

[Edit This Page](#)

# Tasks

This section of the Kubernetes documentation contains pages that show how to do individual tasks. A task page shows how to do a single thing, typically by giving a short sequence of steps.

- [Web UI \(Dashboard\)](#)
- [Using the kubectl Command-line](#)
- [Configuring Pods and Containers](#)
- [Running Applications](#)
- [Running Jobs](#)
- [Accessing Applications in a Cluster](#)
- [Monitoring, Logging, and Debugging](#)
- [Accessing the Kubernetes API](#)
- [Using TLS](#)
- [Administering a Cluster](#)
- [Managing Stateful Applications](#)
- [Cluster Daemons](#)
- [Managing GPUs](#)
- [Managing HugePages](#)
- [What's next](#)

## Web UI (Dashboard)

Deploy and access the Dashboard web user interface to help you manage and monitor containerized applications in a Kubernetes cluster.

## Using the kubectl Command-line

Install and setup the kubectl command-line tool used to directly manage Kubernetes clusters.

## Configuring Pods and Containers

Perform common configuration tasks for Pods and Containers.

## Running Applications

Perform common application management tasks, such as rolling updates, injecting information into pods, and horizontal Pod autoscaling.

## Running Jobs

Run Jobs using parallel processing.

## **Accessing Applications in a Cluster**

Configure load balancing, port forwarding, or setup firewall or DNS configurations to access applications in a cluster.

## **Monitoring, Logging, and Debugging**

Setup monitoring and logging to troubleshoot a cluster or debug a containerized application.

## **Accessing the Kubernetes API**

Learn various methods to directly access the Kubernetes API.

## **Using TLS**

Configure your application to trust and use the cluster root Certificate Authority (CA).

## **Administering a Cluster**

Learn common tasks for administering a cluster.

## **Managing Stateful Applications**

Perform common tasks for managing Stateful applications, including scaling, deleting, and debugging StatefulSets.

## **Cluster Daemons**

Perform common tasks for managing a DaemonSet, such as performing a rolling update.

## **Managing GPUs**

Configure and schedule NVIDIA GPUs for use as a resource by nodes in a cluster.

## **Managing HugePages**

Configure and schedule huge pages as a schedulable resource in a cluster.

## What's next

If you would like to write a task page, see [Creating a Documentation Pull Request](#).

## Feedback

Was this page helpful?

Yes No

Thanks for the feedback. If you have a specific, answerable question about how to use Kubernetes, ask it on [Stack Overflow](#). Open an issue in the GitHub repo if you want to [report a problem](#) or [suggest an improvement](#).

---

[Create an Issue](#) [Edit This Page](#)

Page last modified on February 20, 2020 at 9:15 AM PST by [doc: remove tasks federation index from tasks main page. \(#19205\)](#) ([Page History](#))

[Edit This Page](#)

# Install and Set Up kubectl

The Kubernetes command-line tool, [kubectl](#), allows you to run commands against Kubernetes clusters. You can use kubectl to deploy applications, inspect and manage cluster resources, and view logs. For a complete list of kubectl operations, see [Overview of kubectl](#).

- [Before you begin](#)
- [Install kubectl on Linux](#)
- [Install kubectl on macOS](#)
- [Install kubectl on Windows](#)
- [Download as part of the Google Cloud SDK](#)
- [Verifying kubectl configuration](#)
- [Optional kubectl configurations](#)
- [What's next](#)

## Before you begin

You must use a kubectl version that is within one minor version difference of your cluster. For example, a v1.2 client should work with v1.1, v1.2, and v1.3 master. Using the latest version of kubectl helps avoid unforeseen issues.

## Install kubectl on Linux

### Install kubectl binary with curl on Linux

1. Download the latest release with the command:

```
curl -LO https://storage.googleapis.com/kubernetes-release/release/`curl -s https://storage.googleapis.com/kubernetes-release/release/stable.txt`/bin/linux/amd64/kubectl
```

To download a specific version, replace the `$(curl -s https://storage.googleapis.com/kubernetes-release/release/stable.txt)` portion of the command with the specific version.

For example, to download version v1.17.0 on Linux, type:

```
curl -LO https://storage.googleapis.com/kubernetes-release/release/v1.17.0/bin/linux/amd64/kubectl
```

2. Make the kubectl binary executable.

```
chmod +x ./kubectl
```

3. Move the binary in to your PATH.

```
sudo mv ./kubectl /usr/local/bin/kubectl
```

4. Test to ensure the version you installed is up-to-date:

```
kubectl version --client
```

## Install using native package management

- [Ubuntu, Debian or HypriotOS](#)
- [CentOS, RHEL or Fedora](#)

```
sudo apt-get update && sudo apt-get install -y apt-transport-  
https  
curl -s https://packages.cloud.google.com/apt/doc/apt-key.gpg |  
sudo apt-key add -  
echo "deb https://apt.kubernetes.io/ kubernetes-xenial main" |  
sudo tee -a /etc/apt/sources.list.d/kubernetes.list  
sudo apt-get update  
sudo apt-get install -y kubectl
```

[Edit This Page](#)

# Install Minikube

This page shows you how to install [Minikube](#), a tool that runs a single-node Kubernetes cluster in a virtual machine on your personal computer.

- [Before you begin](#)
- [Confirm Installation](#)
- [Clean up local state](#)
- [What's next](#)

## Before you begin

- [Linux](#)
- [macOS](#)
- [Windows](#)

To check if virtualization is supported on Linux, run the following command and verify that the output is non-empty:

```
grep -E --color 'vmx|svm' /proc/cpuinfo
```

To check if virtualization is supported on macOS, run the following command on your terminal.

```
sysctl -a | grep -E --color 'machdep.cpu.features|VMX'
```

If you see VMX in the output (should be colored), the VT-x feature is enabled in your machine.

To check if virtualization is supported on Windows 8 and above, run the following command on your Windows terminal or command prompt.

```
systeminfo
```

If you see the following output, virtualization is supported on Windows.

Hyper-V Requirements:	VM Monitor Mode Extensions: Yes
	Virtualization Enabled In Firmware: Yes
	Second Level Address Translation: Yes
	Data Execution Prevention Available: Yes

Yes

If you see the following output, your system already has a Hypervisor installed and you can skip the next step.

Hyper-V Requirements:	A hypervisor has been detected.
Features required for Hyper-V will not be displayed.	

[Edit This Page](#)

## Certificate Management with kubeadm

**FEATURE STATE:** Kubernetes v1.15 [stable](#)

This feature is *stable*, meaning:

[Edit This Page](#)

## Upgrading kubeadm clusters

This page explains how to upgrade a Kubernetes cluster created with kubeadm from version 1.16.x to version 1.17.x, and from version 1.17.x to 1.17.y (where y > x).

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

- [Upgrading kubeadm cluster from 1.15 to 1.16](#)
- [Upgrading kubeadm cluster from 1.14 to 1.15](#)
- [Upgrading kubeadm cluster from 1.13 to 1.14](#)

The upgrade workflow at high level is the following:

1. Upgrade the primary control plane node.
2. Upgrade additional control plane nodes.

### 3. Upgrade worker nodes.

- [Before you begin](#)
- [Determine which version to upgrade to](#)
- [Upgrading control plane nodes](#)
- [Upgrade worker nodes](#)
- [Verify the status of the cluster](#)
- [Recovering from a failure state](#)
- [How it works](#)

## Before you begin

- You need to have a kubeadm Kubernetes cluster running version 1.16.0 or later.
- [Swap must be disabled](#).
- The cluster should use a static control plane and etcd pods or external etcd.
- Make sure you read the [release notes](#) carefully.
- 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.

## Additional information

- All containers are restarted after upgrade, because the container spec hash value is changed.
- You only can upgrade from one MINOR version to the next MINOR version, or between PATCH versions of the same MINOR. That is, you cannot skip MINOR versions when you upgrade. For example, you can upgrade from 1.y to 1.y+1, but not from 1.y to 1.y+2.

## Determine which version to upgrade to

### 1. Find the latest stable 1.17 version:

- [Ubuntu, Debian or HypriotOS](#)
- [CentOS, RHEL or Fedora](#)

```
apt update
apt-cache madison kubeadm
# find the latest 1.17 version in the list
# it should look like 1.17.x-00, where x is the latest patch

yum list --showduplicates kubeadm --disableexcludes=kubernetes
# find the latest 1.17 version in the list
# it should look like 1.17.x-0, where x is the latest patch
```

[Edit This Page](#)

# 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](#)
- [Create a namespace](#)
- [Create a LimitRange and a Pod](#)
- [What if you specify a Container's limit, but not its request?](#)
- [What if you specify a Container's request, but not its limit?](#)
- [Motivation for default memory limits and requests](#)
- [Clean up](#)
- [What's next](#)

## Before you begin

You need to have a Kubernetes cluster, and the `kubectl` command-line tool must be configured to communicate with your cluster. 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](#)

```
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-mem-example
```

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](#)

```
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=default-mem-example
```

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](#)

```
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=default-mem-example
```

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=default-mem-example
```

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](#)

## Feedback

Was this page helpful?

Yes No

Thanks for the feedback. If you have a specific, answerable question about how to use Kubernetes, ask it on [Stack Overflow](#). Open an issue in the GitHub repo if you want to [report a problem](#) or [suggest an improvement](#).

---

[Create an Issue](#) [Edit This Page](#)

Page last modified on September 26, 2019 at 6:15 AM PST by [clean up the environment \(#16430\)](#) ([Page History](#))

[Edit This Page](#)

# 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](#)
- [Create a namespace](#)
- [Create a LimitRange and a Pod](#)
- [What if you specify a Container's limit, but not its request?](#)
- [What if you specify a Container's request, but not its limit?](#)
- [Motivation for default CPU limits and requests](#)
- [Clean up](#)
- [What's next](#)

## Before you begin

You need to have a Kubernetes cluster, and the `kubectl` command-line tool must be configured to communicate with your cluster. 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](#)

```
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-example
```

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](#)

```
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-cpu-example
```

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=default-cpu-example
```

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](#)

```
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=default-cpu-example
```

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](#)

## Feedback

Was this page helpful?

Yes No

Thanks for the feedback. If you have a specific, answerable question about how to use Kubernetes, ask it on [Stack Overflow](#). Open an issue in the GitHub repo if you want to [report a problem](#) or [suggest an improvement](#).

---

[Create an Issue](#) [Edit This Page](#)

Page last modified on September 26, 2019 at 6:15 AM PST by [clean up the environment \(#16430\)](#) ([Page History](#))

[Edit This Page](#)

# 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](#)
- [Create a namespace](#)
- [Create a LimitRange and a Pod](#)
- [Attempt to create a Pod that exceeds the maximum memory constraint](#)
- [Attempt to create a Pod that does not meet the minimum memory request](#)
- [Create a Pod that does not specify any memory request or limit](#)
- [Enforcement of minimum and maximum memory constraints](#)
- [Motivation for minimum and maximum memory constraints](#)
- [Clean up](#)
- [What's next](#)

## Before you begin

You need to have a Kubernetes cluster, and the kubectl command-line tool must be configured to communicate with your cluster. 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](#)

```
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=constraints-mem-example
```

View detailed information about the LimitRange:

```
kubectl get limitrange mem-min-max-demo-lr --namespace=constraint
s-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](#)

```
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=constraints-mem-example
```

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=co
nstraints-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.

```
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
```

```
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=constraints-mem-example
```

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-pod-2.yaml":
pods "constraints-mem-demo-2" is forbidden: maximum memory usage per Container is 1Gi, but limit is 1536Mi.
```

## 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](#)

```
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=constraints-mem-example
```

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-pod-3.yaml":
pods "constraints-mem-demo-3" is forbidden: minimum memory usage
per Container is 500Mi, but request is 100Mi.
```

## 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](#)

```
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=constraints-mem-example
```

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](#)

## Feedback

Was this page helpful?

Yes No

Thanks for the feedback. If you have a specific, answerable question about how to use Kubernetes, ask it on [Stack Overflow](#). Open an issue in the GitHub repo if you want to [report a problem](#) or [suggest an improvement](#).

---

[Create an Issue](#) [Edit This Page](#)

Page last modified on March 25, 2019 at 5:06 PM PST by [Official 1.14](#)  
[Release Docs \(#13174\)](#) ([Page History](#))

[Edit This Page](#)

# 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](#)
- [Create a namespace](#)
- [Create a LimitRange and a Pod](#)
- [Delete the Pod](#)
- [Attempt to create a Pod that exceeds the maximum CPU constraint](#)
- [Attempt to create a Pod that does not meet the minimum CPU request](#)
- [Create a Pod that does not specify any CPU request or limit](#)
- [Enforcement of minimum and maximum CPU constraints](#)
- [Motivation for minimum and maximum CPU constraints](#)
- [Clean up](#)
- [What's next](#)

## Before you begin

You need to have a Kubernetes cluster, and the kubectl command-line tool must be configured to communicate with your cluster. 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 CPU.

## 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=constraints-cpu-example
```

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=constraints-cpu-example
```

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=co
nstraints-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](#)

```
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=constraints-cpu-example
```

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.yaml": pods "constraints-cpu-demo-2" is forbidden: maximum cpu usage per Container is 800m, but limit is 1500m.
```

## 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](#)

```
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=constraints-cpu-example
```

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.yaml":
pods "constraints-cpu-demo-3" is forbidden: minimum cpu usage
per Container is 200m, but request is 100m.
```

## 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](#)

```
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=constraints-cpu-example
```

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?

```
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 Nodes have at least 1 CPU. 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](#)

## Feedback

Was this page helpful?

Yes No

Thanks for the feedback. If you have a specific, answerable question about how to use Kubernetes, ask it on [Stack Overflow](#). Open an issue in the GitHub repo if you want to [report a problem](#) or [suggest an improvement](#).

---

[Create an Issue](#) [Edit This Page](#)

Page last modified on September 24, 2019 at 3:49 PM PST by [keep content and file name to be consistent \(#16532\)](#) ([Page History](#))

[Edit This Page](#)

# 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](#)
- [Create a namespace](#)

- [Create a ResourceQuota](#)
- [Create a Pod](#)
- [Attempt to create a second Pod](#)
- [Discussion](#)
- [Clean up](#)
- [What's next](#)

## Before you begin

You need to have a Kubernetes cluster, and the `kubectl` command-line tool must be configured to communicate with your cluster. 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

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-cpu-example
```

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](#)

```
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-mem-cpu-example
```

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:

<a href="#">admin/resource/quota-mem-cpu-pod-2.yaml</a>
<pre>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"</pre>

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.  $600 \text{ MiB} + 700 \text{ MiB} > 1 \text{ GiB}$ .

Attempt to create the Pod:

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

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.yaml": 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](#)

## Feedback

Was this page helpful?

Yes No

Thanks for the feedback. If you have a specific, answerable question about how to use Kubernetes, ask it on [Stack Overflow](#). Open an issue in the GitHub repo if you want to [report a problem](#) or [suggest an improvement](#).

---

[Create an Issue](#) [Edit This Page](#)

Page last modified on March 25, 2019 at 5:06 PM PST by [Official 1.14](#)  
[Release Docs \(#13174\)](#) ([Page History](#))

[Edit This Page](#)

# 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](#)
- [Create a namespace](#)
- [Create a ResourceQuota](#)
- [Clean up](#)
- [What's next](#)

## Before you begin

You need to have a Kubernetes cluster, and the `kubectl` command-line tool must be configured to communicate with your cluster. 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:

<a href="#">admin/resource/quota-pod.yaml</a>
<pre>apiVersion: v1 kind: ResourceQuota metadata:   name: pod-demo spec:   hard:     pods: "2"</pre>

Create the ResourceQuota:

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

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.

```
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](#)

```
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=quota-pod-example
```

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.

```
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](#)

# Feedback

Was this page helpful?

Yes No

Thanks for the feedback. If you have a specific, answerable question about how to use Kubernetes, ask it on [Stack Overflow](#). Open an issue in the GitHub repo if you want to [report a problem](#) or [suggest an improvement](#).

---

[Create an Issue](#) [Edit This Page](#)

Page last modified on March 25, 2019 at 5:06 PM PST by [Official 1.14](#)  
[Release Docs \(#13174\)](#) ([Page History](#))

[Edit This Page](#)

# Use Calico for NetworkPolicy

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

- [Before you begin](#)
- [Creating a Calico cluster with Google Kubernetes Engine \(GKE\)](#)
- [Creating a local Calico cluster with kubeadm](#)
- [What's next](#)

## 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, just 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.

# Feedback

Was this page helpful?

Yes No

Thanks for the feedback. If you have a specific, answerable question about how to use Kubernetes, ask it on [Stack Overflow](#). Open an issue in the GitHub repo if you want to [report a problem](#) or [suggest an improvement](#).

---

[Create an Issue](#) [Edit This Page](#)

Page last modified on May 19, 2018 at 8:54 PM PST by [Add weights, move files: Tasks>AdminClust>NetPol. \(#8634\)](#) ([Page History](#))

[Edit This Page](#)

# 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](#)
- [Deploying Cilium on Minikube for Basic Testing](#)
- [Deploying Cilium for Production Use](#)
- [Understanding Cilium components](#)
- [What's next](#)

## Before you begin

You need to have a Kubernetes cluster, and the kubectl command-line tool must be configured to communicate with your cluster. 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  $\geq v1.3.1$ , 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 bpfss -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/v1.6/install/kubernetes/quick-install.yaml
```

```
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](#).

## Feedback

Was this page helpful?

Yes No

Thanks for the feedback. If you have a specific, answerable question about how to use Kubernetes, ask it on [Stack Overflow](#). Open an issue in the GitHub repo if you want to [report a problem](#) or [suggest an improvement](#).

---

[Create an Issue](#) [Edit This Page](#)

Page last modified on January 13, 2020 at 1:35 PM PST by [Update Cilium related docs \(#18563\)](#) ([Page History](#))

[Edit This Page](#)

# Use Kube-router for NetworkPolicy

This page shows how to use [Kube-router](#) for NetworkPolicy.

- [Before you begin](#)
- [Installing Kube-router addon](#)
- [What's next](#)

## 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.

## Feedback

Was this page helpful?

Yes No

Thanks for the feedback. If you have a specific, answerable question about how to use Kubernetes, ask it on [Stack Overflow](#). Open an issue in the GitHub repo if you want to [report a problem](#) or [suggest an improvement](#).

---

[Create an Issue](#) [Edit This Page](#)

Page last modified on June 02, 2018 at 5:08 AM PST by [kube-router](#)  
[document is moved \(dead link\) \(#8741\)](#) ([Page History](#))

[Edit This Page](#)

# Romana for NetworkPolicy

This page shows how to use Romana for NetworkPolicy.

- [Before you begin](#)
- [Installing Romana with kubeadm](#)
- [Applying network policies](#)
- [What's next](#)

# 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.

## Feedback

Was this page helpful?

Yes No

Thanks for the feedback. If you have a specific, answerable question about how to use Kubernetes, ask it on [Stack Overflow](#). Open an issue in the GitHub repo if you want to [report a problem](#) or [suggest an improvement](#).

---

[Create an Issue](#) [Edit This Page](#)

Page last modified on June 03, 2019 at 2:34 PM PST by [fix typo in romana-network-policy.md \(#14661\)](#) ([Page History](#))

[Edit This Page](#)

# Weave Net for NetworkPolicy

This page shows how to use Weave Net for NetworkPolicy.

- [Before you begin](#)
- [Install the Weave Net addon](#)
- [Test the installation](#)
- [What's next](#)

# 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	RESTARTS	AGE	IP	READY	STATUS
				NODE	
weave-net-1t1qg	0	9d	192.168.2.10	2/2	Running
weave-net-231d7	1	7d	10.2.0.17	2/2	Running
weave-net-7nmwt	3	9d	192.168.2.131	2/2	Running
weave-net-pmw8w	0	9d	192.168.2.216	2/2	Running

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](#).

## Feedback

Was this page helpful?

Yes No

Thanks for the feedback. If you have a specific, answerable question about how to use Kubernetes, ask it on [Stack Overflow](#). Open an issue in the GitHub repo if you want to [report a problem](#) or [suggest an improvement](#).

---

[Create an Issue](#) [Edit This Page](#)

Page last modified on September 25, 2018 at 5:50 AM PST by [Update weave-network-policy.md -- fixing typo. \(#9825\)](#) ([Page History](#))

[Edit This Page](#)

# Access Clusters Using the Kubernetes API

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

- [Before you begin](#)
- [Accessing 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. 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"}{.cluster.server}{"\n"}{end}

# 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.server")

# Gets the token value
TOKEN=$(kubectl get secrets -o jsonpath=".items[?(@.metadata.annotations['kubernetes\\.io/service-account\\.name']=='default')].data.token" | base64 --decode)

# 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') -o jsonpath='$.data.token' | base64 --decode )
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. [Configuring Access to the API](#) describes how a cluster admin can configure this. Such approaches may conflict with future high-availability support.

## 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](#):

```
import (
    "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(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](#):

```
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](#), simply execute :

```
# 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);
        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
```

## Accessing the API from within a Pod

When accessing the API from within a Pod, locating and authenticating to the API server are slightly different to the external client case described above.

The easiest way to use the Kubernetes API from a Pod is to use one of the official [client libraries](#). These libraries can automatically discover the API server and authenticate.

## Using Official Client Libraries

From within a Pod, the recommended ways to connect to the Kubernetes API are:

- For a Go client, use the official [Go client library](#). The `rest.InClusterConfig()` function handles API host discovery and authentication automatically. See [an example here](#).
- For a Python client, use the official [Python client library](#). The `config.load_incluster_config()` function handles API host discovery and authentication automatically. See [an example here](#).
- There are a number of other libraries available, please refer to the [Client Libraries](#) page.

In each case, the service account credentials of the Pod are used to communicate securely with the API server.

## Directly accessing the REST API

While running in a Pod, the Kubernetes apiserver is accessible via a Service named `kubernetes` in the default namespace. Therefore, Pods can use the

`kubernetes.default.svc` hostname to query the API server. Official client libraries do this automatically.

The recommended way to authenticate to the API server is with a [service account](#) credential. By default, a Pod is associated with a service account, and a credential (token) for that service account is placed into the filesystem tree of each container in that Pod, at `/var/run/secrets/kubernetes.io/serviceaccount/token`.

If available, a certificate bundle is placed into the filesystem tree of each container at `/var/run/secrets/kubernetes.io/serviceaccount/ca.crt`, and should be used to verify the serving certificate of the API server.

Finally, the default namespace to be used for namespaced API operations is placed in a file at `/var/run/secrets/kubernetes.io/serviceaccount/namespace` in each container.

## Using `kubectl proxy`

If you would like to query the API without an official client library, you can run `kubectl proxy` as the [command](#) of a new sidecar container in the Pod. This way, `kubectl proxy` will authenticate to the API and expose it on the `localhost` interface of the Pod, so that other containers in the Pod can use it directly.

## Without using a proxy

It is possible to avoid using the `kubectl proxy` by passing the authentication token directly to the API server. The internal certificate secures the connection.

```
# Point to the internal API server hostname
APISERVER=https://kubernetes.default.svc

# Path to ServiceAccount token
SERVICEACCOUNT=/var/run/secrets/kubernetes.io/serviceaccount

# Read this Pod's namespace
NAMESPACE=$(cat ${SERVICEACCOUNT}/namespace)

# Read the ServiceAccount bearer token
TOKEN=$(cat ${SERVICEACCOUNT}/token)

# Reference the internal certificate authority (CA)
CACERT=${SERVICEACCOUNT}/ca.crt

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

The output will be similar to this:

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

## Feedback

Was this page helpful?

Yes No

Thanks for the feedback. If you have a specific, answerable question about how to use Kubernetes, ask it on [Stack Overflow](#). Open an issue in the GitHub repo if you want to [report a problem](#) or [suggest an improvement](#).

---

[Create an Issue](#) [Edit This Page](#)

Page last modified on January 18, 2020 at 7:31 PM PST by [en: Fix minor semantical issue \(#18752\)](#) ([Page History](#))

[Edit This Page](#)

# Access Services Running on Clusters

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

- [Before you begin](#)
- [Accessing services running on the 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. 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 just 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/elasticsearch-logging/proxy
kibana-logging is running at https://104.197.5.247/api/v1/
namespaces/kube-system/services/kibana-logging/proxy
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-grafana/proxy
heapster is running at https://104.197.5.247/api/v1/namespaces/
kube-system/services/monitoring-heapster/proxy
```

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 simply append to the service's proxy URL: `http://kubernetes_master_address/api/v1/namespaces/name_space_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/_search?q=user:kimchy
```

- 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/_cluster/health?pretty=true
```

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/proxy/_cluster/health?pretty=true
```

## 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.

## Feedback

Was this page helpful?

Yes No

Thanks for the feedback. If you have a specific, answerable question about how to use Kubernetes, ask it on [Stack Overflow](#). Open an issue in the GitHub repo if you want to [report a problem](#) or [suggest an improvement](#).

[Create an Issue](#) [Edit This Page](#)

Page last modified on January 23, 2019 at 6:06 PM PST by [remove command prompts and separate commands from output \(#12338\)](#) ([Page History](#))

[Edit This Page](#)

# 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](#)
- [Get the names of your Nodes](#)

- [Advertise a new extended resource on one of your Nodes](#)
- [Discussion](#)
- [Clean up](#)
- [What's next](#)

## Before you begin

You need to have a Kubernetes cluster, and the `kubectl` command-line tool must be configured to communicate with your cluster. 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 just 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"}]\' \\\nhttp://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](#)

## Feedback

Was this page helpful?

Yes No

Thanks for the feedback. If you have a specific, answerable question about how to use Kubernetes, ask it on [Stack Overflow](#). Open an issue in the GitHub repo if you want to [report a problem](#) or [suggest an improvement](#).

---

[Create an Issue](#) [Edit This Page](#)

Page last modified on August 07, 2019 at 12:00 AM PST by [Update device plugin documentation + related pages \(#14331\)](#) ([Page History](#))

[Edit This Page](#)

# 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](#)
- [Determine whether DNS horizontal autoscaling is already enabled](#)
- [Get the name of your DNS Deployment](#)

- [Enable DNS horizontal autoscaling](#)
- [Tune DNS autoscaling parameters](#)
- [Disable DNS horizontal autoscaling](#)
- [Understanding how DNS horizontal autoscaling works](#)
- [What's next](#)

## Before you begin

- You need to have a Kubernetes cluster, and the `kubectl` command-line tool must be configured to communicate with your cluster. 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](#)An API object that manages a replicated application. in your cluster in the kube-system [namespace](#)An abstraction used by Kubernetes to support multiple virtual clusters on the same physical cluster. :

```
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](#)

```
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 [ConfigMapAn API object used to store non-confidential data in key-value pairs. Can be consumed as environment variables, command-line arguments, or config files in a volume.](#) 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.extensions/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](#).

## Feedback

Was this page helpful?

Yes No

Thanks for the feedback. If you have a specific, answerable question about how to use Kubernetes, ask it on [Stack Overflow](#). Open an issue in the GitHub repo if you want to [report a problem](#) or [suggest an improvement](#).

