.

### Restrict access to etcd

Write access to the etcd backend for the API is equivalent to gaining root on the entire cluster, and read access can be used to escalate fairly quickly. Administrators should always use strong credentials from the API servers to their etcd server, such as mutual auth via TLS client certificates, and it is often recommended to isolate the etcd servers behind a firewall that only the API servers may access.

> **Caution:** Allowing other components within the cluster to access the master etcd instance with read or write access to the full keyspace is equivalent to granting cluster-admin access. Using separate etcd instances for non-master components or using etcd ACLs to restrict read and write access to a subset of the keyspace is strongly recommended.

### Enable audit logging

The audit logger is a beta feature that records actions taken by the API for later analysis in the event of a compromise. It is recommended to enable audit logging and archive the audit file on a secure server.

### Restrict access to alpha or beta features

Alpha and beta Kubernetes features are in active development and may have limitations or bugs that result in security vulnerabilities. Always assess the value an alpha or beta feature may provide against the possible risk to your security posture. When in doubt, disable features you do not use.

### Rotate infrastructure credentials frequently

The shorter the lifetime of a secret or credential the harder it is for an attacker to make use of that credential. Set short lifetimes on certificates and automate their rotation. Use an authentication provider that can control how long issued tokens are available and use short lifetimes where possible. If you use service account tokens in external integrations, plan to rotate those tokens frequently. For example, once the bootstrap phase is complete, a bootstrap token used for setting up nodes should be revoked or its authorization removed.

### Review third party integrations before enabling them

Many third party integrations to Kubernetes may alter the security profile of your cluster. When enabling an integration, always review the permissions that an extension requests before granting it access. For example, many security integrations may request access to view all secrets on your cluster which is effectively making that component a cluster admin. When in doubt, restrict the integration to functioning in a single namespace if possible.

Components that create pods may also be unexpectedly powerful if they can do so inside namespaces like the `kube-system` namespace, because those pods can gain access to service account secrets or run with elevated permissions if those service accounts are granted access to permissive pod security policies.

### Encrypt secrets at rest

In general, the etcd database will contain any information accessible via the Kubernetes API and may grant an attacker significant visibility into the state of your cluster. Always encrypt your backups using a well reviewed backup and encryption solution, and consider using full disk encryption where possible.

Kubernetes supports encryption at rest, a feature introduced in 1.7, and beta since 1.13. This will encrypt `Secret` resources in etcd, preventing parties that gain access to your etcd backups from viewing the content of those secrets. While this feature is currently beta, it offers an additional level of defense when backups are not encrypted or an attacker gains read access to etcd.

# Receiving alerts for security updates and reporting vulnerabilities

Join the kubernetes-announce group for emails about security announcements. See the security reporting page for more on how to report vulnerabilities.

# 35 - Set Kubelet parameters via a config file

A subset of the Kubelet's configuration parameters may be set via an on-disk config file, as a substitute for command-line flags.

Providing parameters via a config file is the recommended approach because it simplifies node deployment and configuration management.

## Create the config file

The subset of the Kubelet's configuration that can be configured via a file is defined by the `KubeletConfiguration` struct.

The configuration file must be a JSON or YAML representation of the parameters in this struct. Make sure the Kubelet has read permissions on the file.

Here is an example of what this file might look like:

```
apiVersion: kubelet.config.k8s.io/v1beta1
kind: KubeletConfiguration
evictionHard:
    memory.available:  "200Mi"
```

In the example, the Kubelet is configured to evict Pods when available memory drops below 200Mi. All other Kubelet configuration values are left at their built-in defaults, unless overridden by flags. Command line flags which target the same value as a config file will override that value.

For a trick to generate a configuration file from a live node, see Reconfigure a Node's Kubelet in a Live Cluster.

## Start a Kubelet process configured via the config file

> **Note:** If you use kubeadm to initialize your cluster, use the kubelet-config while creating your cluster with `kubeadmin init`. See configuring kubelet using kubeadm for details.

Start the Kubelet with the `--config` flag set to the path of the Kubelet's config file. The Kubelet will then load its config from this file.

Note that command line flags which target the same value as a config file will override that value. This helps ensure backwards compatibility with the command-line API.

Note that relative file paths in the Kubelet config file are resolved relative to the location of the Kubelet config file, whereas relative paths in command line flags are resolved relative to the Kubelet's current working directory.

Note that some default values differ between command-line flags and the Kubelet config file. If `--config` is provided and the values are not specified via the command line, the defaults for the `KubeletConfiguration` version apply. In the above example, this version is `kubelet.config.k8s.io/v1beta1`.

## Relationship to Dynamic Kubelet Config

If you are using the Dynamic Kubelet Configuration feature, the combination of configuration provided via `--config` and any flags which override these values is considered the default "last known good" configuration by the automatic rollback mechanism.

## What's next

- Learn more about kubelet configuration by checking the `KubeletConfiguration` reference.

# 36 - Set up a High-Availability Control Plane

**FEATURE STATE:** `Kubernetes v1.5 [alpha]`

You can replicate Kubernetes control plane nodes in `kube-up` or `kube-down` scripts for Google Compute Engine. This document describes how to use kube-up/down scripts to manage a highly available (HA) control plane and how HA control planes are implemented for use with GCE.

## Before you begin

You need to have a Kubernetes cluster, and the kubectl command-line tool must be configured to communicate with your cluster. It is recommended to run this tutorial on a cluster with at least two nodes that are not acting as control plane hosts. If you do not already have a cluster, you can create one by using minikube or you can use one of these Kubernetes playgrounds:

- Katacoda
- Play with Kubernetes

To check the version, enter `kubectl version`.

## Starting an HA-compatible cluster

To create a new HA-compatible cluster, you must set the following flags in your `kube-up` script:

- `MULTIZONE=true` - to prevent removal of control plane kubelets from zones different than server's default zone. Required if you want to run control plane nodes in different zones, which is recommended.
- `ENABLE_ETCD_QUORUM_READ=true` - to ensure that reads from all API servers will return most up-to-date data. If true, reads will be directed to leader etcd replica. Setting this value to true is optional: reads will be more reliable but will also be slower.

Optionally, you can specify a GCE zone where the first control plane node is to be created. Set the following flag:

- `KUBE_GCE_ZONE=zone` - zone where the first control plane node will run.

The following sample command sets up a HA-compatible cluster in the GCE zone europe-west1-b:

```
MULTIZONE=true KUBE_GCE_ZONE=europe-west1-b  ENABLE_ETCD_QUORUM_READS=true ./cluster/kube-up.sh
```

Note that the commands above create a cluster with one control plane node; however, you can add new control plane nodes to the cluster with subsequent commands.

## Adding a new control plane node

After you have created an HA-compatible cluster, you can add control plane nodes to it. You add control plane nodes by using a `kube-up` script with the following flags:

- `KUBE_REPLICATE_EXISTING_MASTER=true` - to create a replica of an existing control plane node.
- `KUBE_GCE_ZONE=zone` - zone where the control plane node will run. Must be in the same region as other control plane nodes' zones.

You don't need to set the `MULTIZONE` or `ENABLE_ETCD_QUORUM_READS` flags, as those are inherited from when you started your HA-compatible cluster.

The following sample command replicates the control plane node on an existing HA-compatible cluster:

```
KUBE_GCE_ZONE=europe-west1-c KUBE_REPLICATE_EXISTING_MASTER=true ./cluster/kube-up.sh
```

## Removing a control plane node

You can remove a control plane node from an HA cluster by using a `kube-down` script with the following flags:

- `KUBE_DELETE_NODES=false` - to restrain deletion of kubelets.
- `KUBE_GCE_ZONE=zone` - the zone from where the control plane node will be removed.
- `KUBE_REPLICA_NAME=replica_name` - (optional) the name of control plane node to remove. If empty: any replica from the given zone will be removed.

The following sample command removes a control plane node from an existing HA cluster:

```
KUBE_DELETE_NODES=false KUBE_GCE_ZONE=europe-west1-c ./cluster/kube-down.sh
```

# Handling control plane node failures

If one of the control plane nodes in your HA cluster fails, the best practice is to remove the node from your cluster and add a new control plane node in the same zone. The following sample commands demonstrate this process:

1. Remove the broken replica:

```
KUBE_DELETE_NODES=false KUBE_GCE_ZONE=replica_zone KUBE_REPLICA_NAME=replica_name ./cluster/kube-
```

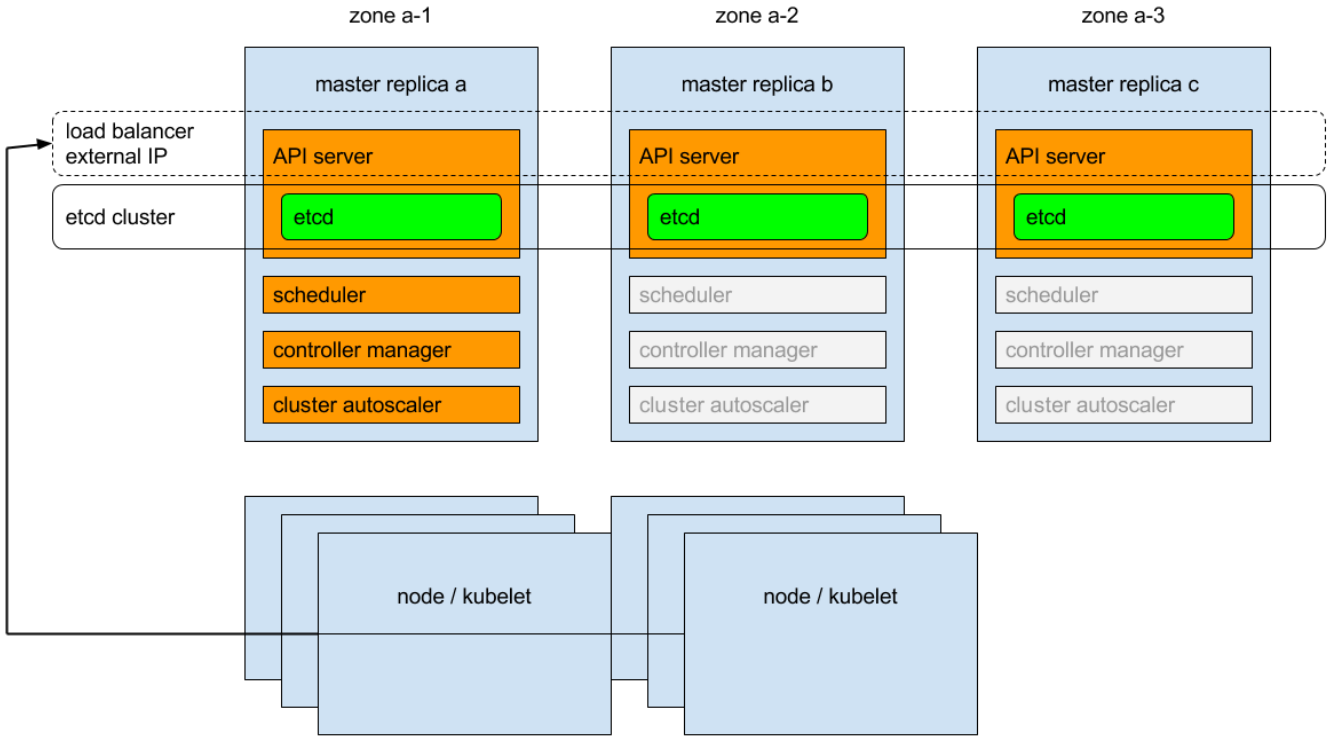
2. Add a new node in place of the old one:

```
KUBE_GCE_ZONE=replica-zone KUBE_REPLICATE_EXISTING_MASTER=true ./cluster/kube-up.sh
```

# Best practices for replicating control plane nodes for HA clusters

- Try to place control plane nodes in different zones. During a zone failure, all control plane nodes placed inside the zone will fail. To survive zone failure, also place nodes in multiple zones (see multiple-zones for details).
- Do not use a cluster with two control plane nodes. Consensus on a two-node control plane requires both nodes running when changing persistent state. As a result, both nodes are needed and a failure of any node turns the cluster into majority failure state. A two-node control plane is thus inferior, in terms of HA, to a cluster with one control plane node.
- When you add a control plane node, cluster state (etcd) is copied to a new instance. If the cluster is large, it may take a long time to duplicate its state. This operation may be sped up by migrating the etcd data directory, as described in the etcd administration guide (we are considering adding support for etcd data dir migration in the future).