---

[Create an Issue](#) [Edit This Page](#)

Page last modified on January 20, 2020 at 11:45 AM PST by [Fix deployment name in docs/tasks/administer-cluster/dns-horizontal-autoscaling.md](#) (#18772) ([Page History](#))

[Edit This Page](#)

# 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](#)
- [Why change the default storage class?](#)
- [Changing the default StorageClass](#)

- [What's next](#)

## Before you begin

You need to have a Kubernetes cluster, and the `kubectl` command-line tool must be configured to communicate with your cluster. 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.

Simply 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 <your-class-name> -p '{"metadata": {"annotations":{"storageclass.kubernetes.io/is-default-class":"false"}}}'
```

where `<your-class-name>` is the name of your chosen StorageClass.

### 3. Mark a StorageClass as default:

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

```
kubectl patch storageclass <your-class-name> -p '{"metadata": {"annotations":{"storageclass.kubernetes.io/is-default-class":"true"}}}'
```

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](#).

## Feedback

Was this page helpful?

Yes No

Thanks for the feedback. If you have a specific, answerable question about how to use Kubernetes, ask it on [Stack Overflow](#). Open an issue in the GitHub repo if you want to [report a problem](#) or [suggest an improvement](#).

---

[Create an Issue](#) [Edit This Page](#)

Page last modified on November 21, 2019 at 4:39 PM PST by [Correct name of link to PersistentVolume \(#17589\)](#) ([Page History](#))

[Edit This Page](#)

# Change the Reclaim Policy of a PersistentVolume

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

- [Before you begin](#)
- [Why change reclaim policy of a PersistentVolume](#)
- [Changing the reclaim policy of a PersistentVolume](#)

- [What's next](#)

## Before you begin

You need to have a Kubernetes cluster, and the `kubectl` command-line tool must be configured to communicate with your cluster. 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` is 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	ACCESSMODES	RECLAIMPOLICY	STATUS	CAPACITY	CLAIM
STORAGECLASS		REASON	AGE		
pvc-b6efd8da-b7b5-11e6-9d58-0ed433a7dd94	Rw0	Delete	Bound	4Gi	default/claim1
manual			10s		
pvc-b95650f8-b7b5-11e6-9d58-0ed433a7dd94	Rw0	Delete	Bound	4Gi	default/claim2
manual			6s		
pvc-bb3ca71d-b7b5-11e6-9d58-0ed433a7dd94	Rw0	Delete	Bound	4Gi	default/claim3
manual			3s		

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	ACCESSMODES	RECLAIMPOLICY	STATUS	CAPACITY	CLAIM
STORAGECLASS		REASON	AGE		
pvc-b6efd8da-b7b5-11e6-9d58-0ed433a7dd94	RWO	Delete	Bound	4Gi	default/claim1
	manual		40s		
pvc-b95650f8-b7b5-11e6-9d58-0ed433a7dd94	RWO	Delete	Bound	4Gi	default/claim2
	manual		36s		
pvc-bb3ca71d-b7b5-11e6-9d58-0ed433a7dd94	RWO	Retain	Bound	4Gi	default/claim3
	manual		33s		

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](#).

## Feedback

Was this page helpful?

Yes No

Thanks for the feedback. If you have a specific, answerable question about how to use Kubernetes, ask it on [Stack Overflow](#). Open an issue in the GitHub repo if you want to [report a problem](#) or [suggest an improvement](#).

---

[Edit This Page](#)

# Cluster Management

This document describes several topics related to the lifecycle of a cluster: creating a new cluster, upgrading your cluster's master and worker nodes, performing node maintenance (e.g. kernel upgrades), and upgrading the Kubernetes API version of a running cluster.

- [Creating and configuring a Cluster](#)
- [Upgrading a cluster](#)
- [Resizing a cluster](#)
- [Maintenance on a Node](#)
- [Advanced Topics](#)

## Creating and configuring a Cluster

To install Kubernetes on a set of machines, consult one of the existing [Getting Started guides](#) depending on your environment.

## Upgrading a cluster

The current state of cluster upgrades is provider dependent, and some releases may require special care when upgrading. It is recommended that administrators consult both the [release notes](#), as well as the version specific upgrade notes prior to upgrading their clusters.

### Upgrading an Azure Kubernetes Service (AKS) cluster

Azure Kubernetes Service enables easy self-service upgrades of the control plane and nodes in your cluster. The process is currently user-initiated and is described in the [Azure AKS documentation](#).

### Upgrading Google Compute Engine clusters

Google Compute Engine Open Source (GCE-OSS) support master upgrades by deleting and recreating the master, while maintaining the same Persistent Disk (PD) to ensure that data is retained across the upgrade.

Node upgrades for GCE use a [Managed Instance Group](#), each node is sequentially destroyed and then recreated with new software. Any Pods that are running on that node need to be controlled by a Replication Controller, or manually re-created after the roll out.

Upgrades on open source Google Compute Engine (GCE) clusters are controlled by the `cluster/gce/upgrade.sh` script.

Get its usage by running `cluster/gce/upgrade.sh -h`.

For example, to upgrade just your master to a specific version (v1.0.2):

```
cluster/gce/upgrade.sh -M v1.0.2
```

Alternatively, to upgrade your entire cluster to the latest stable release:

```
cluster/gce/upgrade.sh release/stable
```

## Upgrading Google Kubernetes Engine clusters

Google Kubernetes Engine automatically updates master components (e.g. `kube-apiserver`, `kube-scheduler`) to the latest version. It also handles upgrading the operating system and other components that the master runs on.

The node upgrade process is user-initiated and is described in the [Google Kubernetes Engine documentation](#).

## Upgrading an Amazon EKS Cluster

Amazon EKS cluster's master components can be upgraded by using `eksctl`, AWS Management Console, or AWS CLI. The process is user-initiated and is described in the [Amazon EKS documentation](#).

## Upgrading an Oracle Cloud Infrastructure Container Engine for Kubernetes (OKE) cluster

Oracle creates and manages a set of master nodes in the Oracle control plane on your behalf (and associated Kubernetes infrastructure such as etcd nodes) to ensure you have a highly available managed Kubernetes control plane. You can also seamlessly upgrade these master nodes to new versions of Kubernetes with zero downtime. These actions are described in the [OKE documentation](#).

## Upgrading clusters on other platforms

Different providers, and tools, will manage upgrades differently. It is recommended that you consult their main documentation regarding upgrades.

- [kops](#)
- [kubespray](#)
- [CoreOS Tectonic](#)
- [Digital Rebar](#)
- ...

To upgrade a cluster on a platform not mentioned in the above list, check the order of component upgrade on the [Skewed versions](#) page.

# Resizing a cluster

If your cluster runs short on resources you can easily add more machines to it if your cluster is running in [Node self-registration mode](#). If you're using GCE or Google Kubernetes Engine it's done by resizing the Instance Group managing your Nodes. It can be accomplished by modifying number of instances on Compute > Compute Engine > Instance groups > your group > Edit group [Google Cloud Console page](#) or using gcloud CLI:

```
gcloud compute instance-groups managed resize kubernetes-node-pool --size=42 --zone=$ZONE
```

The Instance Group will take care of putting appropriate image on new machines and starting them, while the Kubelet will register its Node with the API server to make it available for scheduling. If you scale the instance group down, system will randomly choose Nodes to kill.

In other environments you may need to configure the machine yourself and tell the Kubelet on which machine API server is running.

## Resizing an Azure Kubernetes Service (AKS) cluster

Azure Kubernetes Service enables user-initiated resizing of the cluster from either the CLI or the Azure Portal and is described in the [Azure AKS documentation](#).

## Cluster autoscaling

If you are using GCE or Google Kubernetes Engine, you can configure your cluster so that it is automatically rescaled based on pod needs.

As described in [Compute Resource](#), users can reserve how much CPU and memory is allocated to pods. This information is used by the Kubernetes scheduler to find a place to run the pod. If there is no node that has enough free capacity (or doesn't match other pod requirements) then the pod has to wait until some pods are terminated or a new node is added.

Cluster autoscaler looks for the pods that cannot be scheduled and checks if adding a new node, similar to the other in the cluster, would help. If yes, then it resizes the cluster to accommodate the waiting pods.

Cluster autoscaler also scales down the cluster if it notices that one or more nodes are not needed anymore for an extended period of time (10min but it may change in the future).

Cluster autoscaler is configured per instance group (GCE) or node pool (Google Kubernetes Engine).

If you are using GCE then you can either enable it while creating a cluster with kube-up.sh script. To configure cluster autoscaler you have to set three environment variables:

- KUBE\_ENABLE\_CLUSTER\_AUTOSCALER - it enables cluster autoscaler if set to true.
- KUBE\_AUTOSCALER\_MIN\_NODES - minimum number of nodes in the cluster.
- KUBE\_AUTOSCALER\_MAX\_NODES - maximum number of nodes in the cluster.

Example:

```
KUBE_ENABLE_CLUSTER_AUTOSCALER=true KUBE_AUTOSCALER_MIN_NODES=3 K  
UBE_AUTOSCALER_MAX_NODES=10 NUM_NODES=5 ./cluster/kube-up.sh
```

On Google Kubernetes Engine you configure cluster autoscaler either on cluster creation or update or when creating a particular node pool (which you want to be autoscaled) by passing flags `--enable-autoscaling` `--min-nodes` and `--max-nodes` to the corresponding gcloud commands.

Examples:

```
gcloud container clusters create mytestcluster --zone=us-  
central1-b --enable-autoscaling --min-nodes=3 --max-nodes=10 --  
num-nodes=5
```

```
gcloud container clusters update mytestcluster --enable-  
autoscaling --min-nodes=1 --max-nodes=15
```

**Cluster autoscaler expects that nodes have not been manually modified (e.g. by adding labels via kubectl) as those properties would not be propagated to the new nodes within the same instance group.**

For more details about how the cluster autoscaler decides whether, when and how to scale a cluster, please refer to the [FAQ](#) documentation from the autoscaler project.

## Maintenance on a Node

If you need to reboot a node (such as for a kernel upgrade, libc upgrade, hardware repair, etc.), and the downtime is brief, then when the Kubelet restarts, it will attempt to restart the pods scheduled to it. If the reboot takes longer (the default time is 5 minutes, controlled by `--pod-eviction-timeout` on the controller-manager), then the node controller will terminate the pods that are bound to the unavailable node. If there is a corresponding replica set (or replication controller), then a new copy of the pod will be started on a different node. So, in the case where all pods are replicated, upgrades can be done without special coordination, assuming that not all nodes will go down at the same time.

If you want more control over the upgrading process, you may use the following workflow:

Use `kubectl drain` to gracefully terminate all pods on the node while marking the node as unschedulable:

```
kubectl drain $NODENAME
```

This keeps new pods from landing on the node while you are trying to get them off.

For pods with a replica set, the pod will be replaced by a new pod which will be scheduled to a new node. Additionally, if the pod is part of a service, then clients will automatically be redirected to the new pod.

For pods with no replica set, you need to bring up a new copy of the pod, and assuming it is not part of a service, redirect clients to it.

Perform maintenance work on the node.

Make the node schedulable again:

```
kubectl uncordon $NODENAME
```

If you deleted the node's VM instance and created a new one, then a new schedulable node resource will be created automatically (if you're using a cloud provider that supports node discovery; currently this is only Google Compute Engine, not including CoreOS on Google Compute Engine using `kube-register`). See [Node](#) for more details.

## Advanced Topics

### Upgrading to a different API version

When a new API version is released, you may need to upgrade a cluster to support the new API version (e.g. switching from `v1` to `v2` when `v2` is launched).

This is an infrequent event, but it requires careful management. There is a sequence of steps to upgrade to a new API version.

1. Turn on the new API version.
2. Upgrade the cluster's storage to use the new version.
3. Upgrade all config files. Identify users of the old API version endpoints.
4. Update existing objects in the storage to new version by running `cluster/update-storage-objects.sh`.
5. Turn off the old API version.

### Turn on or off an API version for your cluster

Specific API versions can be turned on or off by passing `--runtime-config=api/<version>` flag while bringing up the API server. For example:

to turn off v1 API, pass `--runtime-config=api/v1=false`. `runtime-config` also supports 2 special keys: `api/all` and `api/legacy` to control all and legacy APIs respectively. For example, for turning off all API versions except v1, pass `--runtime-config=api/all=false,api/v1=true`. For the purposes of these flags, *legacy* APIs are those APIs which have been explicitly deprecated (e.g. `v1beta3`).

## Switching your cluster's storage API version

The objects that are stored to disk 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.

## Switching your config files to a new API version

You can use `kubectl convert` command to convert config files between different API versions.

```
kubectl convert -f pod.yaml --output-version v1
```

For more options, please refer to the usage of [kubectl convert](#) command.

## Feedback

Was this page helpful?

Yes No

Thanks for the feedback. If you have a specific, answerable question about how to use Kubernetes, ask it on [Stack Overflow](#). Open an issue in the GitHub repo if you want to [report a problem](#) or [suggest an improvement](#).

---

[Create an Issue](#) [Edit This Page](#)

Page last modified on November 26, 2019 at 4:51 AM PST by [add EKS cluster upgrading section \(#17800\)](#) ([Page History](#))

[Edit This Page](#)

## Configure Multiple Schedulers

Kubernetes ships with a default scheduler that is described [here](#). If the default scheduler does not suit your needs you can implement your own scheduler. Not just that, you can even run multiple schedulers simultaneously alongside the default scheduler and instruct Kubernetes what scheduler to use for each of your pods. Let's learn how to run multiple schedulers in Kubernetes with an example.

A detailed description of how to implement a scheduler is outside the scope of this document. Please refer to the kube-scheduler implementation in [pkg/scheduler](#) in the Kubernetes source directory for a canonical example.

- [Before you begin](#)
- [Package the scheduler](#)
- [Define a Kubernetes Deployment for the scheduler](#)
- [Run the second scheduler in the cluster](#)
- [Specify schedulers for pods](#)

## Before you begin

You need to have a Kubernetes cluster, and the kubectl command-line tool must be configured to communicate with your cluster. 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`.

## Package the scheduler

Package your scheduler binary into a container image. For the purposes of this example, let's just use the default scheduler (kube-scheduler) as our second scheduler as well. Clone the [Kubernetes source code from GitHub](#) and build the source.

```
git clone https://github.com/kubernetes/kubernetes.git  
cd kubernetes  
make
```

Create a container image containing the kube-scheduler binary. Here is the Dockerfile to build the image:

```
FROM busybox  
ADD ./_output/local/bin/linux/amd64/kube-scheduler /usr/local/  
bin/kube-scheduler
```

Save the file as Dockerfile, build the image and push it to a registry. This example pushes the image to [Google Container Registry \(GCR\)](#). For more details, please read the GCR [documentation](#).

```
docker build -t gcr.io/my-gcp-project/my-kube-scheduler:1.0 .  
gcloud docker -- push gcr.io/my-gcp-project/my-kube-scheduler:1.0
```

## Define a Kubernetes Deployment for the scheduler

Now that we have our scheduler in a container image, we can just create a pod config for it and run it in our Kubernetes cluster. But instead of creating a pod directly in the cluster, let's use a [Deployment](#) for this example. A [Deployment](#) manages a [Replica Set](#) which in turn manages the pods, thereby making the scheduler resilient to failures. Here is the deployment config. Save it as `my-scheduler.yaml`:

```
apiVersion: v1
kind: ServiceAccount
metadata:
  name: my-scheduler
  namespace: kube-system
---
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRoleBinding
metadata:
  name: my-scheduler-as-kube-scheduler
subjects:
- kind: ServiceAccount
  name: my-scheduler
  namespace: kube-system
roleRef:
  kind: ClusterRole
  name: system:kube-scheduler
  apiGroup: rbac.authorization.k8s.io
---
apiVersion: apps/v1
kind: Deployment
metadata:
  labels:
    component: scheduler
    tier: control-plane
  name: my-scheduler
  namespace: kube-system
spec:
  selector:
    matchLabels:
      component: scheduler
      tier: control-plane
  replicas: 1
  template:
    metadata:
      labels:
        component: scheduler
        tier: control-plane
        version: second
  spec:
    serviceAccountName: my-scheduler
    containers:
      - command:
          - /usr/local/bin/kube-scheduler
          - --address=0.0.0.0
          - --leader-elect=false
          - --scheduler-name=my-scheduler
        image: gcr.io/my-gcp-project/my-kube-scheduler:1.0
        livenessProbe:
          httpGet:
            path: /healthz
            port: 10251
        initialDelaySeconds: 15
        name: kube-second-scheduler
        readinessProbe:
          httpGet:
```

An important thing to note here is that the name of the scheduler specified as an argument to the scheduler command in the container spec should be unique. This is the name that is matched against the value of the optional `spec.schedulerName` on pods, to determine whether this scheduler is responsible for scheduling a particular pod.

Note also that we created a dedicated service account `my-scheduler` and bind the cluster role `system:kube-scheduler` to it so that it can acquire the same privileges as `kube-scheduler`.

Please see the [kube-scheduler documentation](#) for detailed description of other command line arguments.

## Run the second scheduler in the cluster

In order to run your scheduler in a Kubernetes cluster, just create the deployment specified in the config above in a Kubernetes cluster:

```
kubectl create -f my-scheduler.yaml
```

Verify that the scheduler pod is running:

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

NAME	READY	
STATUS	RESTARTS	AGE
....		
my-scheduler-lnf4s-4744f		1/1
Running	0	2m
...		

You should see a "Running" `my-scheduler` pod, in addition to the default `kube-scheduler` pod in this list.

To run multiple-scheduler with leader election enabled, you must do the following:

First, update the following fields in your YAML file:

- `--leader-elect=true`
- `--lock-object-namespace=lock-object-namespace`
- `--lock-object-name=lock-object-name`

If RBAC is enabled on your cluster, you must update the `system:kube-scheduler` cluster role. Add your scheduler name to the `resourceNames` of the rule applied for `endpoints` resources, as in the following example:

```
kubectl edit clusterrole system:kube-scheduler
```

```
- apiVersion: rbac.authorization.k8s.io/v1
  kind: ClusterRole
  metadata:
    annotations:
```

```
rbac.authorization.kubernetes.io/autoupdate: "true"
labels:
  kubernetes.io/bootstrapping: rbac-defaults
  name: system:kube-scheduler
rules:
- apiGroups:
  - ""
    resourceNames:
    - kube-scheduler
    - my-scheduler
    resources:
    - endpoints
    verbs:
    - delete
    - get
    - patch
    - update
```

## Specify schedulers for pods

Now that our second scheduler is running, let's create some pods, and direct them to be scheduled by either the default scheduler or the one we just deployed. In order to schedule a given pod using a specific scheduler, we specify the name of the scheduler in that pod spec. Let's look at three examples.

- Pod spec without any scheduler name

### [admin/sched/pod1.yaml](#)

```
apiVersion: v1
kind: Pod
metadata:
  name: no-annotation
  labels:
    name: multischeduler-example
spec:
  containers:
  - name: pod-with-no-annotation-container
    image: k8s.gcr.io/pause:2.0
```

When no scheduler name is supplied, the pod is automatically scheduled using the default-scheduler.

Save this file as `pod1.yaml` and submit it to the Kubernetes cluster.

```
kubectl create -f pod1.yaml
```

- Pod spec with `default-scheduler`

### [admin/sched/pod2.yaml](#)

```
apiVersion: v1
kind: Pod
metadata:
  name: annotation-default-scheduler
  labels:
    name: multischeduler-example
spec:
  schedulerName: default-scheduler
  containers:
  - name: pod-with-default-annotation-container
    image: k8s.gcr.io/pause:2.0
```

A scheduler is specified by supplying the scheduler name as a value to `spec.schedulerName`. In this case, we supply the name of the default scheduler which is `default-scheduler`.

Save this file as `pod2.yaml` and submit it to the Kubernetes cluster.

```
kubectl create -f pod2.yaml
```

- Pod spec with `my-scheduler`

### [admin/sched/pod3.yaml](#)

```
apiVersion: v1
kind: Pod
metadata:
  name: annotation-second-scheduler
  labels:
    name: multischeduler-example
spec:
  schedulerName: my-scheduler
  containers:
  - name: pod-with-second-annotation-container
    image: k8s.gcr.io/pause:2.0
```

In this case, we specify that this pod should be scheduled using the scheduler that we deployed - `my-scheduler`. Note that the value of `spec.schedulerName` should match the name supplied to the `scheduler` command as an argument in the deployment config for the scheduler.

Save this file as `pod3.yaml` and submit it to the Kubernetes cluster.

```
kubectl create -f pod3.yaml
```

Verify that all three pods are running.

```
kubectl get pods
```

## Verifying that the pods were scheduled using the desired schedulers

In order to make it easier to work through these examples, we did not verify that the pods were actually scheduled using the desired schedulers. We can verify that by changing the order of pod and deployment config submissions above. If we submit all the pod configs to a Kubernetes cluster before submitting the scheduler deployment config, we see that the pod `annotation-second-scheduler` remains in "Pending" state forever while the other two pods get scheduled. Once we submit the scheduler deployment config and our new scheduler starts running, the `annotation-second-scheduler` pod gets scheduled as well.

Alternatively, one could just look at the "Scheduled" entries in the event logs to verify that the pods were scheduled by the desired schedulers.

```
kubectl get events
```

## Feedback

Was this page helpful?

Yes No

Thanks for the feedback. If you have a specific, answerable question about how to use Kubernetes, ask it on [Stack Overflow](#). Open an issue in the GitHub repo if you want to [report a problem](#) or [suggest an improvement](#).

---

[Create an Issue](#) [Edit This Page](#)

Page last modified on September 27, 2019 at 5:53 PM PST by [wrong path of kube-scheduler \(#16051\)](#) ([Page History](#))

[Edit This Page](#)

# Configure Out of Resource Handling

This page explains how to configure out of resource handling with kubelet.

The kubelet needs to preserve node stability when available compute resources are low. This is especially important when dealing with

incompressible compute resources, such as memory or disk space. If such resources are exhausted, nodes become unstable.

- [Eviction Policy](#)
- [Node OOM Behavior](#)
- [Best Practices](#)
- [Deprecation of existing feature flags to reclaim disk](#)
- [Known issues](#)

## Eviction Policy

The kubelet can proactively monitor for and prevent total starvation of a compute resource. In those cases, the kubelet can reclaim the starved resource by proactively failing one or more Pods. When the kubelet fails a Pod, it terminates all of its containers and transitions its PodPhase to Failed. If the evicted Pod is managed by a Deployment, the Deployment will create another Pod to be scheduled by Kubernetes.

### Eviction Signals

The kubelet supports eviction decisions based on the signals described in the following table. The value of each signal is described in the Description column, which is based on the kubelet summary API.

Eviction Signal	Description
memory.available	<code>memory.available := node.status.capacity[memory] - node.stats.memory.workingSet</code>
nodefs.available	<code>nodefs.available := node.stats.fs.available</code>
nodefs.inodesFree	<code>nodefs.inodesFree := node.stats.fs.inodesFree</code>
imagefs.available	<code>imagefs.available := node.stats.runtime.imagefs.available</code>
imagefs.inodesFree	<code>imagefs.inodesFree := node.stats.runtime.imagefs.inodesFree</code>

Each of the above signals supports either a literal or percentage based value. The percentage based value is calculated relative to the total capacity associated with each signal.

The value for `memory.available` is derived from the cgroupfs instead of tools like `free -m`. This is important because `free -m` does not work in a container, and if users use the [node allocatable](#) feature, out of resource decisions are made local to the end user Pod part of the cgroup hierarchy as well as the root node. This [script](#) reproduces the same set of steps that the kubelet performs to calculate `memory.available`. The kubelet excludes `inactive_file` (i.e. # of bytes of file-backed memory on inactive LRU list) from its calculation as it assumes that memory is reclaimable under pressure.

kubelet supports only two filesystem partitions.

1. The `nodefs` filesystem that kubelet uses for volumes, daemon logs, etc.
2. The `imagefs` filesystem that container runtimes uses for storing images and container writable layers.

`imagefs` is optional. `kubelet` auto-discovers these filesystems using `cAdvisor`. `kubelet` does not care about any other filesystems. Any other types of configurations are not currently supported by the `kubelet`. For example, it is *not OK* to store volumes and logs in a dedicated filesystem.

In future releases, the `kubelet` will deprecate the existing [garbage collection](#) support in favor of eviction in response to disk pressure.

## Eviction Thresholds

The `kubelet` supports the ability to specify eviction thresholds that trigger the `kubelet` to reclaim resources.

Each threshold has the following form:

`[eviction-signal][operator][quantity]`

where:

- `eviction-signal` is an eviction signal token as defined in the previous table.
- `operator` is the desired relational operator, such as `<` (less than).
- `quantity` is the eviction threshold quantity, such as `1Gi`. These tokens must match the quantity representation used by Kubernetes. An eviction threshold can also be expressed as a percentage using the `%` token.

For example, if a node has `10Gi` of total memory and you want trigger eviction if the available memory falls below `1Gi`, you can define the eviction threshold as either `memory.available<10%` or `memory.available<1Gi`. You cannot use both.

## Soft Eviction Thresholds

A soft eviction threshold pairs an eviction threshold with a required administrator-specified grace period. No action is taken by the `kubelet` to reclaim resources associated with the eviction signal until that grace period has been exceeded. If no grace period is provided, the `kubelet` returns an error on startup.

In addition, if a soft eviction threshold has been met, an operator can specify a maximum allowed Pod termination grace period to use when evicting pods from the node. If specified, the `kubelet` uses the lesser value among the `pod.Spec.TerminationGracePeriodSeconds` and the max allowed grace period. If not specified, the `kubelet` kills Pods immediately with no graceful termination.

To configure soft eviction thresholds, the following flags are supported:

- `eviction-soft` describes a set of eviction thresholds (e.g. `memory.available<1.5Gi`) that if met over a corresponding grace period would trigger a Pod eviction.

- `eviction-soft-grace-period` describes a set of eviction grace periods (e.g. `memory.available=1m30s`) that correspond to how long a soft eviction threshold must hold before triggering a Pod eviction.
- `eviction-max-pod-grace-period` describes the maximum allowed grace period (in seconds) to use when terminating pods in response to a soft eviction threshold being met.

## Hard Eviction Thresholds

A hard eviction threshold has no grace period, and if observed, the kubelet will take immediate action to reclaim the associated starved resource. If a hard eviction threshold is met, the kubelet kills the Pod immediately with no graceful termination.

To configure hard eviction thresholds, the following flag is supported:

- `eviction-hard` describes a set of eviction thresholds (e.g. `memory.available<1Gi`) that if met would trigger a Pod eviction.

The kubelet has the following default hard eviction threshold:

- `memory.available<100Mi`
- `nodefs.available<10%`
- `nodefs.inodesFree<5%`
- `imagefs.available<15%`

## Eviction Monitoring Interval

The kubelet evaluates eviction thresholds per its configured housekeeping interval.

- `housekeeping-interval` is the interval between container housekeepings which defaults to `10s`.

## Node Conditions

The kubelet maps one or more eviction signals to a corresponding node condition.

If a hard eviction threshold has been met, or a soft eviction threshold has been met independent of its associated grace period, the kubelet reports a condition that reflects the node is under pressure.

The following node conditions are defined that correspond to the specified eviction signal.

Node Condition	Eviction Signal	Description
MemoryPressure	memory.available	Available memory on the node has satisfied the eviction threshold.
DiskPressure	nodefs.available, nodefs.inodesFree, imagefs.available, or imagefs.inodesFree	Available disk space and inodes on either the node root filesystem or image filesystem has satisfied an eviction threshold.

The kubelet continues to report node status updates at the frequency specified by `--node-status-update-frequency` which defaults to 10s.

## Oscillation of node conditions

If a node is oscillating above and below a soft eviction threshold, but not exceeding its associated grace period, it would cause the corresponding node condition to constantly oscillate between true and false, and could cause poor scheduling decisions as a consequence.

To protect against this oscillation, the following flag is defined to control how long the kubelet must wait before transitioning out of a pressure condition.

- `eviction-pressure-transition-period` is the duration for which the kubelet has to wait before transitioning out of an eviction pressure condition.

The kubelet would ensure that it has not observed an eviction threshold being met for the specified pressure condition for the period specified before toggling the condition back to `false`.

## Reclaiming node level resources

If an eviction threshold has been met and the grace period has passed, the kubelet initiates the process of reclaiming the pressured resource until it has observed the signal has gone below its defined threshold.

The kubelet attempts to reclaim node level resources prior to evicting end-user Pods. If disk pressure is observed, the kubelet reclaims node level

resources differently if the machine has a dedicated `imagefs` configured for the container runtime.

## With `imagefs`

If `nodefs` filesystem has met eviction thresholds, kubelet frees up disk space by deleting the dead Pods and their containers.

If `imagefs` filesystem has met eviction thresholds, kubelet frees up disk space by deleting all unused images.

## Without `imagefs`

If `nodefs` filesystem has met eviction thresholds, kubelet frees up disk space in the following order:

1. Delete dead Pods and their containers
2. Delete all unused images

## Evicting end-user Pods

If the kubelet is unable to reclaim sufficient resource on the node, kubelet begins evicting Pods.

The kubelet ranks Pods for eviction first by whether or not their usage of the starved resource exceeds requests, then by [Priority](#), and then by the consumption of the starved compute resource relative to the Pods' scheduling requests.

As a result, kubelet ranks and evicts Pods in the following order:

- BestEffort or Burstable Pods whose usage of a starved resource exceeds its request. Such pods are ranked by Priority, and then usage above request.
- Guaranteed pods and Burstable pods whose usage is beneath requests are evicted last. Guaranteed Pods are guaranteed only when requests and limits are specified for all the containers and they are equal. Such pods are guaranteed to never be evicted because of another Pod's resource consumption. If a system daemon (such as kubelet, docker, and journald) is consuming more resources than were reserved via system-reserved or kube-reserved allocations, and the node only has Guaranteed or Burstable Pods using less than requests remaining, then the node must choose to evict such a Pod in order to preserve node stability and to limit the impact of the unexpected consumption to other Pods. In this case, it will choose to evict pods of Lowest Priority first.

If necessary, kubelet evicts Pods one at a time to reclaim disk when `DiskPressure` is encountered. If the kubelet is responding to inode starvation, it reclaims inodes by evicting Pods with the lowest quality of service first. If the kubelet is responding to lack of available disk, it ranks Pods within a quality of service that consumes the largest amount of disk and kills those first.

## With imagefs

If `nodefs` is triggering evictions, `kubelet` sorts Pods based on the usage on `nodefs` - local volumes + logs of all its containers.

If `imagefs` is triggering evictions, `kubelet` sorts Pods based on the writable layer usage of all its containers.

## Without imagefs

If `nodefs` is triggering evictions, `kubelet` sorts Pods based on their total disk usage - local volumes + logs & writable layer of all its containers.

## Minimum eviction reclaim

In certain scenarios, eviction of Pods could result in reclamation of small amount of resources. This can result in `kubelet` hitting eviction thresholds in repeated successions. In addition to that, eviction of resources like `disk`, is time consuming.

To mitigate these issues, `kubelet` can have a per-resource `minimum-reclaim`. Whenever `kubelet` observes resource pressure, `kubelet` attempts to reclaim at least `minimum-reclaim` amount of resource below the configured eviction threshold.

For example, with the following configuration:

```
--eviction-
hard=memory.available<500Mi,nodefs.available<1Gi,imagefs.available<100Gi
--eviction-minimum-
reclaim="memory.available=0Mi,nodefs.available=500Mi,imagefs.available=2Gi"
```

If an eviction threshold is triggered for `memory.available`, the `kubelet` works to ensure that `memory.available` is at least `500Mi`. For `nodefs.available`, the `kubelet` works to ensure that `nodefs.available` is at least `1.5Gi`, and for `imagefs.available` it works to ensure that `imagefs.available` is at least `102Gi` before no longer reporting pressure on their associated resources.

The default `eviction-minimum-reclaim` is `0` for all resources.

## Scheduler

The node reports a condition when a compute resource is under pressure. The scheduler views that condition as a signal to dissuade placing additional pods on the node.

Node Condition	Scheduler Behavior
MemoryPressure	No new BestEffort Pods are scheduled to the node.

<b>Node Condition</b>	<b>Scheduler Behavior</b>
DiskPressure	No new Pods are scheduled to the node.

## Node OOM Behavior

If the node experiences a system OOM (out of memory) event prior to the kubelet being able to reclaim memory, the node depends on the [oom killer](#) to respond.

The kubelet sets a `oom_score_adj` value for each container based on the quality of service for the Pod.

<b>Quality of Service</b>	<b>oom_score_adj</b>
Guaranteed	-998
BestEffort	1000
Burstable	<code>min(max(2, 1000 - (1000 * memoryRequestBytes) / machineMemoryCapacityBytes), 999)</code>

If the kubelet is unable to reclaim memory prior to a node experiencing system OOM, the `oom killer` calculates an `oom_score` based on the percentage of memory it's using on the node, and then add the `oom_score_adj` to get an effective `oom_score` for the container, and then kills the container with the highest score.

The intended behavior should be that containers with the lowest quality of service that are consuming the largest amount of memory relative to the scheduling request should be killed first in order to reclaim memory.

Unlike Pod eviction, if a Pod container is OOM killed, it may be restarted by the kubelet based on its `RestartPolicy`.

## Best Practices

The following sections describe best practices for out of resource handling.

### Schedulable resources and eviction policies

Consider the following scenario:

- Node memory capacity: 10Gi
- Operator wants to reserve 10% of memory capacity for system daemons (kernel, kubelet, etc.)
- Operator wants to evict Pods at 95% memory utilization to reduce incidence of system OOM.

To facilitate this scenario, the kubelet would be launched as follows:

```
--eviction-hard=memory.available<500Mi
--system-reserved=memory=1.5Gi
```

Implicit in this configuration is the understanding that "System reserved" should include the amount of memory covered by the eviction threshold.

To reach that capacity, either some Pod is using more than its request, or the system is using more than  $1.5Gi - 500Mi = 1Gi$ .

This configuration ensures that the scheduler does not place Pods on a node that immediately induce memory pressure and trigger eviction assuming those Pods use less than their configured request.

## DaemonSet

As Priority is a key factor in the eviction strategy, if you do not want pods belonging to a DaemonSet to be evicted, specify a sufficiently high priorityClass in the pod spec template. If you want pods belonging to a DaemonSet to run only if there are sufficient resources, specify a lower or default priorityClass.

## Deprecation of existing feature flags to reclaim disk

kubelet has been freeing up disk space on demand to keep the node stable.

As disk based eviction matures, the following kubelet flags are marked for deprecation in favor of the simpler configuration supported around eviction.

Existing Flag	New Flag
--image-gc-high-threshold	--eviction-hard or eviction-soft
--image-gc-low-threshold	--eviction-minimum-reclaim
--maximum-dead-containers	deprecated
--maximum-dead-containers-per-container	deprecated
--minimum-container-ttl-duration	deprecated
--low-diskspace-threshold-mb	--eviction-hard or eviction-soft
--outofdisk-transition-frequency	--eviction-pressure-transition-period

## Known issues

The following sections describe known issues related to out of resource handling.

### kubelet may not observe memory pressure right away

The kubelet currently polls cAdvisor to collect memory usage stats at a regular interval. If memory usage increases within that window rapidly, the kubelet may not observe MemoryPressure fast enough, and the OOMKiller will still be invoked. We intend to integrate with the memcg notification API in

a future release to reduce this latency, and instead have the kernel tell us when a threshold has been crossed immediately.

If you are not trying to achieve extreme utilization, but a sensible measure of overcommit, a viable workaround for this issue is to set eviction thresholds at approximately 75% capacity. This increases the ability of this feature to prevent system OOMs, and promote eviction of workloads so cluster state can rebalance.

## **kubelet may evict more Pods than needed**

The Pod eviction may evict more Pods than needed due to stats collection timing gap. This can be mitigated by adding the ability to get root container stats on an on-demand basis (<https://github.com/google/cadvisor/issues/1247>) in the future.

## **Feedback**

Was this page helpful?

Yes No

Thanks for the feedback. If you have a specific, answerable question about how to use Kubernetes, ask it on [Stack Overflow](#). Open an issue in the GitHub repo if you want to [report a problem](#) or [suggest an improvement](#).

---

[Create an Issue](#) [Edit This Page](#)

Page last modified on November 25, 2019 at 11:41 PM PST by [Minor heading update according to the guideline \(#17747\)](#) ([Page History](#))

[Edit This Page](#)

# 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](#)
- [Create a namespace](#)
- [Create a ResourceQuota](#)

- [Create a PersistentVolumeClaim](#)
- [Attempt to create a second PersistentVolumeClaim](#)
- [Notes](#)
- [Clean up](#)
- [What's next](#)

## Before you begin

You need to have a Kubernetes cluster, and the kubectl command-line tool must be configured to communicate with your cluster. 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

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-object-example
```

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.

```
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  
  
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-object-example
```

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](https://k8s.io/examples/admin/resource/quota-objects-pvc-2.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=quota-object-example
```

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](#)

# Feedback

Was this page helpful?

Yes No

Thanks for the feedback. If you have a specific, answerable question about how to use Kubernetes, ask it on [Stack Overflow](#). Open an issue in the GitHub repo if you want to [report a problem](#) or [suggest an improvement](#).

---

[Create an Issue](#) [Edit This Page](#)

Page last modified on March 25, 2019 at 5:06 PM PST by [Official 1.14](#)  
[Release Docs \(#13174\)](#) ([Page History](#))

[Edit This Page](#)

# Control CPU Management Policies on the Node

**FEATURE STATE:** Kubernetes v1.12 [beta](#)

This feature is currently in a *beta* state, meaning:

[Edit This Page](#)

# Control Topology Management Policies on a node

**FEATURE STATE:** Kubernetes v1.17 [alpha](#)

This feature is currently in a *alpha* state, meaning:

[Edit This Page](#)

## Customizing DNS Service

This page explains how to configure your DNS Pod and customize the DNS resolution process. In Kubernetes version 1.11 and later, CoreDNS is at GA and is installed by default with kubeadm. See [CoreDNS ConfigMap options](#) and [Using CoreDNS for Service Discovery](#).

- [Before you begin](#)
- [Introduction](#)
- [CoreDNS](#)
- [Kube-dns](#)
- [CoreDNS configuration equivalent to kube-dns](#)
- [Migration to CoreDNS](#)
- [What's next](#)

## Before you begin

- You need to have a Kubernetes cluster, and the `kubectl` command-line tool must be configured to communicate with your cluster. 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 1.6 or later. To work with CoreDNS, version 1.9 or later.
- The appropriate add-on: `kube-dns` or `CoreDNS`. To install with `kubeadm`, see [the kubeadm reference documentation](#).

## 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`. However, `kube-dns` may still be installed by default with

certain Kubernetes installer tools. Refer to the documentation provided by your installer to know which DNS server is installed by default.

The CoreDNS Deployment is exposed as a Kubernetes Service with a static IP. Both the CoreDNS and kube-dns Service are named `kube-dns` in the `meta.name` field. This is done so that there is greater interoperability with workloads that relied on the legacy `kube-dns` Service name to resolve addresses internal to the cluster. It abstracts away the implementation detail of which DNS provider is running behind that common endpoint. The `kubelet` passes DNS 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 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. A cluster administrator can modify the ConfigMap for the CoreDNS Corefile to change how service discovery works.

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 [here](#). `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.
- [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
    }
```

In Kubernetes version 1.10 and later, kubeadm supports automatic translation of the CoreDNS ConfigMap from the kube-dns 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.*

## Kube-dns

Kube-dns is now available as an optional DNS server since CoreDNS is now the default. The running DNS Pod holds 3 containers:

- "kubedns": watches the Kubernetes master for changes in Services and Endpoints, and maintains in-memory lookup structures to serve DNS requests.
- "dnsmasq": adds DNS caching to improve performance.
- "sidecar": provides a single health check endpoint to perform healthchecks for dnsmasq and kubedns.

## Configure stub-domain and upstream DNS servers

Cluster administrators can specify custom stub domains and upstream nameservers by providing a ConfigMap for kube-dns (kube-system:kube-dns).

For example, the following ConfigMap sets up a DNS configuration with a single stub domain and two upstream nameservers:

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: kube-dns
  namespace: kube-system
data:
  stubDomains: /
    {"acme.local": ["1.2.3.4"]}
  upstreamNameservers: /
    ["8.8.8.8", "8.8.4.4"]
```

DNS requests with the ".acme.local" suffix are forwarded to a DNS listening at 1.2.3.4. Google Public DNS serves the upstream queries.

The table below describes how queries with certain domain names map to their destination DNS servers:

Domain name	Server answering the query
kubernetes.default.svc.cluster.local	kube-dns
foo.acme.local	custom DNS (1.2.3.4)

Domain name	Server answering the query
widget.com	upstream DNS (one of 8.8.8.8, 8.8.4.4)

See [ConfigMap options](#) for details about the configuration option format.

## Effects on Pods

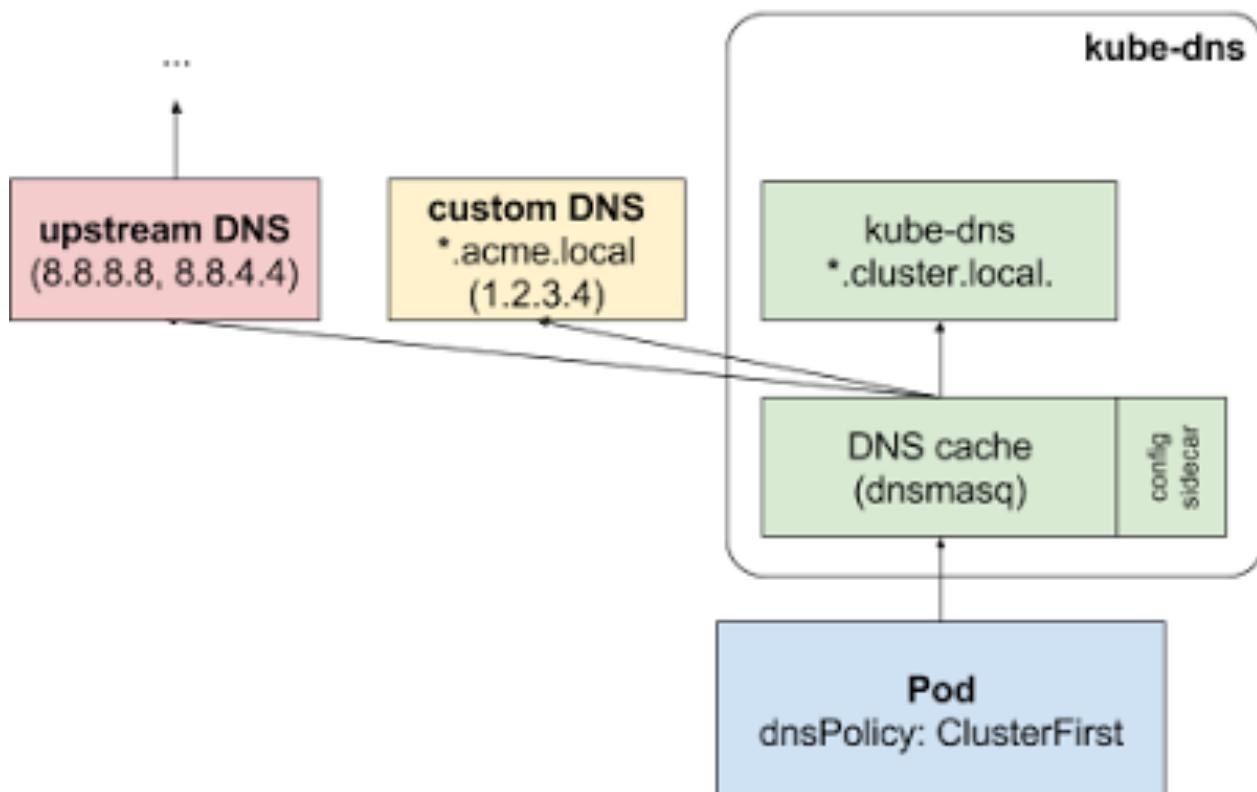
Custom upstream nameservers and stub domains do not affect Pods with a `dnsPolicy` set to "Default" or "None".

If a Pod's `dnsPolicy` is set to "ClusterFirst", its name resolution is handled differently, depending on whether stub-domain and upstream DNS servers are configured.

**Without custom configurations:** Any query that does not match the configured cluster domain suffix, such as "www.kubernetes.io", is forwarded to the upstream nameserver inherited from the node.

**With custom configurations:** If stub domains and upstream DNS servers are configured, DNS queries are routed according to the following flow:

1. The query is first sent to the DNS caching layer in kube-dns.
2. From the caching layer, the suffix of the request is examined and then forwarded to the appropriate DNS, based on the following cases:
  - *Names with the cluster suffix*, for example ".cluster.local": The request is sent to kube-dns.
  - *Names with the stub domain suffix*, for example ".acme.local": The request is sent to the configured custom DNS resolver, listening for example at 1.2.3.4.
  - *Names without a matching suffix*, for example "widget.com": The request is forwarded to the upstream DNS, for example Google public DNS servers at 8.8.8.8 and 8.8.4.4.



## ConfigMap options

Options for the kube-dns `kube-system:kube-dns` ConfigMap:

Field	Format	Description
<code>stubDomains</code> (optional)	A JSON map using a DNS suffix key such as "acme.local", and a value consisting of a JSON array of DNS IPs.	The target nameserver can itself be a Kubernetes Service. For instance, you can run your own copy of dnsmasq to export custom DNS names into the ClusterDNS namespace.
<code>upstreamNameservers</code> (optional)	A JSON array of DNS IPs.	If specified, the values replace the nameservers taken by default from the node's <code>/etc/resolv.conf</code> . Limits: a maximum of three upstream nameservers can be specified.

## Examples

### Example: Stub domain

In this example, the user has a Consul DNS service discovery system they want to integrate with kube-dns. The consul domain server is located at 10.150.0.1, and all consul names have the suffix `.consul.local`. To

configure Kubernetes, the cluster administrator creates the following ConfigMap:

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: kube-dns
  namespace: kube-system
data:
  stubDomains: /
    {"consul.local": ["10.150.0.1"]}
```

Note that the cluster administrator does not want to override the node's upstream nameservers, so they did not specify the optional `upstreamNameservers` field.

#### **Example: Upstream nameserver**

In this example the cluster administrator wants to explicitly force all non-cluster DNS lookups to go through their own nameserver at 172.16.0.1. In this case, they create a ConfigMap with the `upstreamNameservers` field specifying the desired nameserver:

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: kube-dns
  namespace: kube-system
data:
  upstreamNameservers: /
    ["172.16.0.1"]
```

## **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. Similarly, the `Federations` plugin in kube-dns translates to the `federation` plugin in CoreDNS.

#### **Example**

This example ConfigMap for kubedns specifies federations, stubdomains and upstreamnameservers:

```
apiVersion: v1
data:
  federations: /
    {"foo" : "foo.feddomain.com"}
  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 federations:

```
federation cluster.local {  
    foo foo.feddomain.com  
}
```

- 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](#) is available to help users adapt CoreDNS in place of kube-dns. A cluster administrator can also migrate using [the deploy script](#).

## What's next

- [Debugging DNS Resolution](#).

## Feedback

Was this page helpful?

Yes No

Thanks for the feedback. If you have a specific, answerable question about how to use Kubernetes, ask it on [Stack Overflow](#). Open an issue in the GitHub repo if you want to [report a problem](#) or [suggest an improvement](#).

---

[Create an Issue](#) [Edit This Page](#)

Page last modified on December 16, 2019 at 10:27 AM PST by [Updated default Corefile that is installed with the current version of CoreDNS.](#) ([#18158](#)) ([Page History](#))

[Edit This Page](#)

# Debugging DNS Resolution

This page provides hints on diagnosing DNS problems.

- [Before you begin](#)
- [Known issues](#)
- [Kubernetes Federation \(Multiple Zone support\)](#)
- [References](#)
- [What's next](#)

# Before you begin

- You need to have a Kubernetes cluster, and the kubectl command-line tool must be configured to communicate with your cluster. 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 1.6 and above.
- The cluster must be configured to use the `coredns` (or `kube-dns`) addons.

## Create a simple Pod to use as a test environment

Create a file named `dnsutils.yaml` with the following contents:

```
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
```

Then create a pod using this file and verify its status:

```
kubectl apply -f https://k8s.io/examples/admin/dns/dnsutils.yaml
pod/dnsutils created
```

```
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 -ti 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 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.internal
nameserver 10.0.0.10
options ndots:5
```

Errors such as the following indicate a problem with the coredns/kube-dns add-on or associated Services:

```
kubectl exec -ti dnsutils -- nslookup kubernetes.default
Server: 10.0.0.10
Address 1: 10.0.0.10
```

```
nslookup: can't resolve 'kubernetes.default'
```

or

```
kubectl exec -ti 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.

For CoreDNS:

```
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-mvmmmt  1/1     Running   0          1h
...
```

Or for kube-dns:

NAME	READY	STATUS	RESTARTS	AGE
...				
kube-dns-v19-ezoly	3/3	Running	0	1h
...				

If you see that no 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 `kubectl logs` command to see logs for the DNS containers.

For CoreDNS:

```
for p in $(kubectl get pods --namespace=kube-system -l k8s-app=kube-dns -o name); do kubectl logs --namespace=kube-system $p; done
```

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 = 24e6c59e83ce706f07bcc82c31blealc
```

For kube-dns, there are 3 sets of logs:

```
kubectl logs --namespace=kube-system $(kubectl get pods --namespace=kube-system -l k8s-app=kube-dns -o name | head -1) -c kubedns
```

```
kubectl logs --namespace=kube-system $(kubectl get pods --namespace=kube-system -l k8s-app=kube-dns -o name | head -1) -c dnsmasq
```

```
kubectl logs --namespace=kube-system $(kubectl get pods --namespace=kube-system -l k8s-app=kube-dns -o name | head -1) -c sidecar
```

See if there are any suspicious error messages in the logs. In kube-dns, a 'W', 'E' or 'F' at the beginning of a line represents a Warning, Error or Failure. Please search for entries that have these as the logging level and use [kubernetes issues](#) to report unexpected errors.

## 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 that the service name will be "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 ep 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 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 â€¢

```
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
proxy . /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 = 162475cdf272d8aa601e6fe67a6ad42f
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. udp 54 false 512" NOERROR
qr,aa,rd,ra 106 0.000066649s

```

## 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` ( $\geq 1.11$ ) 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 glic-2.17-222 ([the new versions update see this issue](#)), the 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 nameservers or uses more than 3 searches while your glibc versions in the affected list, some of those settings will be lost. For the workaround of 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. For fixing the DNS search records limit, consider upgrading your linux distribution or glibc version.

If you are using Alpine version 3.3 or earlier as your base image, DNS may not work properly owing to a known issue with Alpine. Check [here](#) for more information.

## Kubernetes Federation (Multiple Zone support)

Release 1.3 introduced Cluster Federation support for multi-site Kubernetes installations. This required some minor (backward-compatible) changes to the way the Kubernetes cluster DNS server processes DNS queries, to facilitate the lookup of federated services (which span multiple Kubernetes clusters). See the [Cluster Federation Administrators' Guide](#) for more details on Cluster Federation and multi-site support.

## References

- [DNS for Services and Pods](#)
- [Docs for the kube-dns DNS cluster addon](#)

## What's next

- [Autoscaling the DNS Service in a Cluster.](#)

## Feedback

Was this page helpful?

Yes No

Thanks for the feedback. If you have a specific, answerable question about how to use Kubernetes, ask it on [Stack Overflow](#). Open an issue in the GitHub repo if you want to [report a problem](#) or [suggest an improvement](#).

---

[Create an Issue](#) [Edit This Page](#)

Page last modified on February 08, 2020 at 3:31 AM PST by [fix: add dns search record limit note. \(#18913\)](#) ([Page History](#))

[Edit This Page](#)

# 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.

- [Before you begin](#)
- [Create an nginx deployment and expose it via a service](#)
- [Test the service by accessing it from another Pod](#)
- [Limit access to the nginx service](#)

- [Assign the policy to the service](#)
- [Test access to the service when access label is not defined](#)
- [Define access label and test again](#)

## Before you begin

You need to have a Kubernetes cluster, and the kubectl command-line tool must be configured to communicate with your cluster. 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:

- [Calico](#)
- [Cilium](#)
- [Kube-router](#)
- [Romana](#)
- [Weave Net](#)

**Note:** The above list is sorted alphabetically by product name, not by recommendation or preference. This example is valid for a Kubernetes cluster using any of these providers.

## 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
service/kubernetes	10.100.0.1	<none>
TCP 46m		443/
service/nginx	10.100.0.16	<none>
TCP 33s		80/
NAME	READY	STATUS
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 --generator=run-pod/v1 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:

<a href="#">service/networking/nginx-policy.yaml</a>
<pre>apiVersion: networking.k8s.io/v1 kind: NetworkPolicy metadata:   name: access-nginx spec:   podSelector:     matchLabels:       app: nginx   ingress:   - from:     - podSelector:         matchLabels:           access: "true"</pre>

**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 --generator=run-pod/v1 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 --generator=run-pod/v1 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
```

## Feedback

Was this page helpful?

Yes No

Thanks for the feedback. If you have a specific, answerable question about how to use Kubernetes, ask it on [Stack Overflow](#). Open an issue in the GitHub repo if you want to [report a problem](#) or [suggest an improvement](#).

---

[Create an Issue](#) [Edit This Page](#)

Page last modified on February 12, 2020 at 2:06 PM PST by [Cleanup and implement style guidelines. \(#18980\)](#) ([Page History](#))

[Edit This Page](#)

# Developing Cloud Controller Manager

**FEATURE STATE:** Kubernetes v1.11 [beta](#)

This feature is currently in a *beta* state, meaning:

[Edit This Page](#)

## Enabling EndpointSlices

This page provides an overview of enabling EndpointSlices in Kubernetes.

- [Before you begin](#)
- [Introduction](#)
- [Enabling EndpointSlices](#)
- [Using EndpointSlices](#)
- [What's next](#)

### Before you begin

You need to have a Kubernetes cluster, and the `kubectl` command-line tool must be configured to communicate with your cluster. 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

EndpointSlices provide a scalable and extensible alternative to Endpoints in Kubernetes. They build on top of the base of functionality provided by Endpoints and extend that in a scalable way. When Services have a large number (>100) of network endpoints, they will be split into multiple smaller EndpointSlice resources instead of a single large Endpoints resource.

## Enabling EndpointSlices

**FEATURE STATE:** Kubernetes v1.17 [beta](#)

This feature is currently in a *beta* state, meaning:

[Edit This Page](#)

# Enabling Service Topology

This page provides an overview of enabling Service Topology in Kubernetes.

- [Before you begin](#)
- [Introduction](#)
- [Prerequisites](#)
- [Enable Service Topology](#)
- [What's next](#)

## Before you begin

You need to have a Kubernetes cluster, and the `kubectl` command-line tool must be configured to communicate with your cluster. 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

*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.

## Prerequisites

The following prerequisites are needed in order to enable topology aware service routing:

- Kubernetes 1.17 or later
- [Kube-proxy](#) [kube-proxy is a network proxy that runs on each node in the cluster.](#) running in iptables mode or IPVS mode
- Enable [Endpoint Slices](#)

## Enable Service Topology

**FEATURE STATE:** Kubernetes v1.17 [alpha](#)

This feature is currently in a *alpha* state, meaning:

[Edit This Page](#)

# Encrypting Secret Data at Rest

This page shows how to enable and configure encryption of secret data at rest.

- [Before you begin](#)
- [Configuration and determining whether encryption at rest is already enabled](#)
- [Understanding the encryption at rest configuration.](#)
- [Encrypting your data](#)
- [Verifying that data is encrypted](#)
- [Ensure all secrets are encrypted](#)
- [Rotating a decryption key](#)
- [Decrypting all data](#)

## Before you begin

- You need to have a Kubernetes cluster, and the kubectl command-line tool must be configured to communicate with your cluster. 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.