# Implementation notes




## Overview

Each of the control plane nodes will run the following components in the following mode:

- etcd instance: all instances will be clustered together using consensus;
- API server: each server will talk to local etcd - all API servers in the cluster will be available;
- controllers, scheduler, and cluster auto-scaler: will use lease mechanism - only one instance of each of them will be active in the cluster;
- add-on manager: each manager will work independently trying to keep add-ons in sync.

In addition, there will be a load balancer in front of API servers that will route external and internal traffic to them.

## Load balancing

When starting the second control plane node, a load balancer containing the two replicas will be created and the IP address of the first replica will be promoted to IP address of load balancer. Similarly, after removal of the penultimate control plane node, the load balancer will be removed and its IP address will be assigned to the last remaining replica. Please note that creation and removal of load balancer are complex operations and it may take some time (~20 minutes) for them to propagate.

## Control plane service & kubelets

Instead of trying to keep an up-to-date list of Kubernetes apiserver in the Kubernetes service, the system directs all traffic to the external IP:

- in case of a single node control plane, the IP points to the control plane node,
- in case of an HA control plane, the IP points to the load balancer in-front of the control plane nodes.

Similarly, the external IP will be used by kubelets to communicate with the control plane.

## Control plane node certificates

Kubernetes generates TLS certificates for the external public IP and local IP for each control plane node. There are no certificates for the ephemeral public IP for control plane nodes; to access a control plane node via its ephemeral public IP, you must skip TLS verification.

## Clustering etcd

To allow etcd clustering, ports needed to communicate between etcd instances will be opened (for inside cluster communication). To make such deployment secure, communication between etcd instances is authorized using SSL.

## API server identity

**FEATURE STATE:** `Kubernetes v1.20 [alpha]`

The API Server Identity feature is controlled by a feature gate and is not enabled by default. You can activate API Server Identity by enabling the feature gate named `APIServerIdentity` when you start the API Server:

```
kube-apiserver \
--feature-gates=APIServerIdentity=true \
 # …and other flags as usual
```

During bootstrap, each kube-apiserver assigns a unique ID to itself. The ID is in the format of `kube-apiserver-{UUID}`. Each kube-apiserver creates a Lease in the *kube-system* namespaces. The Lease name is the unique ID for the kube-apiserver. The Lease contains a label `k8s.io/component=kube-apiserver`. Each kube-apiserver refreshes its Lease every `IdentityLeaseRenewIntervalSeconds` (defaults to 10s). Each kube-apiserver also checks all the kube-apiserver identity Leases every `IdentityLeaseDurationSeconds` (defaults to 3600s), and deletes Leases that hasn't got refreshed for more than `IdentityLeaseDurationSeconds`. `IdentityLeaseRenewIntervalSeconds` and `IdentityLeaseDurationSeconds` can be configured by kube-apiserver flags `identity-lease-renew-interval-seconds` and `identity-lease-duration-seconds`.

Enabling this feature is a prerequisite for using features that involve HA API server coordination (for example, the `StorageVersionAPI` feature gate).

# Additional reading

Automated HA master deployment - design doc

# 37 - Share a Cluster with Namespaces

This page shows how to view, work in, and delete namespaces. The page also shows how to use Kubernetes namespaces to subdivide your cluster.

## Before you begin

- Have an existing Kubernetes cluster.
- You have a basic understanding of Kubernetes Pods, Services, and Deployments.

## Viewing namespaces

1. List the current namespaces in a cluster using:

```
kubectl get namespaces
```

```
NAME          STATUS    AGE
default       Active    11d
kube-system   Active    11d
kube-public   Active    11d
```

Kubernetes starts with three initial namespaces:

- `default` The default namespace for objects with no other namespace
- `kube-system` The namespace for objects created by the Kubernetes system
- `kube-public` This namespace is created automatically and is readable by all users (including those not authenticated). This namespace is mostly reserved for cluster usage, in case that some resources should be visible and readable publicly throughout the whole cluster. The public aspect of this namespace is only a convention, not a requirement.

You can also get the summary of a specific namespace using:

```
kubectl get namespaces <name>
```

Or you can get detailed information with:

```
kubectl describe namespaces <name>
```

```
Name:           default
Labels:         <none>
Annotations:    <none>
Status:         Active

No resource quota.

Resource Limits
 Type       Resource    Min Max Default
 ----               --------    --- --- ---
 Container          cpu         -   -   100m
```

Note that these details show both resource quota (if present) as well as resource limit ranges.

Resource quota tracks aggregate usage of resources in the *Namespace* and allows cluster operators to define *Hard* resource usage limits that a *Namespace* may consume.

A limit range defines min/max constraints on the amount of resources a single entity can consume in a *Namespace*.

See Admission control: Limit Range

A namespace can be in one of two phases:

- `Active` the namespace is in use
- `Terminating` the namespace is being deleted, and can not be used for new objects

See the design doc for more details.

# Creating a new namespace

</>

> **Note:** Avoid creating namespace with prefix `kube-`, since it is reserved for Kubernetes system namespaces.

1. Create a new YAML file called `my-namespace.yaml` with the contents:

```
apiVersion: v1
kind: Namespace
metadata:
  name: <insert-namespace-name-here>
```

   Then run:

```
kubectl create -f ./my-namespace.yaml
```

2. Alternatively, you can create namespace using below command:

```
kubectl create namespace <insert-namespace-name-here>
```

The name of your namespace must be a valid DNS label.

There's an optional field `finalizers`, which allows observables to purge resources whenever the namespace is deleted. Keep in mind that if you specify a nonexistent finalizer, the namespace will be created but will get stuck in the `Terminating` state if the user tries to delete it.

More information on `finalizers` can be found in the namespace design doc.

# Deleting a namespace

Delete a namespace with

```
kubectl delete namespaces <insert-some-namespace-name>
```

> **Warning:** This deletes *everything* under the namespace!

This delete is asynchronous, so for a time you will see the namespace in the `Terminating` state.

# Subdividing your cluster using Kubernetes namespaces

1. Understand the default namespace

   By default, a Kubernetes cluster will instantiate a default namespace when provisioning the cluster to hold the default set of Pods, Services, and Deployments used by the cluster.

   Assuming you have a fresh cluster, you can introspect the available namespaces by doing the following:

```
kubectl get namespaces
```

```
NAME      STATUS    AGE
default   Active    13m
```

2. Create new namespaces

   For this exercise, we will create two additional Kubernetes namespaces to hold our content.