```
apiVersion: apiserver.config.k8s.io/v1
kind: EncryptionConfiguration
resources:
  - resources:
    - secrets
  providers:
    - identity: {}
    - aesgcm:
```

```

keys:
- name: key1
  secret: c2VjcmV0IGlzIHNlY3VyZQ==
- name: key2
  secret: dGhpccyBpcyBwYXNzd29yZA==
- aescbc:
  keys:
    - name: key1
      secret: c2VjcmV0IGlzIHNlY3VyZQ==
    - name: key2
      secret: dGhpccyBpcyBwYXNzd29yZA==
- 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
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.

Name	Encryption	Strength	Speed	Key Length	Other Considerations
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. <a href="#">Configure the KMS provider</a>

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. 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 the command `kubectl get secrets --all-namespaces -o json | kubectl replace -f -` to force all secrets to be decrypted.

## Feedback

Was this page helpful?

Yes No

Thanks for the feedback. If you have a specific, answerable question about how to use Kubernetes, ask it on [Stack Overflow](#). Open an issue in the GitHub repo if you want to [report a problem](#) or [suggest an improvement](#).

---

[Create an Issue](#) [Edit This Page](#)

Page last modified on February 06, 2020 at 9:03 PM PST by [Revise "Encrypting Secret Data at Rest" \(#18810\)](#) ([Page History](#))

[Edit This Page](#)

# Guaranteed Scheduling For Critical Add-On Pods

In addition to Kubernetes core components like api-server, scheduler, controller-manager running on a master machine there are a number of add-ons which, for various reasons, must run on a regular cluster node (rather than the Kubernetes master). 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. For static pods, this means it can't be evicted, but for non-static pods, it just means they will always be 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`.

## **Feedback**

Was this page helpful?

Yes No

Thanks for the feedback. If you have a specific, answerable question about how to use Kubernetes, ask it on [Stack Overflow](#). Open an issue in the GitHub repo if you want to [report a problem](#) or [suggest an improvement](#).

---

[Create an Issue](#) [Edit This Page](#)

Page last modified on December 04, 2019 at 10:36 AM PST by [Update guaranteed-scheduling-critical-addon-pods.md \(#17151\)](#) ([Page History](#))

[Edit This Page](#)

# IP Masquerade Agent User Guide

This page shows how to configure and enable the ip-masq-agent.

- [Before you begin](#)
- [Create an ip-masq-agent](#)
- [IP Masquerade Agent User Guide](#)

# Before you begin

You need to have a Kubernetes cluster, and the `kubectl` command-line tool must be configured to communicate with your cluster. 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 an ip-masq-agent

To create an ip-masq-agent, run the following `kubectl` command:

```
kubectl apply -f https://raw.githubusercontent.com/kubernetes-incubator/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 traffic should not be subject to
MASQUERADE */ ADDRTYPE match dst-type !LOCAL
RETURN   all  --  anywhere       10.0.0.0/8        /*
ip-masq-agent: cluster-local
MASQUERADE all  --  anywhere       anywhere          /
* ip-masq-agent: outbound traffic should be subject to
MASQUERADE (this match must come after cluster-local CIDR
matches) */ ADDRTYPE match dst-type !LOCAL

```

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

```

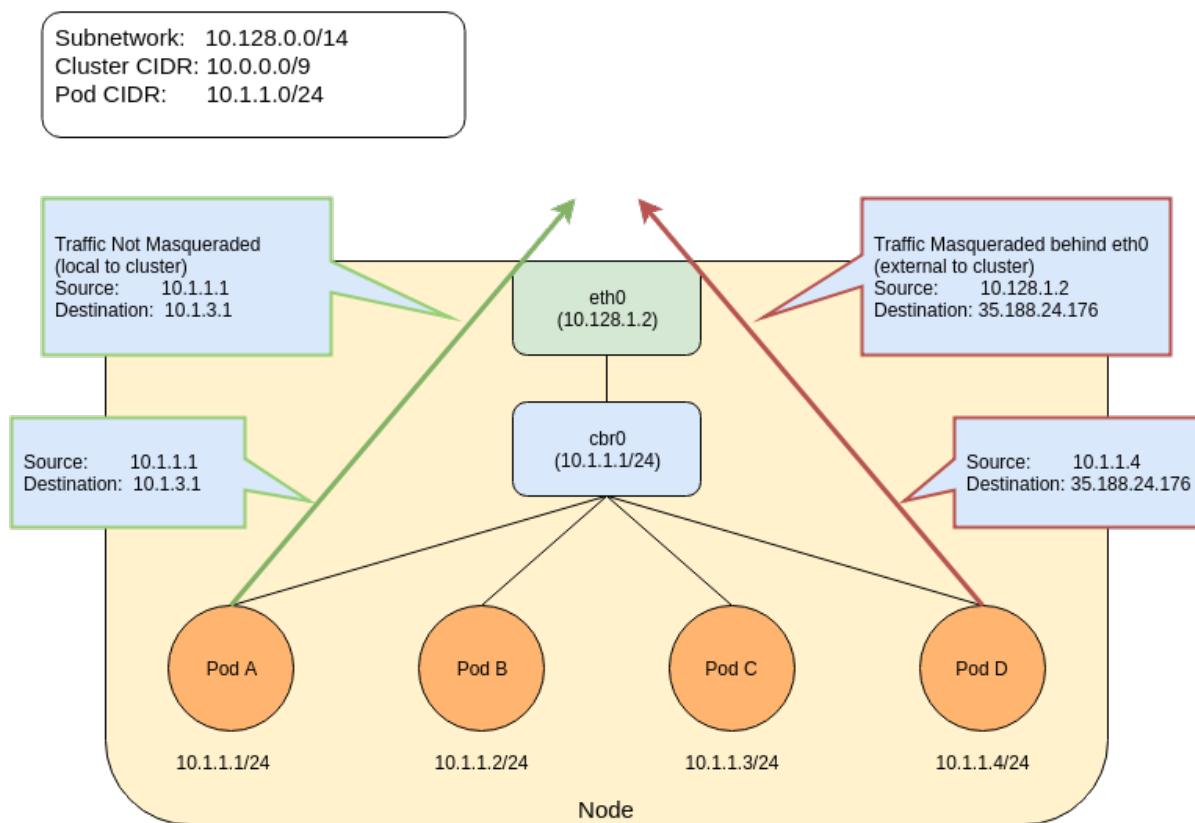
## IP Masquerade Agent User Guide

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**: An interval at which the agent attempts to reload config from disk. e.g. ~30s' where 's' is seconds, ~ms' is 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 traffic should not be subject to
MASQUERADE */ ADDRTYPE match dst-type !LOCAL
RETURN      all  --  anywhere           10.0.0.0/8       /*
ip-masq-agent: cluster-local traffic should not be subject to
MASQUERADE */ ADDRTYPE match dst-type !LOCAL
RETURN      all  --  anywhere           172.16.0.0/12      /*
ip-masq-agent: cluster-local traffic should not be subject to
MASQUERADE */ ADDRTYPE match dst-type !LOCAL
RETURN      all  --  anywhere           192.168.0.0/16     /*
ip-masq-agent: cluster-local traffic should not be subject to
MASQUERADE */ ADDRTYPE match dst-type !LOCAL
MASQUERADE  all  --  anywhere           anywhere          /
* ip-masq-agent: outbound traffic should be subject to
MASQUERADE (this match must come after cluster-local CIDR
matches) */ ADDRTYPE match dst-type !LOCAL
```

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:

## Feedback

Was this page helpful?

Yes No

Thanks for the feedback. If you have a specific, answerable question about how to use Kubernetes, ask it on [Stack Overflow](#). Open an issue in the GitHub repo if you want to [report a problem](#) or [suggest an improvement](#).

---

[Create an Issue](#) [Edit This Page](#)

Page last modified on March 25, 2019 at 5:06 PM PST by [Official 1.14](#)  
[Release Docs \(#13174\)](#) ([Page History](#))

[Edit This Page](#)

# Kubernetes Cloud Controller Manager

**FEATURE STATE:** Kubernetes v1.17 [beta](#)

This feature is currently in a *beta* state, meaning:

[Edit This Page](#)

# Limit Storage Consumption

This example demonstrates an easy way 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](#)
- [Scenario: Limiting Storage Consumption](#)
- [LimitRange to limit requests for storage](#)
- [StorageQuota to limit PVC count and cumulative storage capacity](#)
- [Summary](#)

## Before you begin

- You need to have a Kubernetes cluster, and the kubectl command-line tool must be configured to communicate with your cluster. 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. This allows a cluster-admin to plan their cluster's storage budget without risk of any one project going over their allotment.

## Feedback

Was this page helpful?

Yes No

Thanks for the feedback. If you have a specific, answerable question about how to use Kubernetes, ask it on [Stack Overflow](#). Open an issue in the GitHub repo if you want to [report a problem](#) or [suggest an improvement](#).

---

[Create an Issue](#) [Edit This Page](#)

Page last modified on May 05, 2018 at 6:00 PM PST by [Convert site to Hugo \(#8316\)](#) ([Page History](#))

[Edit This Page](#)

# Namespaces Walkthrough

Kubernetes [namespaces](#)[An abstraction used by Kubernetes to support multiple virtual clusters on the same physical cluster.](#) 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](#)
- [Prerequisites](#)
- [Understand the default namespace](#)
- [Create new namespaces](#)
- [Create pods in each 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. 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.

```
kubectl run snowflake --image=k8s.gcr.io/serve_hostname --
replicas=2
```

We have just created a deployment whose replica size is 2 that is running the pod called `snowflake` with a basic container that just serves the hostname. Note that `kubectl run` creates deployments only on Kubernetes cluster  $\geq v1.2$ . If you are running older versions, it creates replication controllers instead. If you want to obtain the old behavior, use `--generator=run/v1` to create replication controllers. See [kubectl run](#) for more details.

```
kubectl get deployment
```

NAME	READY	UP-TO-DATE	AVAILABLE	AGE
snowflake	2/2	2	2	2m

```
kubectl get pods -l run=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 run 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 run=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.

## Feedback

Was this page helpful?

Yes No

Thanks for the feedback. If you have a specific, answerable question about how to use Kubernetes, ask it on [Stack Overflow](#). Open an issue in the GitHub repo if you want to [report a problem](#) or [suggest an improvement](#).

---

[Create an Issue](#) [Edit This Page](#)

Page last modified on November 02, 2019 at 12:31 PM PST by [change](#)  
[kubectl get deployment output \(#17351\)](#) ([Page History](#))

[Edit This Page](#)

# 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](#)
- [Prerequisites](#)
- [Resource requirements](#)
- [Starting etcd clusters](#)
- [Securing etcd clusters](#)
- [Replacing a failed etcd member](#)
- [Backing up an etcd cluster](#)
- [Scaling up etcd clusters](#)
- [Restoring an etcd cluster](#)
- [Upgrading and rolling back etcd clusters](#)
- [Known issue: etcd client balancer with secure endpoints](#)

## Before you begin

You need to have a Kubernetes cluster, and the `kubectl` command-line tool must be configured to communicate with your cluster. 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 IO. 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 stable etcd clusters 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 documentation](#).

## 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 Kubernetes API server with the flag `--etcd-servers=$PRIVATE_IP:2379`.

Replace `PRIVATE_IP` with 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, http://$IP5:2379 --  
advertise-client-urls=http://$IP1:2379, http://$IP2:2379,  
http://$IP3:2379, http://$IP4:2379, http://$IP5:2379
```

2. Start Kubernetes API servers with the flag `--etcd-servers=$IP1:2379, $IP2:2379, $IP3:2379, $IP4:2379, $IP5:2379`.

Replace IP with 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 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 URL schema.

### **Limiting access of etcd clusters**

After configuring secure communication, restrict the access of etcd cluster to only the Kubernetes API server. 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 server the access, configure it 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,member4=http://
10.0.0.4:2380"
export ETCD_INITIAL_CLUSTER_STATE=existing
etcd [flags]
```

5. Do either of the following:

1. Update its `--etcd-servers` flag to make Kubernetes aware of the configuration changes, then restart the Kubernetes API server.
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 master 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, so backing up an etcd cluster is easy. 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 normally 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
# exit 0

# 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.

## 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](#). 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 server 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 server to fix the issue.

# Upgrading and rolling back etcd clusters

As of Kubernetes v1.13.0, etcd2 is no longer supported as a storage backend for new or existing Kubernetes clusters. The timeline for Kubernetes support for etcd2 and etcd3 is as follows:

- Kubernetes v1.0: etcd2 only
- Kubernetes v1.5.1: etcd3 support added, new clusters still default to etcd2
- Kubernetes v1.6.0: new clusters created with `kube-up.sh` default to etcd3, and `kube-apiserver` defaults to etcd3
- Kubernetes v1.9.0: deprecation of etcd2 storage backend announced
- Kubernetes v1.13.0: etcd2 storage backend removed, `kube-apiserver` will refuse to start with `--storage-backend=etcd2`, with the message `etcd2 is no longer a supported storage backend`

Before upgrading a v1.12.x `kube-apiserver` using `--storage-backend=etcd2` to v1.13.x, etcd v2 data must be migrated to the v3 storage backend and `kube-apiserver` invocations must be changed to use `--storage-backend=etcd3`.

The process for migrating from etcd2 to etcd3 is highly dependent on how the etcd cluster was deployed and configured, as well as how the Kubernetes cluster was deployed and configured. We recommend that you consult your cluster provider's documentation to see if there is a predefined solution.

If your cluster was created via `kube-up.sh` and is still using etcd2 as its storage backend, please consult the [Kubernetes v1.12 etcd cluster upgrade docs](#)

## Known issue: etcd client balancer with secure endpoints

The etcd v3 client, released in etcd v3.3.13 or earlier, has a [critical bug](#) which affects the `kube-apiserver` and HA deployments. The etcd client balancer failover does not properly work against secure endpoints. As a result, etcd servers may fail or disconnect briefly from the `kube-apiserver`. This affects `kube-apiserver` HA deployments.

The fix was made in [etcd v3.4](#) (and backported to v3.3.14 or later): the new client now creates its own credential bundle to correctly set authority target in dial function.

Because the fix requires gRPC dependency upgrade (to v1.23.0), downstream Kubernetes [did not backport etcd upgrades](#). Which means the [etcd fix in kube-apiserver](#) is only available from Kubernetes 1.16.

To urgently fix this bug for Kubernetes 1.15 or earlier, build a custom `kube-apiserver`. You can make local changes to [vendor/google.golang.org/grpc/credentials/credentials.go](#) with [etcd@db61ee106](mailto:etcd@db61ee106).

See "[kube-apiserver 1.13.x refuses to work when first etcd-server is not available](#)".

## Feedback

Was this page helpful?

Yes No

Thanks for the feedback. If you have a specific, answerable question about how to use Kubernetes, ask it on [Stack Overflow](#). Open an issue in the GitHub repo if you want to [report a problem](#) or [suggest an improvement](#).

---

[Create an Issue](#) [Edit This Page](#)

Page last modified on January 29, 2020 at 10:06 PM PST by [Fix small typos \(#18886\)](#) ([Page History](#))

[Edit This Page](#)

# Reconfigure a Node's Kubelet in a Live Cluster

**FEATURE STATE:** Kubernetes v1.11 [beta](#)

This feature is currently in a *beta* state, meaning:

[Edit This Page](#)

# 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](#)
- [Node Allocatable](#)
- [General Guidelines](#)
- [Example Scenario](#)
- [Feature Availability](#)

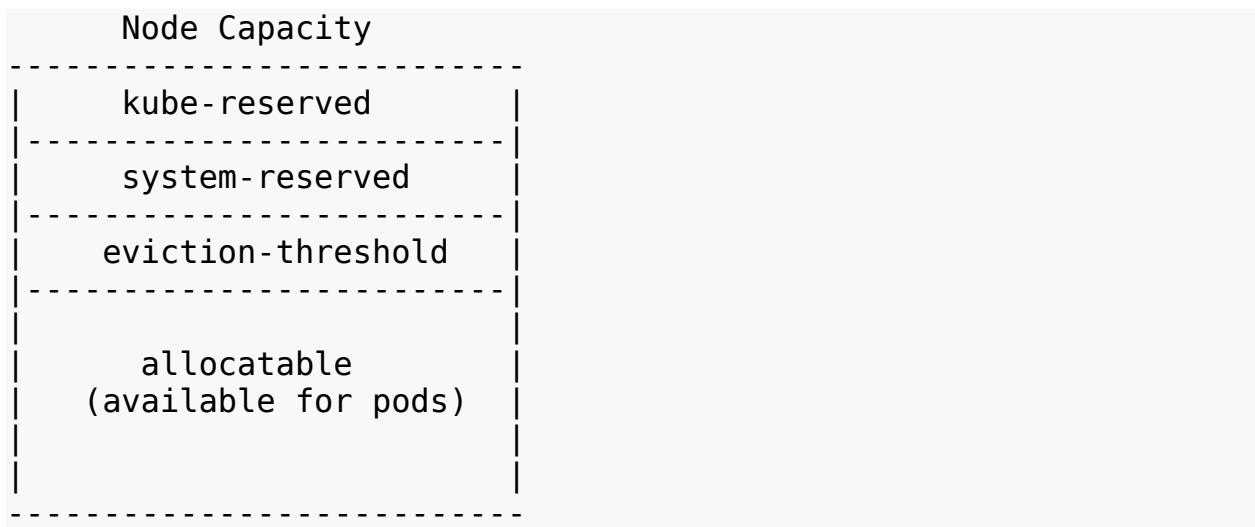
## Before you begin

You need to have a Kubernetes cluster, and the `kubectl` command-line tool must be configured to communicate with your cluster. 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`.

# 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. [This performance dashboard](#) exposes CPU and memory usage profiles of kubelet and docker engine at multiple levels of pod density. [This blog post](#) explains how the dashboard can be interpreted to come up with a suitable `kube-reserved` reservation.

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 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 [this doc](#) 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](#)

This feature is *stable*, meaning:

[Edit This Page](#)

# Safely Drain a Node while Respecting the PodDisruptionBudget

This page shows how to safely drain a machine, respecting the PodDisruptionBudget you have defined.

- [Before you begin](#)
- [Use kubectl drain to remove a node from service](#)
- [Draining multiple nodes in parallel](#)
- [The Eviction API](#)
- [What's next](#)

## Before you begin

This task assumes that you have met the following prerequisites:

- You are using Kubernetes release  $\geq 1.5$ .
- Either:
  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 [Configured PodDisruptionBudgets](#) for applications that need them.

## 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` will ignore 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 Pod DisruptionBudget for that set specifying `minAvailable`:

2. `kubectl drain` will only evict a pod from the StatefulSet if all three pods are ready, and if you issue multiple drain commands in parallel, Kubernetes will respect the PodDisruptionBudget and ensure that only one 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](#).

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

(perhaps more REST-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' http://  
127.0.0.1:8080/api/v1/namespaces/default/pods/quux/eviction -d  
@eviction.json
```

The API can respond in one of three ways:

- If the eviction is granted, then the pod is deleted just as if you had sent a DELETE request to the pod's URL and you get 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. Currently, callers do not get any Retry-After advice, but they may in future versions.
- If there is some kind of misconfiguration, like multiple budgets pointing at the same pod, you will get 500 Internal Server Error.

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.

In some cases, an application may reach a broken state where it will never return anything other than 429 or 500. This can happen, for example, if the replacement pod created by the application's controller does not become ready, or 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 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](#).

## Feedback

Was this page helpful?

Yes No

Thanks for the feedback. If you have a specific, answerable question about how to use Kubernetes, ask it on [Stack Overflow](#). Open an issue in the GitHub repo if you want to [report a problem](#) or [suggest an improvement](#).

---

[Create an Issue](#) [Edit This Page](#)

Page last modified on May 08, 2019 at 2:40 PM PST by [Directly reference the PodDisruptionBudget \(#14209\)](#) ([Page History](#))

[Edit This Page](#)

# 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](#)
- [Controlling access to the Kubernetes API](#)
- [Controlling access to the Kubelet](#)
- [Controlling the capabilities of a workload or user at runtime](#)
- [Protecting cluster components from compromise](#)

## Before you begin

- You need to have a Kubernetes cluster, and the kubectl command-line tool must be configured to communicate with your cluster. 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.

# Feedback

Was this page helpful?

Yes No

Thanks for the feedback. If you have a specific, answerable question about how to use Kubernetes, ask it on [Stack Overflow](#). Open an issue in the GitHub repo if you want to [report a problem](#) or [suggest an improvement](#).

---

[Create an Issue](#) [Edit This Page](#)

Page last modified on October 28, 2019 at 7:15 AM PST by [Update k8s version info for etcd encryption feature in securing-a-cluster. \(#17027\)](#)  
[\(Page History\)](#)

[Edit This Page](#)

# Set Kubelet parameters via a config file

**FEATURE STATE:** Kubernetes v1.17 [beta](#)

This feature is currently in a *beta* state, meaning:

[Edit This Page](#)

# Set up High-Availability Kubernetes Masters

**FEATURE STATE:** Kubernetes 1.5 [alpha](#)

This feature is currently in a *alpha* state, meaning:

[Edit This Page](#)

# Share a Cluster with Namespaces

This page shows how to view, work in, and delete namespacesAn abstraction used by Kubernetes to support multiple virtual clusters on the same physical cluster.. The page also shows how to use Kubernetes namespaces to subdivide your cluster.

- [Before you begin](#)
- [Viewing namespaces](#)
- [Creating a new namespace](#)
- [Deleting a namespace](#)
- [Subdividing your cluster using Kubernetes namespaces](#)
- [Understanding the motivation for using namespaces](#)
- [Understanding namespaces and DNS](#)
- [What's next](#)

## Before you begin

- Have an [existing Kubernetes cluster](#).
- 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

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>
```

Note that the name of your namespace must be a DNS compatible 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.

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 values 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
```

The above commands provided two request contexts you can alternate against depending on what namespace you wish to work against.

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.

```
kubectl run snowflake --image=k8s.gcr.io/serve_hostname --  
replicas=2
```

We have just created a deployment whose replica size is 2 that is running the pod called `snowflake` with a basic container that just serves the hostname. Note that `kubectl run` creates deployments only on Kubernetes cluster  $\geq v1.2$ . If you are running older versions, it creates replication controllers instead. If you want to obtain the old behavior, use `--generator=run/v1` to create replication controllers. See [kubectl run](#) for more details.

```
kubectl get deployment
```

NAME	READY	UP-TO-DATE	AVAILABLE	AGE
snowflake	2/2	2	2	2m

```
kubectl get pods -l run=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 run 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 run=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.

## 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 just 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](#).

## Feedback

Was this page helpful?

Yes No

Thanks for the feedback. If you have a specific, answerable question about how to use Kubernetes, ask it on [Stack Overflow](#). Open an issue in the GitHub repo if you want to [report a problem](#) or [suggest an improvement](#).

---

[Create an Issue](#) [Edit This Page](#)

Page last modified on January 09, 2020 at 10:55 PM PST by [Update namespaces.md \(#18373\)](#) ([Page History](#))

[Edit This Page](#)

# 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](#)
- [Configuring the KMS provider](#)
- [Implementing a KMS plugin](#)

- [Encrypting your data with the KMS provider](#)
- [Verifying that the data is encrypted](#)
- [Ensuring all secrets are encrypted](#)
- [Switching from a local encryption provider to the KMS provider](#)
- [Disabling encryption 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. 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](#)

This feature is currently in a *beta* state, meaning:

[Edit This Page](#)

# Using CoreDNS for Service Discovery

This page describes the CoreDNS upgrade process and how to install CoreDNS instead of kube-dns.

- [Before you begin](#)
- [About CoreDNS](#)
- [Installing CoreDNS](#)
- [Migrating to CoreDNS](#)
- [Upgrading CoreDNS](#)
- [Tuning CoreDNS](#)
- [What's next](#)

## Before you begin

You need to have a Kubernetes cluster, and the kubectl command-line tool must be configured to communicate with your cluster. 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 Cloud Native Computing Foundation](#).

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 federation, 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. Follow the guide outlined [here](#) if you want your upgraded cluster to use kube-dns.

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.

## Installing kube-dns instead of CoreDNS with kubeadm

**Note:** In Kubernetes 1.11, CoreDNS has graduated to General Availability (GA) and is installed by default.

To install kube-dns on versions prior to 1.13, set the CoreDNS feature gate value to `false`:

```
kubeadm init --feature-gates=CoreDNS=false
```

For versions 1.13 and later, follow the guide outlined [here](#).

## 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](#).

## Feedback

Was this page helpful?

Yes No

Thanks for the feedback. If you have a specific, answerable question about how to use Kubernetes, ask it on [Stack Overflow](#). Open an issue in the GitHub repo if you want to [report a problem](#) or [suggest an improvement](#).

---

[Create an Issue](#) [Edit This Page](#)

Page last modified on July 12, 2019 at 2:03 AM PST by [Add "CNCF" to glossary \(#14537\)](#) ([Page History](#))

[Edit This Page](#)

# Using NodeLocal DNSCache in Kubernetes clusters

**FEATURE STATE:** Kubernetes v1.15 [beta](#)

This feature is currently in a *beta* state, meaning:

[Edit This Page](#)

# Using sysctls in a Kubernetes Cluster

**FEATURE STATE:** Kubernetes v1.12 [beta](#)

This feature is currently in a *beta* state, meaning:

[Edit This Page](#)

## Assign Memory Resources to Containers and Pods

This page shows how to assign a memory *request* and a memory *limit* to a Container. A Container is guaranteed to have as much memory as it requests, but is not allowed to use more memory than its limit.

- [Before you begin](#)
- [Create a namespace](#)
- [Specify a memory request and a memory limit](#)
- [Exceed a Container's memory limit](#)
- [Specify a memory request that is too big for your Nodes](#)
- [Memory units](#)
- [If you do not specify a memory limit](#)
- [Motivation for memory requests and limits](#)
- [Clean up](#)
- [What's next](#)

### Before you begin

You need to have a Kubernetes cluster, and the `kubectl` command-line tool must be configured to communicate with your cluster. 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 300 MiB of memory.

A few of the steps on this page require you to run the [metrics-server](#) service in your cluster. If you have the metrics-server running, you can skip those steps.

If you are running Minikube, run the following command to enable the metrics-server:

```
minikube addons enable metrics-server
```

To see whether the metrics-server is running, or another provider of the resource metrics API (`metrics.k8s.io`), run the following command:

```
kubectl get apiservices
```

If the resource metrics API is available, the output includes a reference to `metrics.k8s.io`.

```
NAME
v1beta1.metrics.k8s.io
```

## 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 mem-example
```

## Specify a memory request and a memory limit

To specify a memory request for a Container, include the `resources: requests` field in the Container's resource manifest. To specify a memory limit, include `resources:limits`.

In this exercise, you create a Pod that has one Container. The Container has a memory request of 100 MiB and a memory limit of 200 MiB. Here's the configuration file for the Pod:

```
pods/resource/memory-request-limit.yaml

apiVersion: v1
kind: Pod
metadata:
  name: memory-demo
  namespace: mem-example
spec:
  containers:
    - name: memory-demo-ctr
      image: polinux/stress
      resources:
        limits:
          memory: "200Mi"
        requests:
          memory: "100Mi"
      command: ["stress"]
      args: [--vm, "1", "--vm-bytes", "150M", "--vm-hang", "1"]
```

The `args` section in the configuration file provides arguments for the Container when it starts. The `--vm-bytes`, `"150M"` arguments tell the Container to attempt to allocate 150 MiB of memory.

Create the Pod:

```
kubectl apply -f https://k8s.io/examples/pods/resource/memory-request-limit.yaml --namespace=mem-example
```

Verify that the Pod Container is running:

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

View detailed information about the Pod:

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

The output shows that the one Container in the Pod has a memory request of 100 MiB and a memory limit of 200 MiB.

```
...
resources:
  limits:
    memory: 200Mi
  requests:
    memory: 100Mi
...
```

Run `kubectl top` to fetch the metrics for the pod:

```
kubectl top pod memory-demo --namespace=mem-example
```

The output shows that the Pod is using about 162,900,000 bytes of memory, which is about 150 MiB. This is greater than the Pod's 100 MiB request, but within the Pod's 200 MiB limit.

NAME	CPU(cores)	MEMORY(bytes)
memory-demo	<something>	162856960

Delete your Pod:

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

## Exceed a Container's memory limit

A Container can exceed its memory request if the Node has memory available. But a Container is not allowed to use more than its memory limit. If a Container allocates more memory than its limit, the Container becomes a candidate for termination. If the Container continues to consume memory beyond its limit, the Container is terminated. If a terminated Container can be restarted, the kubelet restarts it, as with any other type of runtime failure.

In this exercise, you create a Pod that attempts to allocate more memory than its limit. Here is the configuration file for a Pod that has one Container with a memory request of 50 MiB and a memory limit of 100 MiB:

## [pods/resource/memory-request-limit-2.yaml](#)

```
apiVersion: v1
kind: Pod
metadata:
  name: memory-demo-2
  namespace: mem-example
spec:
  containers:
    - name: memory-demo-2-ctr
      image: polinux/stress
      resources:
        requests:
          memory: "50Mi"
        limits:
          memory: "100Mi"
      command: ["stress"]
      args: [--vm, "1", "--vm-bytes", "250M", "--vm-hang", "1"]
```

In the `args` section of the configuration file, you can see that the Container will attempt to allocate 250 MiB of memory, which is well above the 100 MiB limit.

Create the Pod:

```
kubectl apply -f https://k8s.io/examples/pods/resource/memory-
request-limit-2.yaml --namespace=mem-example
```

View detailed information about the Pod:

```
kubectl get pod memory-demo-2 --namespace=mem-example
```

At this point, the Container might be running or killed. Repeat the preceding command until the Container is killed:

NAME	READY	STATUS	RESTARTS	AGE
memory-demo-2	0/1	00MKilled	1	24s

Get a more detailed view of the Container status:

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

The output shows that the Container was killed because it is out of memory (OOM):

```
lastState:
  terminated:
    containerID: docker://
65183c1877aaec2e8427bc95609cc52677a454b56fcb24340dbd22917c23b10f
    exitCode: 137
```

```
finishedAt: 2017-06-20T20:52:19Z
reason: OOMKilled
startedAt: null
```

The Container in this exercise can be restarted, so the kubelet restarts it. Repeat this command several times to see that the Container is repeatedly killed and restarted:

```
kubectl get pod memory-demo-2 --namespace=mem-example
```

The output shows that the Container is killed, restarted, killed again, restarted again, and so on:

```
kubectl get pod memory-demo-2 --namespace=mem-example
NAME          READY   STATUS    RESTARTS   AGE
memory-demo-2  0/1     OOMKilled   1          37s
```

```
kubectl get pod memory-demo-2 --namespace=mem-example
NAME          READY   STATUS    RESTARTS   AGE
memory-demo-2  1/1     Running   2          40s
```

View detailed information about the Pod history:

```
kubectl describe pod memory-demo-2 --namespace=mem-example
```

The output shows that the Container starts and fails repeatedly:

```
... Normal Created  Created container with id
66a3a20aa7980e61be4922780bf9d24d1a1d8b7395c09861225b0eba1b1f8511
... Warning BackOff  Back-off restarting failed container
```

View detailed information about your cluster's Nodes:

```
kubectl describe nodes
```

The output includes a record of the Container being killed because of an out-of-memory condition:

```
Warning OOMKilling Memory cgroup out of memory: Kill process
4481 (stress) score 1994 or sacrifice child
```

Delete your Pod:

```
kubectl delete pod memory-demo-2 --namespace=mem-example
```

## Specify a memory request that is too big for your Nodes

Memory requests and limits are associated with Containers, but it is useful to think of a Pod as having a memory request and limit. The memory request for the Pod is the sum of the memory requests for all the Containers in the

Pod. Likewise, the memory limit for the Pod is the sum of the limits of all the Containers in the Pod.

Pod scheduling is based on requests. A Pod is scheduled to run on a Node only if the Node has enough available memory to satisfy the Pod's memory request.

In this exercise, you create a Pod that has a memory request so big that it exceeds the capacity of any Node in your cluster. Here is the configuration file for a Pod that has one Container with a request for 1000 GiB of memory, which likely exceeds the capacity of any Node in your cluster.

#### [pods/resource/memory-request-limit-3.yaml](#)

```
apiVersion: v1
kind: Pod
metadata:
  name: memory-demo-3
  namespace: mem-example
spec:
  containers:
    - name: memory-demo-3-ctr
      image: polinux/stress
      resources:
        limits:
          memory: "1000Gi"
        requests:
          memory: "1000Gi"
      command: ["stress"]
      args: ["--vm", "1", "--vm-bytes", "150M", "--vm-hang", "1"]
```

Create the Pod:

```
kubectl apply -f https://k8s.io/examples/pods/resource/memory-
request-limit-3.yaml --namespace=mem-example
```

View the Pod status:

```
kubectl get pod memory-demo-3 --namespace=mem-example
```

The output shows that the Pod status is PENDING. That is, the Pod is not scheduled to run on any Node, and it will remain in the PENDING state indefinitely:

```
kubectl get pod memory-demo-3 --namespace=mem-example
NAME           READY   STATUS    RESTARTS   AGE
memory-demo-3  0/1     Pending   0          25s
```

View detailed information about the Pod, including events:

```
kubectl describe pod memory-demo-3 --namespace=mem-example
```

The output shows that the Container cannot be scheduled because of insufficient memory on the Nodes:

Events:	Reason	Message
	-----	-----
	... FailedScheduling	No nodes are available that match all of the following predicates:: Insufficient memory (3).

## Memory units

The memory resource is measured in bytes. You can express memory as a plain integer or a fixed-point integer with one of these suffixes: E, P, T, G, M, K, Ei, Pi, Ti, Gi, Mi, Ki. For example, the following represent approximately the same value:

```
128974848, 129e6, 129M , 123Mi
```

Delete your Pod:

```
kubectl delete pod memory-demo-3 --namespace=mem-example
```

## If you do not specify a memory limit

If you do not specify a memory limit for a Container, one of the following situations applies:

- The Container has no upper bound on the amount of memory it uses. The Container could use all of the memory available on the Node where it is running which in turn could invoke the OOM Killer. Further, in case of an OOM Kill, a container with no resource limits will have a greater chance of being killed.
- The Container is running in a namespace that has a default memory limit, and the Container is automatically assigned the default limit. Cluster administrators can use a [LimitRange](#) to specify a default value for the memory limit.

## Motivation for memory requests and limits

By configuring memory requests and limits for the Containers that run in your cluster, you can make efficient use of the memory resources available on your cluster's Nodes. By keeping a Pod's memory request low, you give the Pod a good chance of being scheduled. By having a memory limit that is greater than the memory request, you accomplish two things:

- The Pod can have bursts of activity where it makes use of memory that happens to be available.
- The amount of memory a Pod can use during a burst is limited to some reasonable amount.

# Clean up

Delete your namespace. This deletes all the Pods that you created for this task:

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

## What's next

### For app developers

- [Assign CPU Resources to Containers and Pods](#)
- [Configure Quality of Service for Pods](#)

### 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](#)
- [Configure Quotas for API Objects](#)

## Feedback

Was this page helpful?

Yes No

Thanks for the feedback. If you have a specific, answerable question about how to use Kubernetes, ask it on [Stack Overflow](#). Open an issue in the GitHub repo if you want to [report a problem](#) or [suggest an improvement](#).

---

[Create an Issue](#) [Edit This Page](#)

Page last modified on April 10, 2019 at 6:50 AM PST by [Add Info about OOM Kill \(#13723\)](#) ([Page History](#))

[Edit This Page](#)

# Assign CPU Resources to Containers and Pods

This page shows how to assign a CPU *request* and a CPU *limit* to a container. Containers cannot use more CPU than the configured limit. Provided the system has CPU time free, a container is guaranteed to be allocated as much CPU as it requests.

- [Before you begin](#)

- [Create a namespace](#)
- [Specify a CPU request and a CPU limit](#)
- [CPU units](#)
- [Specify a CPU request that is too big for your Nodes](#)
- [If you do not specify a CPU limit](#)
- [Motivation for CPU requests and limits](#)
- [Clean up](#)
- [What's next](#)

## Before you begin

You need to have a Kubernetes cluster, and the `kubectl` command-line tool must be configured to communicate with your cluster. 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 CPU.

A few of the steps on this page require you to run the [metrics-server](#) service in your cluster. If you have the metrics-server running, you can skip those steps.

If you are running [MinikubeA tool for running Kubernetes locally](#), run the following command to enable metrics-server:

```
minikube addons enable metrics-server
```

To see whether metrics-server (or another provider of the resource metrics API, `metrics.k8s.io`) is running, type the following command:

```
kubectl get apiservices
```

If the resource metrics API is available, the output will include a reference to `metrics.k8s.io`.

```
NAME  
v1beta1.metrics.k8s.io
```

## Create a namespace

Create a [NamespaceAn abstraction used by Kubernetes to support multiple virtual clusters on the same physical cluster](#), so that the resources you create in this exercise are isolated from the rest of your cluster.

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

# Specify a CPU request and a CPU limit

To specify a CPU request for a container, include the `resources.requests` field in the Container resource manifest. To specify a CPU limit, include `resources.limits`.

In this exercise, you create a Pod that has one container. The container has a request of 0.5 CPU and a limit of 1 CPU. Here is the configuration file for the Pod:

[pods/resource/cpu-request-limit.yaml](#)

```
apiVersion: v1
kind: Pod
metadata:
  name: cpu-demo
  namespace: cpu-example
spec:
  containers:
    - name: cpu-demo-ctr
      image: vish/stress
      resources:
        limits:
          cpu: "1"
        requests:
          cpu: "0.5"
      args:
        - -cpus
        - "2"
```

The `args` section of the configuration file provides arguments for the container when it starts. The `-cpus "2"` argument tells the Container to attempt to use 2 CPUs.

Create the Pod:

```
kubectl apply -f https://k8s.io/examples/pods/resource/cpu-
request-limit.yaml --namespace=cpu-example
```

Verify that the Pod is running:

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

View detailed information about the Pod:

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

The output shows that the one container in the Pod has a CPU request of 500 milliCPU and a CPU limit of 1 CPU.

```
resources:  
  limits:  
    cpu: "1"  
  requests:  
    cpu: 500m
```

Use `kubectl top` to fetch the metrics for the pod:

```
kubectl top pod cpu-demo --namespace(cpu-example)
```

This example output shows that the Pod is using 974 milliCPU, which is just a bit less than the limit of 1 CPU specified in the Pod configuration.

NAME	CPU(cores)	MEMORY(bytes)
cpu-demo	974m	<something>

Recall that by setting `-cpu "2"`, you configured the Container to attempt to use 2 CPUs, but the Container is only being allowed to use about 1 CPU. The container's CPU use is being throttled, because the container is attempting to use more CPU resources than its limit.

**Note:** Another possible explanation for the CPU use being below 1.0 is that the Node might not have enough CPU resources available. Recall that the prerequisites for this exercise require each of your Nodes to have at least 1 CPU. If your Container runs on a Node that has only 1 CPU, the Container cannot use more than 1 CPU regardless of the CPU limit specified for the Container.

## CPU units

The CPU resource is measured in *CPU* units. One CPU, in Kubernetes, is equivalent to:

- 1 AWS vCPU
- 1 GCP Core
- 1 Azure vCore
- 1 Hyperthread on a bare-metal Intel processor with Hyperthreading

Fractional values are allowed. A Container that requests 0.5 CPU is guaranteed half as much CPU as a Container that requests 1 CPU. You can use the suffix `m` to mean milli. For example 100m CPU, 100 milliCPU, and 0.1 CPU are all the same. Precision finer than 1m is not allowed.

CPU is always requested as an absolute quantity, never as a relative quantity; 0.1 is the same amount of CPU on a single-core, dual-core, or 48-core machine.

Delete your Pod:

```
kubectl delete pod cpu-demo --namespace(cpu-example)
```

# Specify a CPU request that is too big for your Nodes

CPU requests and limits are associated with Containers, but it is useful to think of a Pod as having a CPU request and limit. The CPU request for a Pod is the sum of the CPU requests for all the Containers in the Pod. Likewise, the CPU limit for a Pod is the sum of the CPU limits for all the Containers in the Pod.

Pod scheduling is based on requests. A Pod is scheduled to run on a Node only if the Node has enough CPU resources available to satisfy the Pod CPU request.

In this exercise, you create a Pod that has a CPU request so big that it exceeds the capacity of any Node in your cluster. Here is the configuration file for a Pod that has one Container. The Container requests 100 CPU, which is likely to exceed the capacity of any Node in your cluster.

## [pods/resource/cpu-request-limit-2.yaml](#)

```
apiVersion: v1
kind: Pod
metadata:
  name: cpu-demo-2
  namespace: cpu-example
spec:
  containers:
    - name: cpu-demo-ctr-2
      image: vish/stress
      resources:
        limits:
          cpu: "100"
        requests:
          cpu: "100"
      args:
        - -cpus
        - "2"
```

Create the Pod:

```
kubectl apply -f https://k8s.io/examples/pods/resource/cpu-
request-limit-2.yaml --namespace=cpu-example
```

View the Pod status:

```
kubectl get pod cpu-demo-2 --namespace=cpu-example
```

The output shows that the Pod status is Pending. That is, the Pod has not been scheduled to run on any Node, and it will remain in the Pending state indefinitely:

```
kubectl get pod cpu-demo-2 --namespace(cpu-example)
```

NAME	READY	STATUS	RESTARTS	AGE
cpu-demo-2	0/1	Pending	0	7m

View detailed information about the Pod, including events:

```
kubectl describe pod cpu-demo-2 --namespace(cpu-example)
```

The output shows that the Container cannot be scheduled because of insufficient CPU resources on the Nodes:

Events:	
Reason	Message
FailedScheduling	No nodes are available that match all of the following predicates:: Insufficient cpu (3).

Delete your Pod:

```
kubectl delete pod cpu-demo-2 --namespace(cpu-example)
```

## If you do not specify a CPU limit

If you do not specify a CPU limit for a Container, then one of these situations applies:

- The Container has no upper bound on the CPU resources it can use. The Container could use all of the CPU resources available on the Node where it is running.
- The Container is running in a namespace that has a default CPU limit, and the Container is automatically assigned the default limit. Cluster administrators can use a [LimitRange](#) to specify a default value for the CPU limit.

## Motivation for CPU requests and limits

By configuring the CPU requests and limits of the Containers that run in your cluster, you can make efficient use of the CPU resources available on your cluster Nodes. By keeping a Pod CPU request low, you give the Pod a good chance of being scheduled. By having a CPU limit that is greater than the CPU request, you accomplish two things:

- The Pod can have bursts of activity where it makes use of CPU resources that happen to be available.
- The amount of CPU resources a Pod can use during a burst is limited to some reasonable amount.

# Clean up

Delete your namespace:

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

# What's next

## For app developers

- [Assign Memory Resources to Containers and Pods](#)
- [Configure Quality of Service for Pods](#)

## 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](#)
- [Configure Quotas for API Objects](#)

# Feedback

Was this page helpful?

Yes No

Thanks for the feedback. If you have a specific, answerable question about how to use Kubernetes, ask it on [Stack Overflow](#). Open an issue in the GitHub repo if you want to [report a problem](#) or [suggest an improvement](#).

---

[Create an Issue](#) [Edit This Page](#)

Page last modified on July 07, 2019 at 5:02 AM PST by [Tidy assign-cpu-resource page \(#15126\)](#) ([Page History](#))

[Edit This Page](#)

# Configure GMSA for Windows Pods and containers

**FEATURE STATE:** Kubernetes v1.16 [beta](#)

This feature is currently in a *beta* state, meaning:

[Edit This Page](#)

# Configure RunAsUserName for Windows pods and containers

**FEATURE STATE:** Kubernetes v1.17 [beta](#)

This feature is currently in a *beta* state, meaning:

[Edit This Page](#)

# Configure Quality of Service for Pods

This page shows how to configure Pods so that they will be assigned particular Quality of Service (QoS) classes. Kubernetes uses QoS classes to make decisions about scheduling and evicting Pods.

- [Before you begin](#)
- [QoS classes](#)
- [Create a namespace](#)
- [Create a Pod that gets assigned a QoS class of Guaranteed](#)
- [Create a Pod that gets assigned a QoS class of Burstable](#)
- [Create a Pod that gets assigned a QoS class of BestEffort](#)
- [Create a Pod that has two Containers](#)
- [Clean up](#)
- [What's next](#)

## Before you begin

You need to have a Kubernetes cluster, and the `kubectl` command-line tool must be configured to communicate with your cluster. 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`.

## QoS classes

When Kubernetes creates a Pod it assigns one of these QoS classes to the Pod:

- Guaranteed
- Burstable
- BestEffort

# 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 qos-example
```

## Create a Pod that gets assigned a QoS class of Guaranteed

For a Pod to be given a QoS class of Guaranteed:

- Every Container in the Pod must have a memory limit and a memory request, and they must be the same.
- Every Container in the Pod must have a CPU limit and a CPU request, and they must be the same.

Here is the configuration file for a Pod that has one Container. The Container has a memory limit and a memory request, both equal to 200 MiB. The Container has a CPU limit and a CPU request, both equal to 700 milliCPU:

```
pods/qos/qos-pod.yaml
```

apiVersion: v1
kind: Pod
metadata:
name: qos-demo
namespace: qos-example
spec:
containers:
- name: qos-demo-ctr
image: nginx
resources:
limits:
memory: "200Mi"
cpu: "700m"
requests:
memory: "200Mi"
cpu: "700m"

Create the Pod:

```
kubectl apply -f https://k8s.io/examples/pods/qos/qos-pod.yaml --namespace=qos-example
```

View detailed information about the Pod:

```
kubectl get pod qos-demo --namespace=qos-example --output=yaml
```

The output shows that Kubernetes gave the Pod a QoS class of Guaranteed. The output also verifies that the Pod Container has a memory request that matches its memory limit, and it has a CPU request that matches its CPU limit.

```
spec:  
  containers:  
    ...  
    resources:  
      limits:  
        cpu: 700m  
        memory: 200Mi  
      requests:  
        cpu: 700m  
        memory: 200Mi  
    ...  
  qosClass: Guaranteed
```

**Note:** If a Container specifies its own memory limit, but does not specify a memory request, Kubernetes automatically assigns a memory request that matches the limit. Similarly, if a Container specifies its own CPU limit, but does not specify a CPU request, Kubernetes automatically assigns a CPU request that matches the limit.

Delete your Pod:

```
kubectl delete pod qos-demo --namespace=qos-example
```

## Create a Pod that gets assigned a QoS class of Burstable

A Pod is given a QoS class of Burstable if:

- The Pod does not meet the criteria for QoS class Guaranteed.
- At least one Container in the Pod has a memory or CPU request.

Here is the configuration file for a Pod that has one Container. The Container has a memory limit of 200 MiB and a memory request of 100 MiB.

### [pods/qos/qos-pod-2.yaml](#)

```
apiVersion: v1
kind: Pod
metadata:
  name: qos-demo-2
  namespace: qos-example
spec:
  containers:
    - name: qos-demo-2-ctr
      image: nginx
      resources:
        limits:
          memory: "200Mi"
        requests:
          memory: "100Mi"
```

Create the Pod:

```
kubectl apply -f https://k8s.io/examples/pods/qos/qos-pod-2.yaml
--namespace=qos-example
```

View detailed information about the Pod:

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

The output shows that Kubernetes gave the Pod a QoS class of Burstable.

```
spec:
  containers:
    - image: nginx
      imagePullPolicy: Always
      name: qos-demo-2-ctr
      resources:
        limits:
          memory: 200Mi
        requests:
          memory: 100Mi
...
  qosClass: Burstable
```

Delete your Pod:

```
kubectl delete pod qos-demo-2 --namespace=qos-example
```

## **Create a Pod that gets assigned a QoS class of BestEffort**

For a Pod to be given a QoS class of BestEffort, the Containers in the Pod must not have any memory or CPU limits or requests.

Here is the configuration file for a Pod that has one Container. The Container has no memory or CPU limits or requests:

#### [pods/qos/qos-pod-3.yaml](#)

```
apiVersion: v1
kind: Pod
metadata:
  name: qos-demo-3
  namespace: qos-example
spec:
  containers:
    - name: qos-demo-3-ctr
      image: nginx
```

Create the Pod:

```
kubectl apply -f https://k8s.io/examples/pods/qos/qos-pod-3.yaml
--namespace=qos-example
```

View detailed information about the Pod:

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

The output shows that Kubernetes gave the Pod a QoS class of BestEffort.

```
spec:
  containers:
    ...
    resources: {}
  ...
  qosClass: BestEffort
```

Delete your Pod:

```
kubectl delete pod qos-demo-3 --namespace=qos-example
```

## Create a Pod that has two Containers

Here is the configuration file for a Pod that has two Containers. One container specifies a memory request of 200 MiB. The other Container does not specify any requests or limits.

### [pods/qos/qos-pod-4.yaml](#)

```
apiVersion: v1
kind: Pod
metadata:
  name: qos-demo-4
  namespace: qos-example
spec:
  containers:
    - name: qos-demo-4-ctr-1
      image: nginx
      resources:
        requests:
          memory: "200Mi"
    - name: qos-demo-4-ctr-2
      image: redis
```

Notice that this Pod meets the criteria for QoS class **Burstable**. That is, it does not meet the criteria for QoS class **Guaranteed**, and one of its Containers has a memory request.

Create the Pod:

```
kubectl apply -f https://k8s.io/examples/pods/qos/qos-pod-4.yaml
--namespace=qos-example
```

View detailed information about the Pod:

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

The output shows that Kubernetes gave the Pod a QoS class of **Burstable**:

```
spec:
  containers:
    ...
    name: qos-demo-4-ctr-1
    resources:
      requests:
        memory: 200Mi
    ...
    name: qos-demo-4-ctr-2
    resources: {}
  ...
  qosClass: Burstable
```

Delete your Pod:

```
kubectl delete pod qos-demo-4 --namespace=qos-example
```

# Clean up

Delete your namespace:

```
kubectl delete namespace qos-example
```

# What's next

## For app developers

- [Assign Memory Resources to Containers and Pods](#)
- [Assign CPU Resources to Containers and Pods](#)

## 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](#)
- [Configure Quotas for API Objects](#)
- [Control Topology Management policies on a node](#)

# Feedback

Was this page helpful?

Yes No

Thanks for the feedback. If you have a specific, answerable question about how to use Kubernetes, ask it on [Stack Overflow](#). Open an issue in the GitHub repo if you want to [report a problem](#) or [suggest an improvement](#).

---

[Create an Issue](#) [Edit This Page](#)

Page last modified on September 09, 2019 at 1:39 PM PST by [Added documentation to support Topology Manager feature in Kubelet. \(#15716\)](#)  
[\(Page History\)](#)

[Edit This Page](#)

# Assign Extended Resources to a Container

**FEATURE STATE:** Kubernetes v1.17 [stable](#)

This feature is *stable*, meaning:

[Edit This Page](#)

# Configure a Pod to Use a Volume for Storage

This page shows how to configure a Pod to use a Volume for storage.

A Container's file system lives only as long as the Container does. So when a Container terminates and restarts, filesystem changes are lost. For more consistent storage that is independent of the Container, you can use a [Volume](#). This is especially important for stateful applications, such as key-value stores (such as Redis) and databases.

- [Before you begin](#)
- [Configure a volume for a Pod](#)
- [What's next](#)

## Before you begin

You need to have a Kubernetes cluster, and the `kubectl` command-line tool must be configured to communicate with your cluster. 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`.

## Configure a volume for a Pod

In this exercise, you create a Pod that runs one Container. This Pod has a Volume of type [emptyDir](#) that lasts for the life of the Pod, even if the Container terminates and restarts. Here is the configuration file for the Pod:

## [pods/storage/redis.yaml](#)

```
apiVersion: v1
kind: Pod
metadata:
  name: redis
spec:
  containers:
    - name: redis
      image: redis
      volumeMounts:
        - name: redis-storage
          mountPath: /data/redis
  volumes:
    - name: redis-storage
      emptyDir: {}
```

1. Create the Pod:

```
kubectl apply -f https://k8s.io/examples/pods/storage/
redis.yaml
```

2. Verify that the Pod's Container is running, and then watch for changes to the Pod:

```
kubectl get pod redis --watch
```

The output looks like this:

NAME	READY	STATUS	RESTARTS	AGE
redis	1/1	Running	0	13s

3. In another terminal, get a shell to the running Container:

```
kubectl exec -it redis -- /bin/bash
```

4. In your shell, go to /data/redis, and then create a file:

```
root@redis:/data# cd /data/redis/
root@redis:/data/redis# echo Hello > test-file
```

5. In your shell, list the running processes:

```
root@redis:/data/redis# apt-get update
root@redis:/data/redis# apt-get install procps
root@redis:/data/redis# ps aux
```

The output is similar to this:

USER	PID	%CPU	%MEM	VSZ	RSS	TTY	STAT	START
TIME	COMMAND							
	redis	1	0.1	0.1	33308	3828	?	Ssl 00:46

```
0:00 redis-server *:6379
root      12  0.0  0.0  20228  3020 ?          Ss  00:47
0:00 /bin/bash
root      15  0.0  0.0  17500  2072 ?          R+  00:48
0:00 ps aux
```

6. In your shell, kill the Redis process:

```
root@redis:/data/redis# kill <pid>
```

where `<pid>` is the Redis process ID (PID).

7. In your original terminal, watch for changes to the Redis Pod. Eventually, you will see something like this:

NAME	READY	STATUS	RESTARTS	AGE
redis	1/1	Running	0	13s
redis	0/1	Completed	0	6m
redis	1/1	Running	1	6m

At this point, the Container has terminated and restarted. This is because the Redis Pod has a [restartPolicy](#) of Always.

1. Get a shell into the restarted Container:

```
kubectl exec -it redis -- /bin/bash
```

2. In your shell, go to `/data/redis`, and verify that `test-file` is still there.