   In a scenario where an organization is using a shared Kubernetes cluster for development and production use cases:

   The development team would like to maintain a space in the cluster where they can get a view on the list of Pods, Services, and Deployments they use to build and run their application. In this space, Kubernetes resources come and go, and the restrictions on who can or cannot modify resources are relaxed to enable agile development.

   The operations team would like to maintain a space in the cluster where they can enforce strict procedures on who can or cannot manipulate the set of Pods, Services, and Deployments that run the production site.

One pattern this organization could follow is to partition the Kubernetes cluster into two namespaces: `development` and `production`.

Let's create two new namespaces to hold our work.

Create the `development` namespace using kubectl:

```
kubectl create -f https://k8s.io/examples/admin/namespace-dev.json
```

And then let's create the `production` namespace using kubectl:

```
kubectl create -f https://k8s.io/examples/admin/namespace-prod.json
```

To be sure things are right, list all of the namespaces in our cluster.

```
kubectl get namespaces --show-labels
```

```
NAME          STATUS    AGE       LABELS
default       Active    32m       <none>
development   Active    29s       name=development
production    Active    23s       name=production
```

3. Create pods in each namespace

   A Kubernetes namespace provides the scope for Pods, Services, and Deployments in the cluster.

   Users interacting with one namespace do not see the content in another namespace.

   To demonstrate this, let's spin up a simple Deployment and Pods in the `development` namespace.

```
kubectl create deployment snowflake --image=k8s.gcr.io/serve_hostname  -n=development --repli
```

   We have created a deployment whose replica size is 2 that is running the pod called `snowflake` with a basic container that serves the hostname.

```
kubectl get deployment -n=development
```

```
NAME         READY   UP-TO-DATE   AVAILABLE   AGE
snowflake    2/2     2            2           2m
```

```
kubectl get pods -l app=snowflake -n=development
```

```
NAME                         READY     STATUS    RESTARTS   AGE
snowflake-3968820950-9dgr8   1/1       Running   0          2m
snowflake-3968820950-vgc4n   1/1       Running   0          2m
```

   And this is great, developers are able to do what they want, and they do not have to worry about affecting content in the `production` namespace.

   Let's switch to the `production` namespace and show how resources in one namespace are hidden from the other.

   The `production` namespace should be empty, and the following commands should return nothing.

```
kubectl get deployment -n=production
kubectl get pods -n=production
```

   Production likes to run cattle, so let's create some cattle pods.

```
kubectl create deployment cattle --image=k8s.gcr.io/serve_hostname -n=production
kubectl scale deployment cattle --replicas=5 -n=production

kubectl get deployment -n=production
```

```
NAME         READY   UP-TO-DATE   AVAILABLE   AGE
cattle       5/5     5            5           10s
```

```
kubectl get pods -l app=cattle -n=production
```

```
NAME                      READY     STATUS    RESTARTS   AGE
cattle-2263376956-41xy6   1/1       Running   0          34s
cattle-2263376956-kw466   1/1       Running   0          34s
cattle-2263376956-n4v97   1/1       Running   0          34s
cattle-2263376956-p5p3i   1/1       Running   0          34s
cattle-2263376956-sxpth   1/1       Running   0          34s
```

At this point, it should be clear that the resources users create in one namespace are hidden from the other namespace.

As the policy support in Kubernetes evolves, we will extend this scenario to show how you can provide different authorization rules for each namespace.

# Understanding the motivation for using namespaces

A single cluster should be able to satisfy the needs of multiple users or groups of users (henceforth a 'user community').

Kubernetes *namespaces* help different projects, teams, or customers to share a Kubernetes cluster.

It does this by providing the following:

1. A scope for Names.
2. A mechanism to attach authorization and policy to a subsection of the cluster.

Use of multiple namespaces is optional.

Each user community wants to be able to work in isolation from other communities.

Each user community has its own:

1. resources (pods, services, replication controllers, etc.)
2. policies (who can or cannot perform actions in their community)
3. constraints (this community is allowed this much quota, etc.)

A cluster operator may create a Namespace for each unique user community.

The Namespace provides a unique scope for:

1. named resources (to avoid basic naming collisions)
2. delegated management authority to trusted users
3. ability to limit community resource consumption

Use cases include:

1. As a cluster operator, I want to support multiple user communities on a single cluster.
2. As a cluster operator, I want to delegate authority to partitions of the cluster to trusted users in those communities.
3. As a cluster operator, I want to limit the amount of resources each community can consume in order to limit the impact to other communities using the cluster.
4. As a cluster user, I want to interact with resources that are pertinent to my user community in isolation of what other user communities are doing on the cluster.

# Understanding namespaces and DNS

When you create a Service, it creates a corresponding DNS entry. This entry is of the form `<service-name>.<namespace-name>.svc.cluster.local`, which means that if a container uses `<service-name>` it will resolve to the service which is local to a namespace. This is useful for using the same configuration across multiple namespaces

such as Development, Staging and Production. If you want to reach across namespaces, you need to use the fully qualified domain name (FQDN).

## What's next

- Learn more about setting the namespace preference.
- Learn more about setting the namespace for a request
- See namespaces design.

# 38 - Upgrade A Cluster

This page provides an overview of the steps you should follow to upgrade a Kubernetes cluster.

The way that you upgrade a cluster depends on how you initially deployed it and on any subsequent changes.

At a high level, the steps you perform are:

- Upgrade the control plane
- Upgrade the nodes in your cluster
- Upgrade clients such as kubectl
- Adjust manifests and other resources based on the API changes that accompany the new Kubernetes version

## Before you begin

You must have an existing cluster. This page is about upgrading from Kubernetes 1.20 to Kubernetes 1.21. If your cluster is not currently running Kubernetes 1.20 then please check the documentation for the version of Kubernetes that you plan to upgrade to.

## Upgrade approaches

### kubeadm

If your cluster was deployed using the `kubeadm` tool, refer to Upgrading kubeadm clusters for detailed information on how to upgrade the cluster.

Once you have upgraded the cluster, remember to install the latest version of `kubectl`.

### Manual deployments

> **Caution:** These steps do not account for third-party extensions such as network and storage plugins.

You should manually update the control plane following this sequence:

- etcd (all instances)
- kube-apiserver (all control plane hosts)
- kube-controller-manager
- kube-scheduler
- cloud controller manager, if you use one

At this point you should install the latest version of `kubectl`.

For each node in your cluster, drain that node and then either replace it with a new node that uses the 1.21 kubelet, or upgrade the kubelet on that node and bring the node back into service.