```
root@redis:/data/redis# cd /data/redis/
root@redis:/data/redis# ls
test-file
```

3. Delete the Pod that you created for this exercise:

```
kubectl delete pod redis
```

## What's next

- See [Volume](#).
- See [Pod](#).
- In addition to the local disk storage provided by `emptyDir`, Kubernetes supports many different network-attached storage solutions, including PD on GCE and EBS on EC2, which are preferred for critical data and will handle details such as mounting and unmounting the devices on the nodes. See [Volumes](#) for more details.

# Feedback

Was this page helpful?

Yes No

Thanks for the feedback. If you have a specific, answerable question about how to use Kubernetes, ask it on [Stack Overflow](#). Open an issue in the GitHub repo if you want to [report a problem](#) or [suggest an improvement](#).

---

[Create an Issue](#) [Edit This Page](#)

Page last modified on August 19, 2019 at 9:56 AM PST by [fixed a typo \(#15918\)](#) ([Page History](#))

[Edit This Page](#)

# Configure a Pod to Use a PersistentVolume for Storage

This page shows you how to configure a Pod to use a [PersistentVolumeClaim](#) [storage resources defined in a PersistentVolume so that it can be mounted as a volume in a container](#), for storage. Here is a summary of the process:

1. You, as cluster administrator, create a PersistentVolume backed by physical storage. You do not associate the volume with any Pod.
2. You, now taking the role of a developer / cluster user, create a PersistentVolumeClaim that is automatically bound to a suitable PersistentVolume.
3. You create a Pod that uses the above PersistentVolumeClaim for storage.
  - [Before you begin](#)
  - [Create an index.html file on your Node](#)
  - [Create a PersistentVolume](#)
  - [Create a PersistentVolumeClaim](#)
  - [Create a Pod](#)
  - [Clean up](#)
  - [Access control](#)
  - [What's next](#)

## Before you begin

- You need to have a Kubernetes cluster that has only one Node, and the [kubectl command line tool for communicating with a Kubernetes API server](#). command-line tool must be configured to communicate with your cluster. If you do not already have a single-node cluster, you can create one by using [Minikube](#).
- Familiarize yourself with the material in [Persistent Volumes](#).

## Create an index.html file on your Node

Open a shell to the single Node in your cluster. How you open a shell depends on how you set up your cluster. For example, if you are using Minikube, you can open a shell to your Node by entering `minikube ssh`.

In your shell on that Node, create a `/mnt/data` directory:

```
# This assumes that your Node uses "sudo" to run commands
# as the superuser
sudo mkdir /mnt/data
```

In the /mnt/data directory, create an index.html file:

```
# This again assumes that your Node uses "sudo" to run commands
# as the superuser
sudo sh -c "echo 'Hello from Kubernetes storage' > /mnt/data/
index.html"
```

**Note:** If your Node uses a tool for superuser access other than sudo, you can usually make this work if you replace sudo with the name of the other tool.

Test that the index.html file exists:

```
cat /mnt/data/index.html
```

The output should be:

```
Hello from Kubernetes storage
```

You can now close the shell to your Node.

## Create a PersistentVolume

In this exercise, you create a *hostPath* PersistentVolume. Kubernetes supports hostPath for development and testing on a single-node cluster. A hostPath PersistentVolume uses a file or directory on the Node to emulate network-attached storage.

In a production cluster, you would not use hostPath. Instead a cluster administrator would provision a network resource like a Google Compute Engine persistent disk, an NFS share, or an Amazon Elastic Block Store volume. Cluster administrators can also use [StorageClasses](#) to set up [dynamic provisioning](#).

Here is the configuration file for the hostPath PersistentVolume:

### [pods/storage/pv-volume.yaml](#)

```
apiVersion: v1
kind: PersistentVolume
metadata:
  name: task-pv-volume
  labels:
    type: local
spec:
  storageClassName: manual
  capacity:
    storage: 10Gi
  accessModes:
    - ReadWriteOnce
  hostPath:
    path: "/mnt/data"
```

The configuration file specifies that the volume is at `/mnt/data` on the cluster's Node. The configuration also specifies a size of 10 gibibytes and an access mode of `ReadWriteOnce`, which means the volume can be mounted as read-write by a single Node. It defines the [StorageClass name](#) `manual` for the `PersistentVolume`, which will be used to bind `PersistentVolumeClaim` requests to this `PersistentVolume`.

Create the `PersistentVolume`:

```
kubectl apply -f https://k8s.io/examples/pods/storage/pv-volume.yaml
```

View information about the `PersistentVolume`:

```
kubectl get pv task-pv-volume
```

The output shows that the `PersistentVolume` has a `STATUS` of `Available`. This means it has not yet been bound to a `PersistentVolumeClaim`.

NAME	CAPACITY	ACCESSMODES	RECLAIMPOLICY	
STATUS	CLAIM	STORAGECLASS	REASON	AGE
task-pv-volume	10Gi	RW0	Retain	
Available		manual		4s

## Create a `PersistentVolumeClaim`

The next step is to create a `PersistentVolumeClaim`. Pods use `PersistentVolumeClaims` to request physical storage. In this exercise, you create a `PersistentVolumeClaim` that requests a volume of at least three gibibytes that can provide read-write access for at least one Node.

Here is the configuration file for the `PersistentVolumeClaim`:

### [pods/storage/pv-claim.yaml](#)

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: task-pv-claim
spec:
  storageClassName: manual
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: 3Gi
```

Create the PersistentVolumeClaim:

```
kubectl apply -f https://k8s.io/examples/pods/storage/pv-claim.yaml
```

After you create the PersistentVolumeClaim, the Kubernetes control plane looks for a PersistentVolume that satisfies the claim's requirements. If the control plane finds a suitable PersistentVolume with the same StorageClass, it binds the claim to the volume.

Look again at the PersistentVolume:

```
kubectl get pv task-pv-volume
```

Now the output shows a STATUS of Bound.

NAME	CAPACITY	ACCESSMODES	RECLAIMPOLICY	AGE
STATUS	CLAIM		STORAGECLASS	REASON
task-pv-volume	10Gi	RW0		Retain
Bound	default/task-pv-claim		manual	2m

Look at the PersistentVolumeClaim:

```
kubectl get pvc task-pv-claim
```

The output shows that the PersistentVolumeClaim is bound to your PersistentVolume, task-pv-volume.

NAME	STATUS	VOLUME	CAPACITY
ACCESSMODES	STORAGECLASS	AGE	
task-pv-claim	Bound	task-pv-volume	10Gi
RW0	manual	30s	

## Create a Pod

The next step is to create a Pod that uses your PersistentVolumeClaim as a volume.

Here is the configuration file for the Pod:

[pods/storage/pv-pod.yaml](#)

```
apiVersion: v1
kind: Pod
metadata:
  name: task-pv-pod
spec:
  volumes:
    - name: task-pv-storage
      persistentVolumeClaim:
        claimName: task-pv-claim
  containers:
    - name: task-pv-container
      image: nginx
      ports:
        - containerPort: 80
          name: "http-server"
      volumeMounts:
        - mountPath: "/usr/share/nginx/html"
          name: task-pv-storage
```

Notice that the Pod's configuration file specifies a PersistentVolumeClaim, but it does not specify a PersistentVolume. From the Pod's point of view, the claim is a volume.

Create the Pod:

```
kubectl apply -f https://k8s.io/examples/pods/storage/pv-pod.yaml
```

Verify that the container in the Pod is running;

```
kubectl get pod task-pv-pod
```

Get a shell to the container running in your Pod:

```
kubectl exec -it task-pv-pod -- /bin/bash
```

In your shell, verify that nginx is serving the `index.html` file from the `hostPath` volume:

```
# Be sure to run these 3 commands inside the root shell that
comes from
# running "kubectl exec" in the previous step
apt update
apt install curl
curl http://localhost/
```

The output shows the text that you wrote to the `index.html` file on the `hostPath` volume:

```
Hello from Kubernetes storage
```

If you see that message, you have successfully configured a Pod to use storage from a `PersistentVolumeClaim`.

## Clean up

Delete the Pod, the `PersistentVolumeClaim` and the `PersistentVolume`:

```
kubectl delete pod task-pv-pod
kubectl delete pvc task-pv-claim
kubectl delete pv task-pv-volume
```

If you don't already have a shell open to the Node in your cluster, open a new shell the same way that you did earlier.

In the shell on your Node, remove the file and directory that you created:

```
# This assumes that your Node uses "sudo" to run commands
# as the superuser
sudo rm /mnt/data/index.html
sudo rmdir /mnt/data
```

You can now close the shell to your Node.

## Access control

Storage configured with a group ID (GID) allows writing only by Pods using the same GID. Mismatched or missing GIDs cause permission denied errors. To reduce the need for coordination with users, an administrator can annotate a `PersistentVolume` with a GID. Then the GID is automatically added to any Pod that uses the `PersistentVolume`.

Use the `pv.beta.kubernetes.io/gid` annotation as follows:

```
apiVersion: v1
kind: PersistentVolume
metadata:
  name: pv1
  annotations:
    pv.beta.kubernetes.io/gid: "1234"
```

When a Pod consumes a `PersistentVolume` that has a GID annotation, the annotated GID is applied to all containers in the Pod in the same way that GIDs specified in the Pod's security context are. Every GID, whether it originates from a `PersistentVolume` annotation or the Pod's specification, is applied to the first process run in each container.

**Note:** When a Pod consumes a PersistentVolume, the GIDs associated with the PersistentVolume are not present on the Pod resource itself.

## What's next

- Learn more about [PersistentVolumes](#).
- Read the [Persistent Storage design document](#).

## Reference

- [PersistentVolume](#)
- [PersistentVolumeSpec](#)
- [PersistentVolumeClaim](#)
- [PersistentVolumeClaimSpec](#)

## Feedback

Was this page helpful?

Yes No

Thanks for the feedback. If you have a specific, answerable question about how to use Kubernetes, ask it on [Stack Overflow](#). Open an issue in the GitHub repo if you want to [report a problem](#) or [suggest an improvement](#).

---

[Create an Issue](#) [Edit This Page](#)

Page last modified on December 20, 2019 at 10:49 PM PST by [Fix en language misspell \(#18201\)](#) ([Page History](#))

[Edit This Page](#)

# Configure a Pod to Use a Projected Volume for Storage

This page shows how to use a [projected](#) Volume to mount several existing volume sources into the same directory. Currently, secret, configMap, downwardAPI, and serviceAccountToken volumes can be projected.

**Note:** serviceAccountToken is not a volume type.

- [Before you begin](#)
- [Configure a projected volume for a pod](#)
- [Clean up](#)
- [What's next](#)

## Before you begin

You need to have a Kubernetes cluster, and the kubectl command-line tool must be configured to communicate with your cluster. 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`.

## Configure a projected volume for a pod

In this exercise, you create username and password [Secrets Stores sensitive information, such as passwords, OAuth tokens, and ssh keys](#). from local files. You then create a Pod that runs one container, using a [projected](#) Volume to mount the Secrets into the same shared directory.

Here is the configuration file for the Pod:

## [pods/storage/projected.yaml](#)

```
apiVersion: v1
kind: Pod
metadata:
  name: test-projected-volume
spec:
  containers:
    - name: test-projected-volume
      image: busybox
      args:
        - sleep
        - "86400"
      volumeMounts:
        - name: all-in-one
          mountPath: "/projected-volume"
          readOnly: true
  volumes:
    - name: all-in-one
      projected:
        sources:
          - secret:
              name: user
          - secret:
              name: pass
```

### 1. Create the Secrets:

```
# Create files containing the username and password:
echo -n "admin" > ./username.txt
echo -n "1f2d1e2e67df" > ./password.txt

# Package these files into secrets:
kubectl create secret generic user --from-file=./username.txt
kubectl create secret generic pass --from-file=./password.txt
```

### 2. Create the Pod:

```
kubectl apply -f https://k8s.io/examples/pods/storage/
projected.yaml
```

### 3. Verify that the Pod's container is running, and then watch for changes to the Pod:

```
kubectl get --watch pod test-projected-volume
```

The output looks like this:

NAME	READY	STATUS	RESTARTS	AGE
test-projected-volume	1/1	Running	0	14s

4. In another terminal, get a shell to the running container:

```
kubectl exec -it test-projected-volume -- /bin/sh
```

5. In your shell, verify that the `projected-volume` directory contains your projected sources:

```
ls /projected-volume/
```

## Clean up

Delete the Pod and the Secrets:

```
kubectl delete pod test-projected-volume  
kubectl delete secret user pass
```

## What's next

- Learn more about [projected](#) volumes.
- Read the [all-in-one volume](#) design document.

## Feedback

Was this page helpful?

Yes No

Thanks for the feedback. If you have a specific, answerable question about how to use Kubernetes, ask it on [Stack Overflow](#). Open an issue in the GitHub repo if you want to [report a problem](#) or [suggest an improvement](#).

---

[Create an Issue](#) [Edit This Page](#)

Page last modified on May 29, 2019 at 7:20 AM PST by [Delete resources created during this task \(#14536\)](#) ([Page History](#))

[Edit This Page](#)

## Configure a Security Context for a Pod or Container

A security context defines privilege and access control settings for a Pod or Container. Security context settings include:

- Discretionary Access Control: Permission to access an object, like a file, is based on [user ID \(UID\) and group ID \(GID\)](#).

- [Security Enhanced Linux \(SELinux\)](#): Objects are assigned security labels.
- Running as privileged or unprivileged.
- [Linux Capabilities](#): Give a process some privileges, but not all the privileges of the root user.
- [AppArmor](#): Use program profiles to restrict the capabilities of individual programs.
- [Seccomp](#): Filter a process's system calls.
- AllowPrivilegeEscalation: Controls whether a process can gain more privileges than its parent process. This bool directly controls whether the [no\\_new\\_privs](#) flag gets set on the container process.  
AllowPrivilegeEscalation is true always when the container is: 1) run as Privileged OR 2) has CAP\_SYS\_ADMIN.

For more information about security mechanisms in Linux, see [Overview of Linux Kernel Security Features](#)

- [Before you begin](#)
- [Set the security context for a Pod](#)
- [Set the security context for a Container](#)
- [Set capabilities for a Container](#)
- [Assign SELinux labels to a Container](#)
- [Discussion](#)
- [Clean up](#)
- [What's next](#)

## Before you begin

You need to have a Kubernetes cluster, and the kubectl command-line tool must be configured to communicate with your cluster. 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`.

## Set the security context for a Pod

To specify security settings for a Pod, include the `securityContext` field in the Pod specification. The `securityContext` field is a [PodSecurityContext](#) object. The security settings that you specify for a Pod apply to all Containers in the Pod. Here is a configuration file for a Pod that has a `securityContext` and an `emptyDir` volume:

### [pods/security/security-context.yaml](#)

```
apiVersion: v1
kind: Pod
metadata:
  name: security-context-demo
spec:
  securityContext:
    runAsUser: 1000
    runAsGroup: 3000
    fsGroup: 2000
  volumes:
    - name: sec-ctx-vol
      emptyDir: {}
  containers:
    - name: sec-ctx-demo
      image: busybox
      command: [ "sh", "-c", "sleep 1h" ]
      volumeMounts:
        - name: sec-ctx-vol
          mountPath: /data/demo
      securityContext:
        allowPrivilegeEscalation: false
```

In the configuration file, the `runAsUser` field specifies that for any Containers in the Pod, all processes run with user ID 1000. The `runAsGroup` field specifies the primary group ID of 3000 for all processes within any containers of the Pod. If this field is omitted, the primary group ID of the containers will be root(0). Any files created will also be owned by user 1000 and group 3000 when `runAsGroup` is specified. Since `fsGroup` field is specified, all processes of the container are also part of the supplementary group ID 2000. The owner for volume `/data/demo` and any files created in that volume will be Group ID 2000.

Create the Pod:

```
kubectl apply -f https://k8s.io/examples/pods/security/security-
context.yaml
```

Verify that the Pod's Container is running:

```
kubectl get pod security-context-demo
```

Get a shell to the running Container:

```
kubectl exec -it security-context-demo -- sh
```

In your shell, list the running processes:

```
ps
```

The output shows that the processes are running as user 1000, which is the value of `runAsUser`:

```
PID    USER      TIME  COMMAND
 1  1000      0:00  sleep 1h
 6  1000      0:00  sh
...
```

In your shell, navigate to `/data`, and list the one directory:

```
cd /data
ls -l
```

The output shows that the `/data/demo` directory has group ID 2000, which is the value of `fsGroup`.

```
drwxrwsrwx 2 root 2000 4096 Jun  6 20:08 demo
```

In your shell, navigate to `/data/demo`, and create a file:

```
cd demo
echo hello > testfile
```

List the file in the `/data/demo` directory:

```
ls -l
```

The output shows that `testfile` has group ID 2000, which is the value of `fsGroup`.

```
-rw-r--r-- 1 1000 2000 6 Jun  6 20:08 testfile
```

Run the following command:

```
$ id
uid=1000 gid=3000 groups=2000
```

You will see that `gid` is 3000 which is same as `runAsGroup` field. If the `runAsGroup` was omitted the `gid` would remain as 0(root) and the process will be able to interact with files that are owned by root(0) group and that have the required group permissions for root(0) group.

Exit your shell:

```
exit
```

## Set the security context for a Container

To specify security settings for a Container, include the `securityContext` field in the Container manifest. The `securityContext` field is a [SecurityContext](#) object. Security settings that you specify for a Container apply only to the individual Container, and they override settings made at

the Pod level when there is overlap. Container settings do not affect the Pod's Volumes.

Here is the configuration file for a Pod that has one Container. Both the Pod and the Container have a `securityContext` field:

#### [`pods/security/security-context-2.yaml`](#)

```
apiVersion: v1
kind: Pod
metadata:
  name: security-context-demo-2
spec:
  securityContext:
    runAsUser: 1000
  containers:
  - name: sec-ctx-demo-2
    image: gcr.io/google-samples/node-hello:1.0
    securityContext:
      runAsUser: 2000
      allowPrivilegeEscalation: false
```

Create the Pod:

```
kubectl apply -f https://k8s.io/examples/pods/security/security-
context-2.yaml
```

Verify that the Pod's Container is running:

```
kubectl get pod security-context-demo-2
```

Get a shell into the running Container:

```
kubectl exec -it security-context-demo-2 -- sh
```

In your shell, list the running processes:

```
ps aux
```

The output shows that the processes are running as user 2000. This is the value of `runAsUser` specified for the Container. It overrides the value 1000 that is specified for the Pod.

USER	PID	%CPU	%MEM	VSZ	RSS	TTY	STAT	START	TIME
COMMAND									
2000	1	0.0	0.0	4336	764	?	Ss	20:36	
0:00	/bin/sh	-c	node server.js						
2000	8	0.1	0.5	772124	22604	?	S1	20:36	0:00
node	server.js								
...									

Exit your shell:

```
exit
```

## Set capabilities for a Container

With [Linux capabilities](#), you can grant certain privileges to a process without granting all the privileges of the root user. To add or remove Linux capabilities for a Container, include the `capabilities` field in the `securityContext` section of the Container manifest.

First, see what happens when you don't include a `capabilities` field. Here is configuration file that does not add or remove any Container capabilities:

### [pods/security/security-context-3.yaml](#)

```
apiVersion: v1
kind: Pod
metadata:
  name: security-context-demo-3
spec:
  containers:
    - name: sec-ctx-3
      image: gcr.io/google-samples/node-hello:1.0
```

Create the Pod:

```
kubectl apply -f https://k8s.io/examples/pods/security/security-
context-3.yaml
```

Verify that the Pod's Container is running:

```
kubectl get pod security-context-demo-3
```

Get a shell into the running Container:

```
kubectl exec -it security-context-demo-3 -- sh
```

In your shell, list the running processes:

```
ps aux
```

The output shows the process IDs (PIDs) for the Container:

USER	PID	%CPU	%MEM	VSZ	RSS	TTY	STAT	START	TIME	COMMAND
root	1	0.0	0.0	4336	796	?	Ss	18:17	0:00	/bin/sh
				-c node server.js						
root	5	0.1	0.5	772124	22700	?	Sl	18:17	0:00	node server.js

In your shell, view the status for process 1:

```
cd /proc/1  
cat status
```

The output shows the capabilities bitmap for the process:

```
...  
CapPrm: 00000000a80425fb  
CapEff: 00000000a80425fb  
...
```

Make a note of the capabilities bitmap, and then exit your shell:

```
exit
```

Next, run a Container that is the same as the preceding container, except that it has additional capabilities set.

Here is the configuration file for a Pod that runs one Container. The configuration adds the CAP\_NET\_ADMIN and CAP\_SYS\_TIME capabilities:

```
pods/security/security-context-4.yaml  
  
apiVersion: v1  
kind: Pod  
metadata:  
  name: security-context-demo-4  
spec:  
  containers:  
    - name: sec-ctx-4  
      image: gcr.io/google-samples/node-hello:1.0  
      securityContext:  
        capabilities:  
          add: ["NET_ADMIN", "SYS_TIME"]
```

Create the Pod:

```
kubectl apply -f https://k8s.io/examples/pods/security/security-  
context-4.yaml
```

Get a shell into the running Container:

```
kubectl exec -it security-context-demo-4 -- sh
```

In your shell, view the capabilities for process 1:

```
cd /proc/1  
cat status
```

The output shows capabilities bitmap for the process:

```
...
CapPrm: 00000000aa0435fb
CapEff: 00000000aa0435fb
...
```

Compare the capabilities of the two Containers:

```
00000000a80425fb
00000000aa0435fb
```

In the capability bitmap of the first container, bits 12 and 25 are clear. In the second container, bits 12 and 25 are set. Bit 12 is CAP\_NET\_ADMIN, and bit 25 is CAP\_SYS\_TIME. See [capability.h](#) for definitions of the capability constants.

**Note:** Linux capability constants have the form CAP\_XXX. But when you list capabilities in your Container manifest, you must omit the CAP\_ portion of the constant. For example, to add CAP\_SYS\_TIME, include SYS\_TIME in your list of capabilities.

## Assign SELinux labels to a Container

To assign SELinux labels to a Container, include the `seLinuxOptions` field in the `securityContext` section of your Pod or Container manifest. The `seLinuxOptions` field is an [SELinuxOptions](#) object. Here's an example that applies an SELinux level:

```
...
securityContext:
  seLinuxOptions:
    level: "s0:c123,c456"
```

**Note:** To assign SELinux labels, the SELinux security module must be loaded on the host operating system.

## Discussion

The security context for a Pod applies to the Pod's Containers and also to the Pod's Volumes when applicable. Specifically `fsGroup` and `seLinuxOptions` are applied to Volumes as follows:

- `fsGroup`: Volumes that support ownership management are modified to be owned and writable by the GID specified in `fsGroup`. See the [Ownership Management design document](#) for more details.
- `seLinuxOptions`: Volumes that support SELinux labeling are relabeled to be accessible by the label specified under `seLinuxOptions`. Usually you only need to set the `level` section. This sets the [Multi-Category Security \(MCS\)](#) label given to all Containers in the Pod as well as the Volumes.

**Warning:** After you specify an MCS label for a Pod, all Pods with the same label can access the Volume. If you need inter-Pod protection, you must assign a unique MCS label to each Pod.

## Clean up

Delete the Pod:

```
kubectl delete pod security-context-demo  
kubectl delete pod security-context-demo-2  
kubectl delete pod security-context-demo-3  
kubectl delete pod security-context-demo-4
```

## What's next

- [PodSecurityContext](#)
- [SecurityContext](#)
- [Tuning Docker with the newest security enhancements](#)
- [Security Contexts design document](#)
- [Ownership Management design document](#)
- [Pod Security Policies](#)
- [AllowPrivilegeEscalation design document](#)

## Feedback

Was this page helpful?

Yes No

Thanks for the feedback. If you have a specific, answerable question about how to use Kubernetes, ask it on [Stack Overflow](#). Open an issue in the GitHub repo if you want to [report a problem](#) or [suggest an improvement](#).

---

[Create an Issue](#) [Edit This Page](#)

Page last modified on September 26, 2019 at 6:15 AM PST by [clean up the environment \(#16430\)](#) ([Page History](#))

[Edit This Page](#)

# Configure Service Accounts for Pods

A service account provides an identity for processes that run in a Pod.

**Note:** This document is a user introduction to Service Accounts and describes how service accounts behave in a cluster set up as recommended by the Kubernetes project. Your cluster

administrator may have customized the behavior in your cluster, in which case this documentation may not apply.

When you (a human) access the cluster (for example, using `kubectl`), you are authenticated by the apiserver as a particular User Account (currently this is usually `admin`, unless your cluster administrator has customized your cluster). Processes in containers inside pods can also contact the apiserver. When they do, they are authenticated as a particular Service Account (for example, `default`).

- [Before you begin](#)
- [Use the Default Service Account to access the API server.](#)
- [Use Multiple Service Accounts.](#)
- [Manually create a service account API token.](#)
- [Add ImagePullSecrets to a service account](#)
- [Service Account Token Volume Projection](#)
- [What's next](#)

## Before you begin

You need to have a Kubernetes cluster, and the `kubectl` command-line tool must be configured to communicate with your cluster. 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`.

## Use the Default Service Account to access the API server.

When you create a pod, if you do not specify a service account, it is automatically assigned the `default` service account in the same namespace. If you get the raw json or yaml for a pod you have created (for example, `kubectl get pods/<podname> -o yaml`), you can see the `spec.serviceAccountName` field has been [automatically set](#).

You can access the API from inside a pod using automatically mounted service account credentials, as described in [Accessing the Cluster](#). The API permissions of the service account depend on the [authorization plugin and policy](#) in use.