### Other deployments

Refer to the documentation for your cluster deployment tool to learn the recommended set up steps for maintenance.

## Post-upgrade tasks

### Switch your cluster's storage API version

The objects that are serialized into etcd for a cluster's internal representation of the Kubernetes resources active in the cluster are written using a particular version of the API.

When the supported API changes, these objects may need to be rewritten in the newer API. Failure to do this will eventually result in resources that are no longer decodable or usable by the Kubernetes API server.

For each affected object, fetch it using the latest supported API and then write it back also using the latest supported API.

### Update manifests

Upgrading to a new Kubernetes version can provide new APIs.

You can use `kubectl convert` command to convert manifests between different API versions. For example:

```
kubectl convert -f pod.yaml --output-version v1
```

The `kubectl` tool replaces the contents of `pod.yaml` with a manifest that sets `kind` to Pod (unchanged), but with a revised `apiVersion`.

# 39 - Using a KMS provider for data encryption

This page shows how to configure a Key Management Service (KMS) provider and plugin to enable secret data encryption.

## Before you begin

- You need to have a Kubernetes cluster, and the kubectl command-line tool must be configured to communicate with your cluster. It is recommended to run this tutorial on a cluster with at least two nodes that are not acting as control plane hosts. If you do not already have a cluster, you can create one by using minikube or you can use one of these Kubernetes playgrounds:
  - Katacoda
  - Play with Kubernetes

  To check the version, enter `kubectl version`.
- Kubernetes version 1.10.0 or later is required
- etcd v3 or later is required

**FEATURE STATE:** `Kubernetes v1.12 [beta]`

The KMS encryption provider uses an envelope encryption scheme to encrypt data in etcd. The data is encrypted using a data encryption key (DEK); a new DEK is generated for each encryption. The DEKs are encrypted with a key encryption key (KEK) that is stored and managed in a remote KMS. The KMS provider uses gRPC to communicate with a specific KMS plugin. The KMS plugin, which is implemented as a gRPC server and deployed on the same host(s) as the Kubernetes master(s), is responsible for all communication with the remote KMS.

## Configuring the KMS provider

To configure a KMS provider on the API server, include a provider of type `kms` in the providers array in the encryption configuration file and set the following properties:

- `name`: Display name of the KMS plugin.
- `endpoint`: Listen address of the gRPC server (KMS plugin). The endpoint is a UNIX domain socket.
- `cachesize`: Number of data encryption keys (DEKs) to be cached in the clear. When cached, DEKs can be used without another call to the KMS; whereas DEKs that are not cached require a call to the KMS to unwrap.
- `timeout`: How long should kube-apiserver wait for kms-plugin to respond before returning an error (default is 3 seconds).

See Understanding the encryption at rest configuration.

## Implementing a KMS plugin

To implement a KMS plugin, you can develop a new plugin gRPC server or enable a KMS plugin already provided by your cloud provider. You then integrate the plugin with the remote KMS and deploy it on the Kubernetes master.

### Enabling the KMS supported by your cloud provider

Refer to your cloud provider for instructions on enabling the cloud provider-specific KMS plugin.

### Developing a KMS plugin gRPC server

You can develop a KMS plugin gRPC server using a stub file available for Go. For other languages, you use a proto file to create a stub file that you can use to develop the gRPC server code.

- Using Go: Use the functions and data structures in the stub file: service.pb.go to develop the gRPC server code
- Using languages other than Go: Use the protoc compiler with the proto file: service.proto to generate a stub file for the specific language

Then use the functions and data structures in the stub file to develop the server code.

**Notes:**

- kms plugin version: `v1beta1`

  In response to procedure call Version, a compatible KMS plugin should return v1beta1 as VersionResponse.version.
- message version: `v1beta1`

  All messages from KMS provider have the version field set to current version v1beta1.

- protocol: UNIX domain socket ( `unix` )

  The gRPC server should listen at UNIX domain socket.

### Integrating a KMS plugin with the remote KMS

The KMS plugin can communicate with the remote KMS using any protocol supported by the KMS. All configuration data, including authentication credentials the KMS plugin uses to communicate with the remote KMS, are stored and managed by the KMS plugin independently. The KMS plugin can encode the ciphertext with additional metadata that may be required before sending it to the KMS for decryption.

### Deploying the KMS plugin

Ensure that the KMS plugin runs on the same host(s) as the Kubernetes master(s).

## Encrypting your data with the KMS provider

To encrypt the data:

1. Create a new encryption configuration file using the appropriate properties for the `kms` provider:

```
apiVersion: apiserver.config.k8s.io/v1
kind: EncryptionConfiguration
resources:
  - resources:
      - secrets
    providers:
      - kms:
          name: myKmsPlugin
          endpoint: unix:///tmp/socketfile.sock
          cachesize: 100
          timeout: 3s
      - identity: {}
```

2. Set the `--encryption-provider-config` flag on the kube-apiserver to point to the location of the configuration file.

3. Restart your API server.

## Verifying that the data is encrypted

Data is encrypted when written to etcd. After restarting your `kube-apiserver` , any newly created or updated secret should be encrypted when stored. To verify, you can use the `etcdctl` command line program to retrieve the contents of your secret.

1. Create a new secret called secret1 in the default namespace:

```
kubectl create secret generic secret1 -n default --from-literal=mykey=mydata
```

2. Using the etcdctl command line, read that secret out of etcd:

```
ETCDCTL_API=3 etcdctl get /kubernetes.io/secrets/default/secret1 [...] | hexdump -C
```

   where `[...]` must be the additional arguments for connecting to the etcd server.

3. Verify the stored secret is prefixed with `k8s:enc:kms:v1:` , which indicates that the `kms` provider has encrypted the resulting data.

4. Verify that the secret is correctly decrypted when retrieved via the API:

```
kubectl describe secret secret1 -n default
```

   should match `mykey: mydata`

## Ensuring all secrets are encrypted

Because secrets are encrypted on write, performing an update on a secret encrypts that content.

The following command reads all secrets and then updates them to apply server side encryption. If an error occurs due to a conflicting write, retry the command. For larger clusters, you may wish to subdivide the secrets by namespace or script an update.

```
kubectl get secrets --all-namespaces -o json | kubectl replace -f -
```

# Switching from a local encryption provider to the KMS provider

To switch from a local encryption provider to the `kms` provider and re-encrypt all of the secrets:

1. Add the `kms` provider as the first entry in the configuration file as shown in the following example.

```
apiVersion: apiserver.config.k8s.io/v1
kind: EncryptionConfiguration
resources:
  - resources:
      - secrets
    providers:
      - kms:
          name : myKmsPlugin
          endpoint: unix:///tmp/socketfile.sock
          cachesize: 100
      - aescbc:
          keys:
            - name: key1
              secret: <BASE 64 ENCODED SECRET>
```

2. Restart all kube-apiserver processes.

3. Run the following command to force all secrets to be re-encrypted using the `kms` provider.

```
kubectl get secrets --all-namespaces -o json| kubectl replace -f -
```

# Disabling encryption at rest

To disable encryption at rest:

1. Place the `identity` provider as the first entry in the configuration file:

```
apiVersion: apiserver.config.k8s.io/v1
kind: EncryptionConfiguration
resources:
  - resources:
      - secrets
    providers:
      - identity: {}
      - kms:
          name : myKmsPlugin
          endpoint: unix:///tmp/socketfile.sock
          cachesize: 100
```

2. Restart all kube-apiserver processes.

3. Run the following command to force all secrets to be decrypted.

```
kubectl get secrets --all-namespaces -o json | kubectl replace -f -
```

# 40 - Using CoreDNS for Service Discovery

This page describes the CoreDNS upgrade process and how to install CoreDNS instead of kube-dns.

## Before you begin

You need to have a Kubernetes cluster, and the kubectl command-line tool must be configured to communicate with your cluster. It is recommended to run this tutorial on a cluster with at least two nodes that are not acting as control plane hosts. If you do not already have a cluster, you can create one by using minikube or you can use one of these Kubernetes playgrounds:

- Katacoda
- Play with Kubernetes

Your Kubernetes server must be at or later than version v1.9. To check the version, enter `kubectl version`.

## About CoreDNS

CoreDNS is a flexible, extensible DNS server that can serve as the Kubernetes cluster DNS. Like Kubernetes, the CoreDNS project is hosted by the CNCF.

You can use CoreDNS instead of kube-dns in your cluster by replacing kube-dns in an existing deployment, or by using tools like kubeadm that will deploy and upgrade the cluster for you.

## Installing CoreDNS

For manual deployment or replacement of kube-dns, see the documentation at the CoreDNS GitHub project.

## Migrating to CoreDNS

### Upgrading an existing cluster with kubeadm

In Kubernetes version 1.10 and later, you can also move to CoreDNS when you use `kubeadm` to upgrade a cluster that is using `kube-dns`. In this case, `kubeadm` will generate the CoreDNS configuration ("Corefile") based upon the `kube-dns` ConfigMap, preserving configurations for stub domains, and upstream name server.

If you are moving from kube-dns to CoreDNS, make sure to set the `CoreDNS` feature gate to `true` during an upgrade. For example, here is what a `v1.11.0` upgrade would look like:

```
kubeadm upgrade apply v1.11.0 --feature-gates=CoreDNS=true
```

In Kubernetes version 1.13 and later the `CoreDNS` feature gate is removed and CoreDNS is used by default.

In versions prior to 1.11 the Corefile will be **overwritten** by the one created during upgrade. **You should save your existing ConfigMap if you have customized it.** You may re-apply your customizations after the new ConfigMap is up and running.

If you are running CoreDNS in Kubernetes version 1.11 and later, during upgrade, your existing Corefile will be retained.

In Kubernetes version 1.21, support for `kube-dns` is removed from kubeadm.

## Upgrading CoreDNS

CoreDNS is available in Kubernetes since v1.9. You can check the version of CoreDNS shipped with Kubernetes and the changes made to CoreDNS here.

CoreDNS can be upgraded manually in case you want to only upgrade CoreDNS or use your own custom image. There is a helpful guideline and walkthrough available to ensure a smooth upgrade.

## Tuning CoreDNS

When resource utilisation is a concern, it may be useful to tune the configuration of CoreDNS. For more details, check out the documentation on scaling CoreDNS.

## What's next

You can configure CoreDNS to support many more use cases than kube-dns by modifying the `Corefile`. For more information, see the CoreDNS site.

# 41 - Using NodeLocal DNSCache in Kubernetes clusters

**FEATURE STATE:** `Kubernetes v1.18 [stable]`

This page provides an overview of NodeLocal DNSCache feature in Kubernetes.

## Before you begin

You need to have a Kubernetes cluster, and the kubectl command-line tool must be configured to communicate with your cluster. It is recommended to run this tutorial on a cluster with at least two nodes that are not acting as control plane hosts. If you do not already have a cluster, you can create one by using minikube or you can use one of these Kubernetes playgrounds:

- Katacoda
- Play with Kubernetes

To check the version, enter `kubectl version`.