In version 1.6+, you can opt out of automounting API credentials for a service account by setting `automountServiceAccountToken: false` on the service account:

```
apiVersion: v1
kind: ServiceAccount
metadata:
```

```
  name: build-robot
automountServiceAccountToken: false
...
```

In version 1.6+, you can also opt out of automounting API credentials for a particular pod:

```
apiVersion: v1
kind: Pod
metadata:
  name: my-pod
spec:
  serviceAccountName: build-robot
  automountServiceAccountToken: false
...
```

The pod spec takes precedence over the service account if both specify a `automountServiceAccountToken` value.

## Use Multiple Service Accounts.

Every namespace has a default service account resource called `default`. You can list this and any other serviceAccount resources in the namespace with this command:

```
kubectl get serviceaccounts
```

The output is similar to this:

NAME	SECRETS	AGE
default	1	1d

You can create additional ServiceAccount objects like this:

```
kubectl apply -f - <<EOF
apiVersion: v1
kind: ServiceAccount
metadata:
  name: build-robot
EOF
```

If you get a complete dump of the service account object, like this:

```
kubectl get serviceaccounts/build-robot -o yaml
```

The output is similar to this:

```
apiVersion: v1
kind: ServiceAccount
metadata:
  creationTimestamp: 2015-06-16T00:12:59Z
  name: build-robot
  namespace: default
```

```
resourceVersion: "272500"
uid: 721ab723-13bc-11e5-aec2-42010af0021e
secrets:
- name: build-robot-token-bvbk5
```

then you will see that a token has automatically been created and is referenced by the service account.

You may use authorization plugins to [set permissions on service accounts](#).

To use a non-default service account, simply set the `spec.serviceAccountName` field of a pod to the name of the service account you wish to use.

The service account has to exist at the time the pod is created, or it will be rejected.

You cannot update the service account of an already created pod.

You can clean up the service account from this example like this:

```
kubectl delete serviceaccount/build-robot
```

## Manually create a service account API token.

Suppose we have an existing service account named "build-robot" as mentioned above, and we create a new secret manually.

```
kubectl apply -f - <<EOF
apiVersion: v1
kind: Secret
metadata:
  name: build-robot-secret
  annotations:
    kubernetes.io/service-account.name: build-robot
type: kubernetes.io/service-account-token
EOF
```

Now you can confirm that the newly built secret is populated with an API token for the "build-robot" service account.

Any tokens for non-existent service accounts will be cleaned up by the token controller.

```
kubectl describe secrets/build-robot-secret
```

The output is similar to this:

```
Name:          build-robot-secret
Namespace:     default
Labels:        <none>
Annotations:   kubernetes.io/service-account.name=build-robot
               kubernetes.io/service-
account.uid=da68f9c6-9d26-11e7-b84e-002dc52800da
```

```
Type: kubernetes.io/service-account-token

Data
=====
ca.crt:       1338 bytes
namespace:    7 bytes
token:        ...
```

**Note:** The content of token is elided here.

## Add ImagePullSecrets to a service account

First, create an imagePullSecret, as described [here](#). Next, verify it has been created. For example:

```
kubectl get secrets myregistrykey
```

The output is similar to this:

NAME	TYPE	DATA	AGE
myregistrykey	kubernetes.io/.dockerconfigjson	1	1d

Next, modify the default service account for the namespace to use this secret as an imagePullSecret.

```
kubectl patch serviceaccount default -p '{"imagePullSecrets": [{"name": "myregistrykey"}]}'
```

Interactive version requires manual edit:

```
kubectl get serviceaccounts default -o yaml > ./sa.yaml
```

The output of the `sa.yaml` file is similar to this:

```
apiVersion: v1
kind: ServiceAccount
metadata:
  creationTimestamp: 2015-08-07T22:02:39Z
  name: default
  namespace: default
  resourceVersion: "243024"
  uid: 052fb0f4-3d50-11e5-b066-42010af0d7b6
secrets:
- name: default-token-uudge
```

Using your editor of choice (for example `vi`), open the `sa.yaml` file, delete line with key `resourceVersion`, add lines with `imagePullSecrets:` and save.

The output of the `sa.yaml` file is similar to this:

```
apiVersion: v1
kind: ServiceAccount
metadata:
  creationTimestamp: 2015-08-07T22:02:39Z
  name: default
  namespace: default
  uid: 052fb0f4-3d50-11e5-b066-42010af0d7b6
secrets:
- name: default-token-uudge
imagePullSecrets:
- name: myregistrykey
```

Finally replace the serviceaccount with the new updated `sa.yaml` file

```
kubectl replace serviceaccount default -f ./sa.yaml
```

Now, any new pods created in the current namespace will have this added to their spec:

```
spec:
  imagePullSecrets:
    - name: myregistrykey
```

## Service Account Token Volume Projection

**FEATURE STATE:** Kubernetes v1.12 [beta](#)

This feature is currently in a *beta* state, meaning:

[Edit This Page](#)

## Pull an Image from a Private Registry

This page shows how to create a Pod that uses a Secret to pull an image from a private Docker registry or repository.

- [Before you begin](#)
- [Log in to Docker](#)
- [Create a Secret based on existing Docker credentials](#)
- [Create a Secret by providing credentials on the command line](#)
- [Inspecting the Secret regcred](#)
- [Create a Pod that uses your Secret](#)
- [What's next](#)

## Before you begin

- You need to have a Kubernetes cluster, and the `kubectl` command-line tool must be configured to communicate with your cluster. 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`.

- To do this exercise, you need a [Docker ID](#) and password.

## Log in to Docker

On your laptop, you must authenticate with a registry in order to pull a private image:

```
docker login
```

When prompted, enter your Docker username and password.

The login process creates or updates a `config.json` file that holds an authorization token.

View the `config.json` file:

```
cat ~/.docker/config.json
```

The output contains a section similar to this:

```
{
  "auths": {
    "https://index.docker.io/v1/": {
      "auth": "c3R...zE2"
    }
  }
}
```

**Note:** If you use a Docker credentials store, you won't see that `auth` entry but a `credsStore` entry with the name of the store as value.

## Create a Secret based on existing Docker credentials

A Kubernetes cluster uses the Secret of `docker-registry` type to authenticate with a container registry to pull a private image.

If you already ran `docker login`, you can copy that credential into Kubernetes:

```
kubectl create secret generic regcred \
  --from-file=.dockerconfigjson=<path/to/.docker/config.json> \
  --type=kubernetes.io/dockerconfigjson
```

If you need more control (for example, to set a namespace or a label on the new secret) then you can customise the Secret before storing it. Be sure to:

- set the name of the data item to `.dockerconfigjson`
  - base64 encode the docker file and paste that string, unbroken as the value for field `data[".dockerconfigjson"]`
  - set `type` to `kubernetes.io/dockerconfigjson`

## Example:

```
apiVersion: v1
kind: Secret
metadata:
  name: myregistrykey
  namespace: awesomeapps
data:
  .dockerconfigjson: UmVhbGx5IHJlYWxseSByZWVlZWVlZWVlZWFrYWFrYWFr
YWFhYWFrYWFrYWFrYWFrYWFrYWFrYWFrYWFrYWFrYWFrYWFrYWFrYWFrYWFr
Gx5eXl5eXl5eXl5eXl5eXl5eXl5eSBsbGxsBGSbGxsBGSbGxsBGSbGxsBGSbGxsB
Gx5eXl5eXl5eXl5eXl5eXl5eXl5eXl5eSBsbGxsBGSbGxsBGSbGxsBGSbGxsBGSbGxsB
9vb29vb29vb29vb29vb29vb25ubm5ubm5ubm5ubm5ubm5ubm5ubm5ubm5ubm5ubm5ubm
nZ2dnZ2dnZ2dnZ2cgYXV0aCBrZXlzCg==
type: kubernetes.io/dockerconfigjson
```

If you get the error message `error: no objects passed to create`, it may mean the base64 encoded string is invalid. If you get an error message like `Secret "myregistrykey" is invalid: data[.dockerconfigjson]: invalid value ...`, it means the base64 encoded string in the data was successfully decoded, but could not be parsed as a `.docker/config.json` file.

**Create a Secret by providing credentials on the command line**

Create this Secret, naming it reqcred:

```
kubectl create secret docker-registry regcred --docker-server=<your-registry-server> --docker-username=<your-name> --docker-password=<your-pword> --docker-email=<your-email>
```

where:

- <your-registry-server> is your Private Docker Registry FQDN.  
(<https://index.docker.io/v1/> for DockerHub)
  - <your-name> is your Docker username.
  - <your-pword> is your Docker password.
  - <your-email> is your Docker email.

You have successfully set your Docker credentials in the cluster as a Secret called `reqcred`.

**Note:** Typing secrets on the command line may store them in your shell history unprotected, and those secrets might also be visible to other users on your PC during the time that `kubectl` is running.

## Inspecting the Secret `regcred`

To understand the contents of the `regcred` Secret you just created, start by viewing the Secret in YAML format:

```
kubectl get secret regcred --output=yaml
```

The output is similar to this:

```
apiVersion: v1
kind: Secret
metadata:
  ...
  name: regcred
  ...
data:
  .dockerconfigjson: eyJodHRwczovL2luZGV4L ... J0QUl6RTIifX0=
type: kubernetes.io/dockerconfigjson
```

The value of the `.dockerconfigjson` field is a base64 representation of your Docker credentials.

To understand what is in the `.dockerconfigjson` field, convert the secret data to a readable format:

```
kubectl get secret regcred --output="jsonpath={.data.\.dockerconfigjson}" | base64 --decode
```

The output is similar to this:

```
{"auths": {"your.private.registry.example.com": {"username": "janedoe", "password": "xxxxxxxxxxxx", "email": "jdoe@example.com", "auth": "c3R...zE2"}}}
```

To understand what is in the `auth` field, convert the base64-encoded data to a readable format:

```
echo "c3R...zE2" | base64 --decode
```

The output, username and password concatenated with a `:`, is similar to this:

```
janedoe:xxxxxxxxxxxx
```

Notice that the Secret data contains the authorization token similar to your local `~/.docker/config.json` file.

You have successfully set your Docker credentials as a Secret called `regcred` in the cluster.

# Create a Pod that uses your Secret

Here is a configuration file for a Pod that needs access to your Docker credentials in `regcred`:

## [pods/private-reg-pod.yaml](#)

```
apiVersion: v1
kind: Pod
metadata:
  name: private-reg
spec:
  containers:
    - name: private-reg-container
      image: <your-private-image>
  imagePullSecrets:
    - name: regcred
```

Download the above file:

```
wget -O my-private-reg-pod.yaml https://k8s.io/examples/pods/
private-reg-pod.yaml
```

In file `my-private-reg-pod.yaml`, replace `<your-private-image>` with the path to an image in a private registry such as:

```
your.private.registry.example.com/janedoe/jdoe-private:v1
```

To pull the image from the private registry, Kubernetes needs credentials. The `imagePullSecrets` field in the configuration file specifies that Kubernetes should get the credentials from a Secret named `regcred`.

Create a Pod that uses your Secret, and verify that the Pod is running:

```
kubectl apply -f my-private-reg-pod.yaml
kubectl get pod private-reg
```

## What's next

- Learn more about [Secrets](#).
- Learn more about [using a private registry](#).
- Learn more about [adding image pull secrets to a service account](#).
- See [kubectl create secret docker-registry](#).
- See [Secret](#).
- See the `imagePullSecrets` field of [PodSpec](#).

# Feedback

Was this page helpful?

Yes No

Thanks for the feedback. If you have a specific, answerable question about how to use Kubernetes, ask it on [Stack Overflow](#). Open an issue in the GitHub repo if you want to [report a problem](#) or [suggest an improvement](#).

---

[Create an Issue](#) [Edit This Page](#)

Page last modified on June 06, 2019 at 6:18 AM PST by [changed example domain to be \\*\\*example.com\\*\\* \(#14740\)](#) ([Page History](#))

[Edit This Page](#)

# Configure Liveness, Readiness and Startup Probes

This page shows how to configure liveness, readiness and startup probes for Containers.

The [kubelet](#) uses liveness probes to know when to restart a Container. For example, liveness probes could catch a deadlock, where an application is running, but unable to make progress. Restarting a Container in such a state can help to make the application more available despite bugs.

The kubelet uses readiness probes to know when a Container is ready to start accepting traffic. A Pod is considered ready when all of its Containers are ready. One use of this signal is to control which Pods are used as backends for Services. When a Pod is not ready, it is removed from Service load balancers.

The kubelet uses startup probes to know when a Container application has started. If such a probe is configured, it disables liveness and readiness checks until it succeeds, making sure those probes don't interfere with the application startup. This can be used to adopt liveness checks on slow starting containers, avoiding them getting killed by the kubelet before they are up and running.

- [Before you begin](#)
- [Define a liveness command](#)
- [Define a liveness HTTP request](#)
- [Define a TCP liveness probe](#)
- [Use a named port](#)
- [Protect slow starting containers with startup probes](#)
- [Define readiness probes](#)
- [Configure Probes](#)
- [What's next](#)

## Before you begin

You need to have a Kubernetes cluster, and the kubectl command-line tool must be configured to communicate with your cluster. 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`.

## Define a liveness command

Many applications running for long periods of time eventually transition to broken states, and cannot recover except by being restarted. Kubernetes provides liveness probes to detect and remedy such situations.

In this exercise, you create a Pod that runs a Container based on the k8s.gcr.io/busybox image. Here is the configuration file for the Pod:

### [pods/probe/exec-liveness.yaml](#)

```
apiVersion: v1
kind: Pod
metadata:
  labels:
    test: liveness
    name: liveness-exec
spec:
  containers:
    - name: liveness
      image: k8s.gcr.io/busybox
      args:
        - /bin/sh
        - -c
        - touch /tmp/healthy; sleep 30; rm -rf /tmp/healthy; sleep 6
          00
      livenessProbe:
        exec:
          command:
            - cat
            - /tmp/healthy
        initialDelaySeconds: 5
        periodSeconds: 5
```

In the configuration file, you can see that the Pod has a single Container. The `periodSeconds` field specifies that the kubelet should perform a liveness probe every 5 seconds. The `initialDelaySeconds` field tells the kubelet that it should wait 5 second before performing the first probe. To perform a probe, the kubelet executes the command `cat /tmp/healthy` in the Container. If the command succeeds, it returns 0, and the kubelet considers the Container to be alive and healthy. If the command returns a non-zero value, the kubelet kills the Container and restarts it.

When the Container starts, it executes this command:

```
/bin/sh -c "touch /tmp/healthy; sleep 30; rm -rf /tmp/healthy;
sleep 600"
```

For the first 30 seconds of the Container's life, there is a /tmp/healthy file. So during the first 30 seconds, the command cat /tmp/healthy returns a success code. After 30 seconds, cat /tmp/healthy returns a failure code.

Create the Pod:

```
kubectl apply -f https://k8s.io/examples/pods/probe/exec-liveness.yaml
```

Within 30 seconds, view the Pod events:

```
kubectl describe pod liveness-exec
```

The output indicates that no liveness probes have failed yet:

FirstSeen SubobjectPath	LastSeen Type	Count	From Reason	Message
24s Normal	24s Scheduled	1	{default-scheduler } Successfully assigned liveness-exec to worker0	
23s Normal	23s Pulling	1	{kubelet worker0} spec.containers{livenes s}	image "k8s.gcr.io/busybox"
23s Normal	23s Pulled	1	{kubelet worker0} spec.containers{livenes s}	Successfully pulled image "k8s.gcr.i o/busybox"
23s Normal	23s Created	1	{kubelet worker0} spec.containers{livenes s}	Created container with docker id 86849c15382e; Security:[seccomp=unconfined]
23s Normal	23s Started	1	{kubelet worker0} spec.containers{livenes s}	Started container with docker id 86849c15382e

After 35 seconds, view the Pod events again:

```
kubectl describe pod liveness-exec
```

At the bottom of the output, there are messages indicating that the liveness probes have failed, and the containers have been killed and recreated.

FirstSeen SubobjectPath	LastSeen Type	Count	From Reason	Message
37s Normal	37s Scheduled	1	{default-scheduler } Successfully assigned liveness-exec to worker0	
36s Normal	36s Pulling	1	{kubelet worker0} spec.containers{livenes s}	image "k8s.gcr.io/busybox"
36s Normal	36s Pulled	1	{kubelet worker0} spec.containers{livenes s}	Successfully pulled image "k8s.gcr.i o/busybox"

```
36s      36s      1  {kubelet worker0}  spec.containers{liveness}
s}  Normal    Created    Created container with docker id
86849c15382e; Security:[seccomp=unconfined]
36s      36s      1  {kubelet worker0}  spec.containers{liveness}
s}  Normal    Started   Started container with docker id
86849c15382e
2s      2s      1  {kubelet worker0}  spec.containers{liveness}
s}  Warning   Unhealthy Liveness probe failed: cat: can't
open '/tmp/healthy': No such file or directory
```

Wait another 30 seconds, and verify that the Container has been restarted:

```
kubectl get pod liveness-exec
```

The output shows that RESTARTS has been incremented:

NAME	READY	STATUS	RESTARTS	AGE
liveness-exec	1/1	Running	1	1m

## Define a liveness HTTP request

Another kind of liveness probe uses an HTTP GET request. Here is the configuration file for a Pod that runs a container based on the k8s.gcr.io/liveness image.

<a href="#">pods/probe/http-liveness.yaml</a>
<pre>apiVersion: v1 kind: Pod metadata:   labels:     test: liveness     name: liveness-http spec:   containers:     - name: liveness       image: k8s.gcr.io/liveness       args:         - /server       livenessProbe:         httpGet:           path: /healthz           port: 8080           httpHeaders:             - name: Custom-Header               value: Awesome         initialDelaySeconds: 3         periodSeconds: 3</pre>

In the configuration file, you can see that the Pod has a single Container. The `periodSeconds` field specifies that the kubelet should perform a liveness probe every 3 seconds. The `initialDelaySeconds` field tells the kubelet that it should wait 3 seconds before performing the first probe. To perform a probe, the kubelet sends an HTTP GET request to the server that is running in the Container and listening on port 8080. If the handler for the server's `/healthz` path returns a success code, the kubelet considers the Container to be alive and healthy. If the handler returns a failure code, the kubelet kills the Container and restarts it.

Any code greater than or equal to 200 and less than 400 indicates success. Any other code indicates failure.

You can see the source code for the server in [server.go](#).

For the first 10 seconds that the Container is alive, the `/healthz` handler returns a status of 200. After that, the handler returns a status of 500.

```
http.HandleFunc("/healthz", func(w http.ResponseWriter, r *http.Request) {
    duration := time.Now().Sub(started)
    if duration.Seconds() > 10 {
        w.WriteHeader(500)
        w.Write([]byte(fmt.Sprintf("error: %v", duration.Seconds())))
    } else {
        w.WriteHeader(200)
        w.Write([]byte("ok"))
    }
})
```

The kubelet starts performing health checks 3 seconds after the Container starts. So the first couple of health checks will succeed. But after 10 seconds, the health checks will fail, and the kubelet will kill and restart the Container.

To try the HTTP liveness check, create a Pod:

```
kubectl apply -f https://k8s.io/examples/pods/probe/http-liveness.yaml
```

After 10 seconds, view Pod events to verify that liveness probes have failed and the Container has been restarted:

```
kubectl describe pod liveness-http
```

In releases prior to v1.13 (including v1.13), if the environment variable `http_proxy` (or `HTTP_PROXY`) is set on the node where a pod is running, the HTTP liveness probe uses that proxy. In releases after v1.13, local HTTP proxy environment variable settings do not affect the HTTP liveness probe.

## Define a TCP liveness probe

A third type of liveness probe uses a TCP Socket. With this configuration, the kubelet will attempt to open a socket to your container on the specified port. If it can establish a connection, the container is considered healthy, if it can't it is considered a failure.

### [pods/probe/tcp-liveness-readiness.yaml](#)

```
apiVersion: v1
kind: Pod
metadata:
  name: goproxy
  labels:
    app: goproxy
spec:
  containers:
    - name: goproxy
      image: k8s.gcr.io/goproxy:0.1
      ports:
        - containerPort: 8080
      readinessProbe:
        tcpSocket:
          port: 8080
        initialDelaySeconds: 5
        periodSeconds: 10
      livenessProbe:
        tcpSocket:
          port: 8080
        initialDelaySeconds: 15
        periodSeconds: 20
```

As you can see, configuration for a TCP check is quite similar to an HTTP check. This example uses both readiness and liveness probes. The kubelet will send the first readiness probe 5 seconds after the container starts. This will attempt to connect to the goproxy container on port 8080. If the probe succeeds, the pod will be marked as ready. The kubelet will continue to run this check every 10 seconds.

In addition to the readiness probe, this configuration includes a liveness probe. The kubelet will run the first liveness probe 15 seconds after the container starts. Just like the readiness probe, this will attempt to connect to the goproxy container on port 8080. If the liveness probe fails, the container will be restarted.

To try the TCP liveness check, create a Pod:

```
kubectl apply -f https://k8s.io/examples/pods/probe/tcp-liveness-readiness.yaml
```

After 15 seconds, view Pod events to verify that liveness probes:

```
kubectl describe pod goproxy
```

## Use a named port

You can use a named [ContainerPort](#) for HTTP or TCP liveness checks:

```
ports:
- name: liveness-port
  containerPort: 8080
  hostPort: 8080

livenessProbe:
  httpGet:
    path: /healthz
    port: liveness-port
```

## Protect slow starting containers with startup probes

Sometimes, you have to deal with legacy applications that might require an additional startup time on their first initialization. In such cases, it can be tricky to set up liveness probe parameters without compromising the fast response to deadlocks that motivated such a probe. The trick is to set up a startup probe with the same command, HTTP or TCP check, with a `failureThreshold * periodSeconds` long enough to cover the worse case startup time.

So, the previous example would become:

```
ports:
- name: liveness-port
  containerPort: 8080
  hostPort: 8080

livenessProbe:
  httpGet:
    path: /healthz
    port: liveness-port
  failureThreshold: 1
  periodSeconds: 10

startupProbe:
  httpGet:
    path: /healthz
    port: liveness-port
  failureThreshold: 30
  periodSeconds: 10
```

Thanks to the startup probe, the application will have a maximum of 5 minutes ( $30 * 10 = 300$ s) to finish its startup. Once the startup probe has succeeded once, the liveness probe takes over to provide a fast response to container deadlocks. If the startup probe never succeeds, the container is killed after 300s and subject to the pod's `restartPolicy`.

## Define readiness probes

Sometimes, applications are temporarily unable to serve traffic. For example, an application might need to load large data or configuration files during startup, or depend on external services after startup. In such cases, you don't want to kill the application, but you don't want to send it requests either. Kubernetes provides readiness probes to detect and mitigate these situations. A pod with containers reporting that they are not ready does not receive traffic through Kubernetes Services.

**Note:** Readiness probes runs on the container during its whole lifecycle.

Readiness probes are configured similarly to liveness probes. The only difference is that you use the `readinessProbe` field instead of the `livenessProbe` field.

```
readinessProbe:  
  exec:  
    command:  
      - cat  
      - /tmp/healthy  
  initialDelaySeconds: 5  
  periodSeconds: 5
```

Configuration for HTTP and TCP readiness probes also remains identical to liveness probes.

Readiness and liveness probes can be used in parallel for the same container. Using both can ensure that traffic does not reach a container that is not ready for it, and that containers are restarted when they fail.

## Configure Probes

[Probes](#) have a number of fields that you can use to more precisely control the behavior of liveness and readiness checks:

- `initialDelaySeconds`: Number of seconds after the container has started before liveness or readiness probes are initiated. Defaults to 0 seconds. Minimum value is 0.
- `periodSeconds`: How often (in seconds) to perform the probe. Default to 10 seconds. Minimum value is 1.
- `timeoutSeconds`: Number of seconds after which the probe times out. Defaults to 1 second. Minimum value is 1.

- `successThreshold`: Minimum consecutive successes for the probe to be considered successful after having failed. Defaults to 1. Must be 1 for liveness. Minimum value is 1.
- `failureThreshold`: When a Pod starts and the probe fails, Kubernetes will try `failureThreshold` times before giving up. Giving up in case of liveness probe means restarting the container. In case of readiness probe the Pod will be marked Unready. Defaults to 3. Minimum value is 1.

[HTTP probes](#) have additional fields that can be set on `httpGet`:

- `host`: Host name to connect to, defaults to the pod IP. You probably want to set "Host" in `httpHeaders` instead.
- `scheme`: Scheme to use for connecting to the host (HTTP or HTTPS). Defaults to HTTP.
- `path`: Path to access on the HTTP server.
- `httpHeaders`: Custom headers to set in the request. HTTP allows repeated headers.
- `port`: Name or number of the port to access on the container. Number must be in the range 1 to 65535.

For an HTTP probe, the kubelet sends an HTTP request to the specified path and port to perform the check. The kubelet sends the probe to the pod's IP address, unless the address is overridden by the optional `host` field in `httpGet`. If `scheme` field is set to HTTPS, the kubelet sends an HTTPS request skipping the certificate verification. In most scenarios, you do not want to set the `host` field. Here's one scenario where you would set it. Suppose the Container listens on 127.0.0.1 and the Pod's `hostNetwork` field is true. Then `host`, under `httpGet`, should be set to 127.0.0.1. If your pod relies on virtual hosts, which is probably the more common case, you should not use `host`, but rather set the `Host` header in `httpHeaders`.

For a TCP probe, the kubelet makes the probe connection at the node, not in the pod, which means that you can not use a service name in the `host` parameter since the kubelet is unable to resolve it.

## What's next

- Learn more about [Container Probes](#).

## Reference

- [Pod](#)
- [Container](#)
- [Probe](#)

## Feedback

Was this page helpful?

Yes No

Thanks for the feedback. If you have a specific, answerable question about how to use Kubernetes, ask it on [Stack Overflow](#). Open an issue in the GitHub repo if you want to [report a problem](#) or [suggest an improvement](#).

---

[Create an Issue](#) [Edit This Page](#)

Page last modified on January 12, 2020 at 11:21 AM PST by [Be more explicit about what kind of probe \(#17943\)](#) ([Page History](#))

[Edit This Page](#)

# Assign Pods to Nodes

This page shows how to assign a Kubernetes Pod to a particular node in a Kubernetes cluster.

- [Before you begin](#)
- [Add a label to a node](#)
- [Create a pod that gets scheduled to your chosen node](#)
- [Create a pod that gets scheduled to specific node](#)
- [What's next](#)

## Before you begin

You need to have a Kubernetes cluster, and the `kubectl` command-line tool must be configured to communicate with your cluster. 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`.

## Add a label to a node

1. List the nodes in your cluster, along with their labels:

```
kubectl get nodes --show-labels
```

The output is similar to this:

NAME	STATUS	ROLES	AGE	VERSION	LABELS
worker0	Ready	<none>	1d	v1.13.0	...,kubernetes.io/hostname=worker0
worker1	Ready	<none>	1d	v1.13.0	...,kubernetes.io/hostname=worker1
worker2	Ready	<none>	1d	v1.13.0	...,kubernetes.io/hostname=worker2

2. Choose one of your nodes, and add a label to it:

```
kubectl label nodes <your-node-name> disktype:ssd
```

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

3. Verify that your chosen node has a `disktype:ssd` label:

```
kubectl get nodes --show-labels
```

The output is similar to this:

NAME	STATUS	ROLES	AGE	VERSION	LABELS
worker0	Ready	<none>	1d		
v1.13.0		...,disktype:ssd,kubernetes.io/hostname=worker0			
worker1	Ready	<none>	1d		
v1.13.0		...,kubernetes.io/hostname=worker1			
worker2	Ready	<none>	1d		
v1.13.0		...,kubernetes.io/hostname=worker2			

In the preceding output, you can see that the worker0 node has a disktype:ssd label.