## Introduction

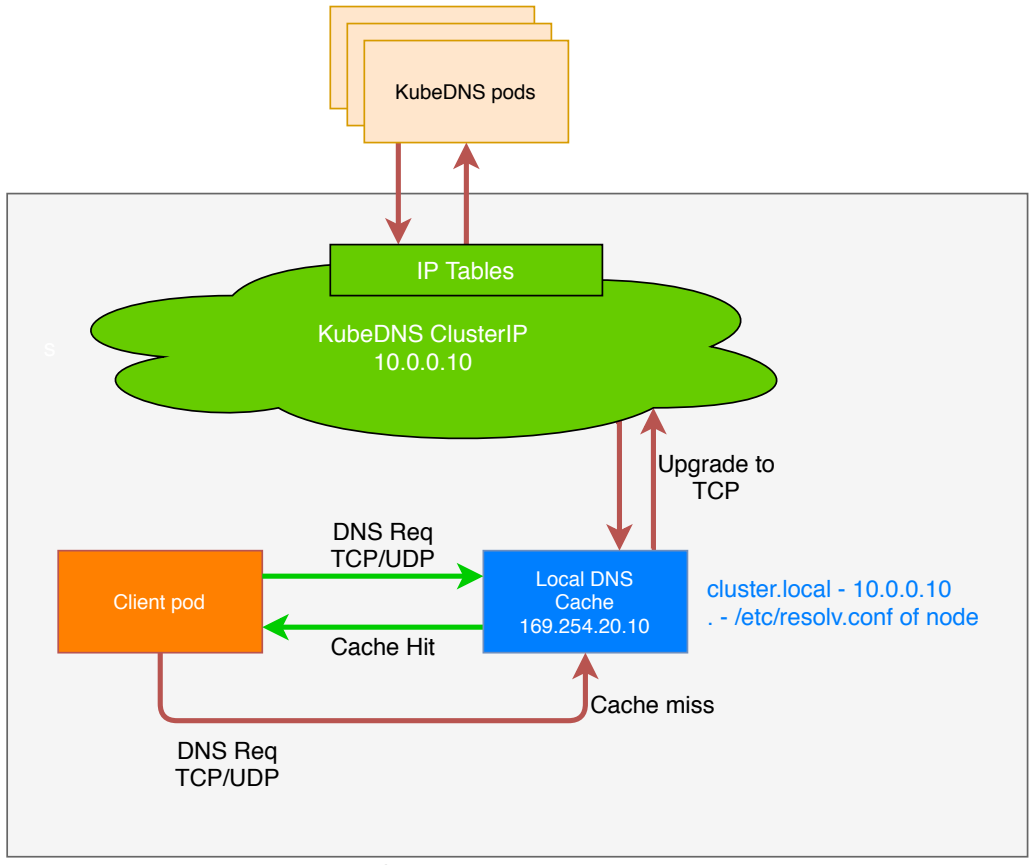
NodeLocal DNSCache improves Cluster DNS performance by running a dns caching agent on cluster nodes as a DaemonSet. In today's architecture, Pods in ClusterFirst DNS mode reach out to a kube-dns serviceIP for DNS queries. This is translated to a kube-dns/CoreDNS endpoint via iptables rules added by kube-proxy. With this new architecture, Pods will reach out to the dns caching agent running on the same node, thereby avoiding iptables DNAT rules and connection tracking. The local caching agent will query kube-dns service for cache misses of cluster hostnames(cluster.local suffix by default).

## Motivation

- With the current DNS architecture, it is possible that Pods with the highest DNS QPS have to reach out to a different node, if there is no local kube-dns/CoreDNS instance. Having a local cache will help improve the latency in such scenarios.
- Skipping iptables DNAT and connection tracking will help reduce conntrack races and avoid UDP DNS entries filling up conntrack table.
- Connections from local caching agent to kube-dns service can be upgraded to TCP. TCP conntrack entries will be removed on connection close in contrast with UDP entries that have to timeout (default `nf_conntrack_udp_timeout` is 30 seconds)
- Upgrading DNS queries from UDP to TCP would reduce tail latency attributed to dropped UDP packets and DNS timeouts usually up to 30s (3 retries + 10s timeout). Since the nodelocal cache listens for UDP DNS queries, applications don't need to be changed.
- Metrics & visibility into dns requests at a node level.
- Negative caching can be re-enabled, thereby reducing number of queries to kube-dns service.

## Architecture Diagram

This is the path followed by DNS Queries after NodeLocal DNSCache is enabled:




Nodelocal DNSCache flow

This image shows how NodeLocal DNSCache handles DNS queries.

# Configuration

> **Note:** The local listen IP address for NodeLocal DNSCache can be any address that can be guaranteed to not collide with any existing IP in your cluster. It's recommended to use an address with a local scope, per example, from the link-local range 169.254.0.0/16 for IPv4 or from the Unique Local Address range in IPv6 fd00::/8.

This feature can be enabled using the following steps:

- Prepare a manifest similar to the sample `nodelocaldns.yaml` and save it as `nodelocaldns.yaml.`
- If using IPv6, the CoreDNS configuration file need to enclose all the IPv6 addresses into square brackets if used in IP:Port format. If you are using the sample manifest from the previous point, this will require to modify the configuration line L70 like this `health [__PILLAR__LOCAL__DNS__]:8080`
- Substitute the variables in the manifest with the right values:
  - kubedns=`kubectl get svc kube-dns -n kube-system -o jsonpath={.spec.clusterIP}`
  - domain=`<cluster-domain>`
  - localdns=`<node-local-address>`

  `<cluster-domain>` is "cluster.local" by default. `<node-local-address>` is the local listen IP address chosen for NodeLocal DNSCache.
  - If kube-proxy is running in IPTABLES mode:

```
sed -i "s/__PILLAR__LOCAL__DNS__/$localdns/g; s/__PILLAR__DNS__DOMAIN__/$domain/g; s/__P
```

  `__PILLAR__CLUSTER__DNS__` and `__PILLAR__UPSTREAM__SERVERS__` will be populated by the node-local-dns pods. In this mode, node-local-dns pods listen on both the kube-dns service IP as well as `<node-local-address>`, so pods can lookup DNS records using either IP address.
  - If kube-proxy is running in IPVS mode:

```
sed -i "s/__PILLAR__LOCAL__DNS__/$localdns/g; s/__PILLAR__DNS__DOMAIN__/$domain/g; s/,_
```

  In this mode, node-local-dns pods listen only on `<node-local-address>`. The node-local-dns interface cannot bind the kube-dns cluster IP since the interface used for IPVS loadbalancing already uses this address. `__PILLAR__UPSTREAM__SERVERS__` will be populated by the node-local-dns pods.
- Run `kubectl create -f nodelocaldns.yaml`
- If using kube-proxy in IPVS mode, `--cluster-dns` flag to kubelet needs to be modified to use `<node-local-address>` that NodeLocal DNSCache is listening on. Otherwise, there is no need to modify the value of the `--cluster-dns` flag, since NodeLocal DNSCache listens on both the kube-dns service IP as well as `<node-local-address>`.

Once enabled, node-local-dns Pods will run in the kube-system namespace on each of the cluster nodes. This Pod runs CoreDNS in cache mode, so all CoreDNS metrics exposed by the different plugins will be available on a per-node basis.

You can disable this feature by removing the DaemonSet, using `kubectl delete -f <manifest>` . You should also revert any changes you made to the kubelet configuration.

# 42 - Using sysctls in a Kubernetes Cluster

**FEATURE STATE:** `Kubernetes v1.21 [stable]`

This document describes how to configure and use kernel parameters within a Kubernetes cluster using the sysctl interface.

## Before you begin

You need to have a Kubernetes cluster, and the kubectl command-line tool must be configured to communicate with your cluster. It is recommended to run this tutorial on a cluster with at least two nodes that are not acting as control plane hosts. If you do not already have a cluster, you can create one by using minikube or you can use one of these Kubernetes playgrounds:

- Katacoda
- Play with Kubernetes

For some steps, you also need to be able to reconfigure the command line options for the kubelets running on your cluster.

## Listing all Sysctl Parameters

In Linux, the sysctl interface allows an administrator to modify kernel parameters at runtime. Parameters are available via the `/proc/sys/` virtual process file system. The parameters cover various subsystems such as:

- kernel (common prefix: `kernel.`)
- networking (common prefix: `net.`)
- virtual memory (common prefix: `vm.`)
- MDADM (common prefix: `dev.`)
- More subsystems are described in Kernel docs.

To get a list of all parameters, you can run

```
sudo sysctl -a
```

## Enabling Unsafe Sysctls

Sysctls are grouped into *safe* and *unsafe* sysctls. In addition to proper namespacing, a *safe* sysctl must be properly *isolated* between pods on the same node. This means that setting a *safe* sysctl for one pod

- must not have any influence on any other pod on the node
- must not allow to harm the node's health
- must not allow to gain CPU or memory resources outside of the resource limits of a pod.

By far, most of the *namespaced* sysctls are not necessarily considered *safe*. The following sysctls are supported in the *safe* set:

- `kernel.shm_rmid_forced`,
- `net.ipv4.ip_local_port_range`,
- `net.ipv4.tcp_syncookies`,
- `net.ipv4.ping_group_range` (since Kubernetes 1.18).

> **Note:** The example `net.ipv4.tcp_syncookies` is not namespaced on Linux kernel version 4.4 or lower.

This list will be extended in future Kubernetes versions when the kubelet supports better isolation mechanisms.

All *safe* sysctls are enabled by default.

All *unsafe* sysctls are disabled by default and must be allowed manually by the cluster admin on a per-node basis. Pods with disabled unsafe sysctls will be scheduled, but will fail to launch.

With the warning above in mind, the cluster admin can allow certain *unsafe* sysctls for very special situations such as high-performance or real-time application tuning. *Unsafe* sysctls are enabled on a node-by-node basis with a flag of the kubelet; for example:

```
kubelet --allowed-unsafe-sysctls \
  'kernel.msg*,net.core.somaxconn' ...
```

For Minikube, this can be done via the `extra-config` flag:

```
minikube start --extra-config="kubelet.allowed-unsafe-sysctls=kernel.msg*,net.core.somaxconn"...
```

Only *namespaced* sysctls can be enabled this way.

# Setting Sysctls for a Pod

A number of sysctls are *namespaced* in today's Linux kernels. This means that they can be set independently for each pod on a node. Only namespaced sysctls are configurable via the pod securityContext within Kubernetes.

The following sysctls are known to be namespaced. This list could change in future versions of the Linux kernel.

- `kernel.shm*`,
- `kernel.msg*`,
- `kernel.sem`,
- `fs.mqueue.*`,
- The parameters under `net.*` that can be set in container networking namespace. However, there are exceptions (e.g., `net.netfilter.nf_conntrack_max` and `net.netfilter.nf_conntrack_expect_max` can be set in container networking namespace but they are unnamespaced).

Sysctls with no namespace are called *node-level* sysctls. If you need to set them, you must manually configure them on each node's operating system, or by using a DaemonSet with privileged containers.

Use the pod securityContext to configure namespaced sysctls. The securityContext applies to all containers in the same pod.

This example uses the pod securityContext to set a safe sysctl `kernel.shm_rmid_forced` and two unsafe sysctls `net.core.somaxconn` and `kernel.msgmax`. There is no distinction between *safe* and *unsafe* sysctls in the specification.

> **Warning:** Only modify sysctl parameters after you understand their effects, to avoid destabilizing your operating system.

```
apiVersion: v1
kind: Pod
metadata:
  name: sysctl-example
spec:
  securityContext:
    sysctls:
    - name: kernel.shm_rmid_forced
      value: "0"
    - name: net.core.somaxconn
      value: "1024"
    - name: kernel.msgmax
      value: "65536"
  ...
```

> **Warning:** Due to their nature of being *unsafe*, the use of *unsafe* sysctls is at-your-own-risk and can lead to severe problems like wrong behavior of containers, resource shortage or complete breakage of a node.

It is good practice to consider nodes with special sysctl settings as *tainted* within a cluster, and only schedule pods onto them which need those sysctl settings. It is suggested to use the Kubernetes *taints and toleration* feature to implement this.

A pod with the *unsafe* sysctls will fail to launch on any node which has not enabled those two *unsafe* sysctls explicitly. As with *node-level* sysctls it is recommended to use *taints and toleration* feature or taints on nodes to schedule those pods onto the right nodes.

# PodSecurityPolicy

**FEATURE STATE:** `Kubernetes v1.21 [deprecated]`

You can further control which sysctls can be set in pods by specifying lists of sysctls or sysctl patterns in the `forbiddenSysctls` and/or `allowedUnsafeSysctls` fields of the PodSecurityPolicy. A sysctl pattern ends with a `*` character, such as `kernel.*`. A `*` character on its own matches all sysctls.

By default, all safe sysctls are allowed.

Both `forbiddenSysctls` and `allowedUnsafeSysctls` are lists of plain sysctl names or sysctl patterns (which end with `*`). The string `*` matches all sysctls.

The `forbiddenSysctls` field excludes specific sysctls. You can forbid a combination of safe and unsafe sysctls in the list. To forbid setting any sysctls, use `*` on its own.

If you specify any unsafe sysctl in the `allowedUnsafeSysctls` field and it is not present in the `forbiddenSysctls` field, that sysctl can be used in Pods using this PodSecurityPolicy. To allow all unsafe sysctls in the PodSecurityPolicy to be set, use `*` on its own.

Do not configure these two fields such that there is overlap, meaning that a given sysctl is both allowed and forbidden.

> **Warning:** If you allow unsafe sysctls via the `allowedUnsafeSysctls` field in a PodSecurityPolicy, any pod using such a sysctl will fail to start if the sysctl is not allowed via the `--allowed-unsafe-sysctls` kubelet flag as well on that node.

This example allows unsafe sysctls prefixed with `kernel.msg` to be set and disallows setting of the `kernel.shm_rmid_forced` sysctl.

```
apiVersion: policy/v1beta1
kind: PodSecurityPolicy
metadata:
  name: sysctl-psp
spec:
  allowedUnsafeSysctls:
  - kernel.msg*
  forbiddenSysctls:
  - kernel.shm_rmid_forced
 ...
```