## Create a pod that gets scheduled to your chosen node

This pod configuration file describes a pod that has a node selector, disktype: ssd. This means that the pod will get scheduled on a node that has a disktype:ssd label.

### [pods/pod-nginx.yaml](#)

```
apiVersion: v1
kind: Pod
metadata:
  name: nginx
  labels:
    env: test
spec:
  containers:
  - name: nginx
    image: nginx
    imagePullPolicy: IfNotPresent
  nodeSelector:
    disktype: ssd
```

1. Use the configuration file to create a pod that will get scheduled on your chosen node:

```
kubectl apply -f https://k8s.io/examples/pods/pod-nginx.yaml
```

2. Verify that the pod is running on your chosen node:

```
kubectl get pods --output=wide
```

The output is similar to this:

NAME	READY	STATUS	RESTARTS	AGE	IP
NODE					
nginx	1/1	Running	0	13s	10.200.0.4
worker0					

# Create a pod that gets scheduled to specific node

You can also schedule a pod to one specific node via setting `nodeName`.

## [pods/pod-nginx-specific-node.yaml](#)

```
apiVersion: v1
kind: Pod
metadata:
  name: nginx
spec:
  nodeName: foo-node # schedule pod to specific node
  containers:
    - name: nginx
      image: nginx
      imagePullPolicy: IfNotPresent
```

Use the configuration file to create a pod that will get scheduled on `foo-node` only.

## What's next

Learn more about [labels and selectors](#).

## Feedback

Was this page helpful?

Yes No

Thanks for the feedback. If you have a specific, answerable question about how to use Kubernetes, ask it on [Stack Overflow](#). Open an issue in the GitHub repo if you want to [report a problem](#) or [suggest an improvement](#).

---

[Create an Issue](#) [Edit This Page](#)

Page last modified on October 23, 2019 at 9:08 PM PST by [Modifying adding label to node steps \(#16993\)](#) ([Page History](#))

[Edit This Page](#)

# Configure Pod Initialization

This page shows how to use an Init Container to initialize a Pod before an application Container runs.

- [Before you begin](#)
- [Create a Pod that has an Init Container](#)
- [What's next](#)

## Before you begin

You need to have a Kubernetes cluster, and the `kubectl` command-line tool must be configured to communicate with your cluster. 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 Pod that has an Init Container

In this exercise you create a Pod that has one application Container and one Init Container. The init container runs to completion before the application container starts.

Here is the configuration file for the Pod:

## [pods/init-containers.yaml](#)

```
apiVersion: v1
kind: Pod
metadata:
  name: init-demo
spec:
  containers:
    - name: nginx
      image: nginx
      ports:
        - containerPort: 80
      volumeMounts:
        - name: workdir
          mountPath: /usr/share/nginx/html
    # These containers are run during pod initialization
  initContainers:
    - name: install
      image: busybox
      command:
        - wget
        - "-O"
        - "/work-dir/index.html"
      http://kubernetes.io
      volumeMounts:
        - name: workdir
          mountPath: "/work-dir"
  dnsPolicy: Default
  volumes:
    - name: workdir
      emptyDir: {}
```

In the configuration file, you can see that the Pod has a Volume that the init container and the application container share.

The init container mounts the shared Volume at `/work-dir`, and the application container mounts the shared Volume at `/usr/share/nginx/html`. The init container runs the following command and then terminates:

```
wget -O /work-dir/index.html http://kubernetes.io
```

Notice that the init container writes the `index.html` file in the root directory of the nginx server.

Create the Pod:

```
kubectl apply -f https://k8s.io/examples/pods/init-containers.yaml
```

Verify that the nginx container is running:

```
kubectl get pod init-demo
```

The output shows that the nginx container is running:

NAME	READY	STATUS	RESTARTS	AGE
init-demo	1/1	Running	0	1m

Get a shell into the nginx container running in the init-demo Pod:

```
kubectl exec -it init-demo -- /bin/bash
```

In your shell, send a GET request to the nginx server:

```
root@nginx:~# apt-get update
root@nginx:~# apt-get install curl
root@nginx:~# curl localhost
```

The output shows that nginx is serving the web page that was written by the init container:

```
<!Doctype html>
<html id="home">

<head>
...
"url": "http://kubernetes.io/"</script>
</head>
<body>
...
<p>Kubernetes is open source giving you the freedom to take
advantage ...</p>
...
```

## What's next

- Learn more about [communicating between Containers running in the same Pod](#).
- Learn more about [Init Containers](#).
- Learn more about [Volumes](#).
- Learn more about [Debugging Init Containers](#)

## Feedback

Was this page helpful?

Yes No

Thanks for the feedback. If you have a specific, answerable question about how to use Kubernetes, ask it on [Stack Overflow](#). Open an issue in the GitHub repo if you want to [report a problem](#) or [suggest an improvement](#).

[Create an Issue](#) [Edit This Page](#)

Page last modified on March 25, 2019 at 5:06 PM PST by [Official 1.14](#)  
[Release Docs \(#13174\)](#) ([Page History](#))

[Edit This Page](#)

# Attach Handlers to Container Lifecycle Events

This page shows how to attach handlers to Container lifecycle events. Kubernetes supports the postStart and preStop events. Kubernetes sends the postStart event immediately after a Container is started, and it sends the preStop event immediately before the Container is terminated.

- [Before you begin](#)

- [Define postStart and preStop handlers](#)
- [Discussion](#)
- [What's next](#)

## Before you begin

You need to have a Kubernetes cluster, and the kubectl command-line tool must be configured to communicate with your cluster. 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`.

## Define postStart and preStop handlers

In this exercise, you create a Pod that has one Container. The Container has handlers for the postStart and preStop events.

Here is the configuration file for the Pod:

```
pods/lifecycle-events.yaml

apiVersion: v1
kind: Pod
metadata:
  name: lifecycle-demo
spec:
  containers:
    - name: lifecycle-demo-container
      image: nginx
      lifecycle:
        postStart:
          exec:
            command: ["/bin/sh", "-c", "echo Hello from the
postStart handler > /usr/share/message"]
        preStop:
          exec:
            command: ["/bin/sh", "-c", "nginx -s quit; while
killall -0 nginx; do sleep 1; done"]
```

In the configuration file, you can see that the postStart command writes a message file to the Container's /usr/share directory. The preStop command shuts down nginx gracefully. This is helpful if the Container is being terminated because of a failure.

Create the Pod:

```
kubectl apply -f https://k8s.io/examples/pods/lifecycle-events.yaml
```

Verify that the Container in the Pod is running:

```
kubectl get pod lifecycle-demo
```

Get a shell into the Container running in your Pod:

```
kubectl exec -it lifecycle-demo -- /bin/bash
```

In your shell, verify that the `postStart` handler created the message file:

```
root@lifecycle-demo:/# cat /usr/share/message
```

The output shows the text written by the `postStart` handler:

```
Hello from the postStart handler
```

## Discussion

Kubernetes sends the `postStart` event immediately after the Container is created. There is no guarantee, however, that the `postStart` handler is called before the Container's entrypoint is called. The `postStart` handler runs asynchronously relative to the Container's code, but Kubernetes' management of the container blocks until the `postStart` handler completes. The Container's status is not set to `RUNNING` until the `postStart` handler completes.

Kubernetes sends the `preStop` event immediately before the Container is terminated. Kubernetes' management of the Container blocks until the `preStop` handler completes, unless the Pod's grace period expires. For more details, see [Termination of Pods](#).

**Note:** Kubernetes only sends the `preStop` event when a Pod is *terminated*. This means that the `preStop` hook is not invoked when the Pod is *completed*. This limitation is tracked in [issue #55087](#).

## What's next

- Learn more about [Container lifecycle hooks](#).
- Learn more about the [lifecycle of a Pod](#).

## Reference

- [Lifecycle](#)
- [Container](#)
- See `terminationGracePeriodSeconds` in [PodSpec](#)

# Feedback

Was this page helpful?

Yes No

Thanks for the feedback. If you have a specific, answerable question about how to use Kubernetes, ask it on [Stack Overflow](#). Open an issue in the GitHub repo if you want to [report a problem](#) or [suggest an improvement](#).

---

[Create an Issue](#) [Edit This Page](#)

Page last modified on October 23, 2019 at 6:32 PM PST by [Update link](#) ([#16966](#)) ([Page History](#))

[Edit This Page](#)

# Configure a Pod to Use a ConfigMap

ConfigMaps allow you to decouple configuration artifacts from image content to keep containerized applications portable. This page provides a series of usage examples demonstrating how to create ConfigMaps and configure Pods using data stored in ConfigMaps.

- [Before you begin](#)
- [Create a ConfigMap](#)
- [Define container environment variables using ConfigMap data](#)
- [Configure all key-value pairs in a ConfigMap as container environment variables](#)
- [Use ConfigMap-defined environment variables in Pod commands](#)
- [Add ConfigMap data to a Volume](#)
- [Understanding ConfigMaps and Pods](#)
- [What's next](#)

## Before you begin

You need to have a Kubernetes cluster, and the `kubectl` command-line tool must be configured to communicate with your cluster. 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 ConfigMap

You can use either `kubectl create configmap` or a ConfigMap generator in `kustomization.yaml` to create a ConfigMap. Note that `kubectl` starts to support `kustomization.yaml` since 1.14.

### Create a ConfigMap Using `kubectl create configmap`

Use the `kubectl create configmap` command to create ConfigMaps from [directories](#), [files](#), or [literal values](#):

```
kubectl create configmap <map-name> <data-source>
```

where `<map-name>` is the name you want to assign to the ConfigMap and `<data-source>` is the directory, file, or literal value to draw the data from.

When you are creating a ConfigMap based on a file, the key in the `<data-source>` defaults to the basename of the file, and the value defaults to the file content.

You can use [kubectl describe](#) or [kubectl get](#) to retrieve information about a ConfigMap.

## Create ConfigMaps from directories

You can use `kubectl create configmap` to create a ConfigMap from multiple files in the same directory. When you are creating a ConfigMap based on a directory, kubectl identifies files whose basename is a valid key in the directory and packages each of those files into the new ConfigMap. Any directory entries except regular files are ignored (e.g. subdirectories, symlinks, devices, etc).

For example:

```
# Create the local directory
mkdir -p configure-pod-container/configmap/

# Download the sample files into `configure-pod-container/
# configmap/` directory
wget https://kubernetes.io/examples/configmap/game.properties -O
configure-pod-container/configmap/game.properties
wget https://kubernetes.io/examples/configmap/ui.properties -O
configure-pod-container/configmap/ui.properties

# Create the configmap
kubectl create configmap game-config --from-file=configure-pod-
container/configmap/
```

The above command packages each file, in this case, `game.properties` and `ui.properties` in the `configure-pod-container/configmap/` directory into the `game-config` ConfigMap. You can display details of the ConfigMap using the following command:

```
kubectl describe configmaps game-config
```

The output is similar to this:

```
Name:          game-config
Namespace:    default
Labels:        <none>
Annotations:  <none>

Data
=====
game.properties:
-----
enemies=aliens
lives=3
enemies.cheat=true
enemies.cheat.level=noGoodRotten
secret.code.passphrase=UUDDLRLRBABAS
secret.code.allowed=true
```

```
secret.code.lives=30
ui.properties:
-----
color.good=purple
color.bad=yellow
allow.textmode=true
how.nice.to.look=fairlyNice
```

The game.properties and ui.properties files in the configure-pod-container/configmap/ directory are represented in the data section of the ConfigMap.

```
kubectl get configmaps game-config -o yaml
```

The output is similar to this:

```
apiVersion: v1
kind: ConfigMap
metadata:
  creationTimestamp: 2016-02-18T18:52:05Z
  name: game-config
  namespace: default
  resourceVersion: "516"
  uid: b4952dc3-d670-11e5-8cd0-68f728db1985
data:
  game.properties: |
    enemies=aliens
    lives=3
    enemies.cheat=true
    enemies.cheat.level=noGoodRotten
    secret.code.passphrase=UUDDLRLRBABAS
    secret.code.allowed=true
    secret.code.lives=30
  ui.properties: |
    color.good=purple
    color.bad=yellow
    allow.textmode=true
    how.nice.to.look=fairlyNice
```

## Create ConfigMaps from files

You can use `kubectl create configmap` to create a ConfigMap from an individual file, or from multiple files.

For example,

```
kubectl create configmap game-config-2 --from-file=configure-pod-
container/configmap/game.properties
```

would produce the following ConfigMap:

```
kubectl describe configmaps game-config-2
```

where the output is similar to this:

```
Name:          game-config-2
Namespace:    default
Labels:        <none>
Annotations:  <none>

Data
=====
game.properties:
-----
enemies=aliens
lives=3
enemies.cheat=true
enemies.cheat.level=noGoodRotten
secret.code.passphrase=UUDDLRLRBABAS
secret.code.allowed=true
secret.code.lives=30
```

You can pass in the `--from-file` argument multiple times to create a ConfigMap from multiple data sources.

```
kubectl create configmap game-config-2 --from-file=configure-pod-
container/configmap/game.properties --from-file=configure-pod-
container/configmap/ui.properties
```

You can display details of the `game-config-2` ConfigMap using the following command:

```
kubectl describe configmaps game-config-2
```

The output is similar to this:

```
Name:          game-config-2
Namespace:    default
Labels:        <none>
Annotations:  <none>

Data
=====
game.properties:
-----
enemies=aliens
lives=3
enemies.cheat=true
enemies.cheat.level=noGoodRotten
secret.code.passphrase=UUDDLRLRBABAS
secret.code.allowed=true
secret.code.lives=30
ui.properties:
-----
color.good=purple
color.bad=yellow
```

```
allow.textmode=true  
how.nice.to.look=fairlyNice
```

Use the option `--from-env-file` to create a ConfigMap from an env-file, for example:

```
# Env-files contain a list of environment variables.  
# These syntax rules apply:  
#   Each line in an env file has to be in VAR=VAL format.  
#   Lines beginning with # (i.e. comments) are ignored.  
#   Blank lines are ignored.  
#   There is no special handling of quotation marks (i.e. they  
will be part of the ConfigMap value)).  
  
# Download the sample files into `configure-pod-container/  
configmap/` directory  
wget https://kubernetes.io/examples/configmap/game-env-  
file.properties -O configure-pod-container/configmap/game-env-  
file.properties  
  
# The env-file `game-env-file.properties` looks like below  
cat configure-pod-container/configmap/game-env-file.properties  
enemies=aliens  
lives=3  
allowed="true"  
  
# This comment and the empty line above it are ignored
```

```
kubectl create configmap game-config-env-file \  
--from-env-file=configure-pod-container/configmap/game-  
env-file.properties
```

would produce the following ConfigMap:

```
kubectl get configmap game-config-env-file -o yaml
```

where the output is similar to this:

```
apiVersion: v1  
kind: ConfigMap  
metadata:  
  creationTimestamp: 2017-12-27T18:36:28Z  
  name: game-config-env-file  
  namespace: default  
  resourceVersion: "809965"  
  uid: d9d1ca5b-eb34-11e7-887b-42010a8002b8  
data:  
  allowed: '"true"'  
  enemies: aliens  
  lives: "3"
```

**Caution:** When passing `--from-env-file` multiple times to create a ConfigMap from multiple data sources, only the last env-file is used.

The behavior of passing `--from-env-file` multiple times is demonstrated by:

```
# Download the sample files into `configure-pod-container/
configmap/` directory
wget https://kubernetes.io/examples/configmap/ui-env-
file.properties -O configure-pod-container/configmap/ui-env-
file.properties

# Create the configmap
kubectl create configmap config-multi-env-files \
    --from-env-file=configure-pod-container/configmap/game-
env-file.properties \
    --from-env-file=configure-pod-container/configmap/ui-env-
file.properties
```

would produce the following ConfigMap:

```
kubectl get configmap config-multi-env-files -o yaml
```

where the output is similar to this:

```
apiVersion: v1
kind: ConfigMap
metadata:
  creationTimestamp: 2017-12-27T18:38:34Z
  name: config-multi-env-files
  namespace: default
  resourceVersion: "810136"
  uid: 252c4572-eb35-11e7-887b-42010a8002b8
data:
  color: purple
  how: fairlyNice
  textmode: "true"
```

## Define the key to use when creating a ConfigMap from a file

You can define a key other than the file name to use in the data section of your ConfigMap when using the `--from-file` argument:

```
kubectl create configmap game-config-3 --from-file=<my-key-
name>=<path-to-file>
```

where `<my-key-name>` is the key you want to use in the ConfigMap and `<path-to-file>` is the location of the data source file you want the key to represent.

For example:

```
kubectl create configmap game-config-3 --from-file=game-special-key=configure-pod-container/configmap/game.properties
```

would produce the following ConfigMap:

```
kubectl get configmaps game-config-3 -o yaml
```

where the output is similar to this:

```
apiVersion: v1
kind: ConfigMap
metadata:
  creationTimestamp: 2016-02-18T18:54:22Z
  name: game-config-3
  namespace: default
  resourceVersion: "530"
  uid: 05f8da22-d671-11e5-8cd0-68f728db1985
data:
  game-special-key: /
    enemies=aliens
    lives=3
    enemies.cheat=true
    enemies.cheat.level=noGoodRotten
    secret.code.passphrase=UUDDLRLRBABAS
    secret.code.allowed=true
    secret.code.lives=30
```

## Create ConfigMaps from literal values

You can use `kubectl create configmap` with the `--from-literal` argument to define a literal value from the command line:

```
kubectl create configmap special-config --from-literal=special.how=very --from-literal=special.type=charm
```

You can pass in multiple key-value pairs. Each pair provided on the command line is represented as a separate entry in the `data` section of the ConfigMap.

```
kubectl get configmaps special-config -o yaml
```

The output is similar to this:

```
apiVersion: v1
kind: ConfigMap
metadata:
  creationTimestamp: 2016-02-18T19:14:38Z
  name: special-config
  namespace: default
  resourceVersion: "651"
  uid: dadce046-d673-11e5-8cd0-68f728db1985
data:
```

```
special.how: very
special.type: charm
```

## Create a ConfigMap from generator

kubectl supports `kustomization.yaml` since 1.14. You can also create a ConfigMap from generators and then apply it to create the object on the Apiserver. The generators should be specified in a `kustomization.yaml` inside a directory.

### Generate ConfigMaps from files

For example, to generate a ConfigMap from files `configure-pod-container/configmap/game.properties`

```
# Create a kustomization.yaml file with ConfigMapGenerator
cat <<EOF > ./kustomization.yaml
configMapGenerator:
- name: game-config-4
  files:
  - configure-pod-container/configmap/game.properties
EOF
```

Apply the kustomization directory to create the ConfigMap object.

```
kubectl apply -k .
configmap/game-config-4-m9dm2f92bt created
```

You can check that the ConfigMap was created like this:

```
kubectl get configmap
NAME                      DATA   AGE
game-config-4-m9dm2f92bt    1      37s
```

```
kubectl describe configmaps/game-config-4-m9dm2f92bt
Name:          game-config-4-m9dm2f92bt
Namespace:     default
Labels:        <none>
Annotations:  kubectl.kubernetes.io/last-applied-configuration:
              {"apiVersion":"v1","data":{"game.properties":"enemies=aliens\nlives=3\nenemies.cheat=true\nenemies.cheat.level=noGoodRotten\nsecret.code.p..."}}

Data
====
```

```
game.properties:
-----
enemies=aliens
lives=3
enemies.cheat=true
enemies.cheat.level=noGoodRotten
```

```
secret.code.passphrase=UUDDLRLRBABAS
secret.code.allowed=true
secret.code.lives=30
Events: <none>
```

Note that the generated ConfigMap name has a suffix appended by hashing the contents. This ensures that a new ConfigMap is generated each time the content is modified.

## Define the key to use when generating a ConfigMap from a file

You can define a key other than the file name to use in the ConfigMap generator. For example, to generate a ConfigMap from files `configure-pod-container/configmap/game.properties` with the key `game-special-key`

```
# Create a kustomization.yaml file with ConfigMapGenerator
cat <<EOF >./kustomization.yaml
configMapGenerator:
- name: game-config-5
  files:
    - game-special-key=configure-pod-container/configmap/
game.properties
EOF
```

Apply the kustomization directory to create the ConfigMap object.

```
kubectl apply -k .
configmap/game-config-5-m67dt67794 created
```

## Generate ConfigMaps from Literals

To generate a ConfigMap from literals `special.type=charm` and `special.how=very`, you can specify the ConfigMap generator in `kustomization.yaml` as

```
# Create a kustomization.yaml file with ConfigMapGenerator
cat <<EOF >./kustomization.yaml
configMapGenerator:
- name: special-config-2
  literals:
    - special.how=very
    - special.type=charm
EOF
```

Apply the kustomization directory to create the ConfigMap object.

```
kubectl apply -k .
configmap/special-config-2-c92b5mmcf2 created
```

# Define container environment variables using ConfigMap data

## Define a container environment variable with data from a single ConfigMap

1. Define an environment variable as a key-value pair in a ConfigMap:

```
kubectl create configmap special-config --from-literal=specia  
l.how=very
```

2. Assign the `special.how` value defined in the ConfigMap to the `SPECIAL_LEVEL_KEY` environment variable in the Pod specification.

### [pods/pod-single-configmap-env-variable.yaml](#)

```
apiVersion: v1  
kind: Pod  
metadata:  
  name: dapi-test-pod  
spec:  
  containers:  
    - name: test-container  
      image: k8s.gcr.io/busybox  
      command: [ "/bin/sh", "-c", "env" ]  
      env:  
        # Define the environment variable  
        - name: SPECIAL_LEVEL_KEY  
          valueFrom:  
            configMapKeyRef:  
              # The ConfigMap containing the value you want to  
              assign to SPECIAL_LEVEL_KEY  
              name: special-config  
              # Specify the key associated with the value  
              key: special.how  
  restartPolicy: Never
```

Create the Pod:

```
kubectl create -f https://kubernetes.io/examples/pods/pod-  
single-configmap-env-variable.yaml
```

Now, the Pod's output includes environment variable `SPECIAL_LEVEL_KEY=ve  
ry`.

## Define container environment variables with data from multiple ConfigMaps

- As with the previous example, create the ConfigMaps first.

## [configmap/configmaps.yaml](#)

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: special-config
  namespace: default
data:
  special.how: very
---
apiVersion: v1
kind: ConfigMap
metadata:
  name: env-config
  namespace: default
data:
  log_level: INFO
```

Create the ConfigMap:

```
kubectl create -f https://kubernetes.io/examples/configmap/
configmaps.yaml
```

- Define the environment variables in the Pod specification.

## [pods/pod-multiple-configmap-env-variable.yaml](#)

```
apiVersion: v1
kind: Pod
metadata:
  name: dapi-test-pod
spec:
  containers:
    - name: test-container
      image: k8s.gcr.io/busybox
      command: [ "/bin/sh", "-c", "env" ]
      env:
        - name: SPECIAL_LEVEL_KEY
          valueFrom:
            configMapKeyRef:
              name: special-config
              key: special.how
        - name: LOG_LEVEL
          valueFrom:
            configMapKeyRef:
              name: env-config
              key: log_level
  restartPolicy: Never
```

Create the Pod:

```
kubectl create -f https://kubernetes.io/examples/pods/pod-multiple-configmap-env-variable.yaml
```

Now, the Pod's output includes environment variables SPECIAL\_LEVEL\_KEY=very and LOG\_LEVEL=INFO.

## Configure all key-value pairs in a ConfigMap as container environment variables

**Note:** This functionality is available in Kubernetes v1.6 and later.

- Create a ConfigMap containing multiple key-value pairs.

### [configmap/configmap-multikeys.yaml](#)

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: special-config
  namespace: default
data:
  SPECIAL_LEVEL: very
  SPECIAL_TYPE: charm
```

Create the ConfigMap:

```
kubectl create -f https://kubernetes.io/examples/configmap/configmap-multikeys.yaml
```

- Use `envFrom` to define all of the ConfigMap's data as container environment variables. The key from the ConfigMap becomes the environment variable name in the Pod.

## [pods/pod-configmap-envFrom.yaml](#)

```
apiVersion: v1
kind: Pod
metadata:
  name: dapi-test-pod
spec:
  containers:
    - name: test-container
      image: k8s.gcr.io/busybox
      command: [ "/bin/sh", "-c", "env" ]
      envFrom:
        - configMapRef:
            name: special-config
  restartPolicy: Never
```

Create the Pod:

```
kubectl create -f https://kubernetes.io/examples/pods/pod-
configmap-envFrom.yaml
```

Now, the Pod's output includes environment variables `SPECIAL_LEVEL=very` and `SPECIAL_TYPE=charm`.

## **Use ConfigMap-defined environment variables in Pod commands**

You can use ConfigMap-defined environment variables in the command section of the Pod specification using the `$(VAR_NAME)` Kubernetes substitution syntax.

For example, the following Pod specification

## [pods/pod-configmap-env-var-valueFrom.yaml](#)

```
apiVersion: v1
kind: Pod
metadata:
  name: dapi-test-pod
spec:
  containers:
    - name: test-container
      image: k8s.gcr.io/busybox
      command: [ "/bin/sh", "-c", "echo $(SPECIAL_LEVEL_KEY) $(SPECIAL_TYPE_KEY)" ]
      env:
        - name: SPECIAL_LEVEL_KEY
          valueFrom:
            configMapKeyRef:
              name: special-config
              key: SPECIAL_LEVEL
        - name: SPECIAL_TYPE_KEY
          valueFrom:
            configMapKeyRef:
              name: special-config
              key: SPECIAL_TYPE
  restartPolicy: Never
```

created by running

```
kubectl create -f https://kubernetes.io/examples/pods/pod-
configmap-env-var-valueFrom.yaml
```

produces the following output in the test-container container:

```
very charm
```

## Add ConfigMap data to a Volume

As explained in [Create ConfigMaps from files](#), when you create a ConfigMap using `--from-file`, the filename becomes a key stored in the `data` section of the ConfigMap. The file contents become the key's value.

The examples in this section refer to a ConfigMap named `special-config`, shown below.

### [configmap/configmap-multikeys.yaml](#)

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: special-config
  namespace: default
data:
  SPECIAL_LEVEL: very
  SPECIAL_TYPE: charm
```

Create the ConfigMap:

```
kubectl create -f https://kubernetes.io/examples/configmap/
configmap-multikeys.yaml
```

## Populate a Volume with data stored in a ConfigMap

Add the ConfigMap name under the `volumes` section of the Pod specification. This adds the ConfigMap data to the directory specified as `volumeMounts.mountPath` (in this case, `/etc/config`). The `command` section lists directory files with names that match the keys in ConfigMap.

### [pods/pod-configmap-volume.yaml](#)

```
apiVersion: v1
kind: Pod
metadata:
  name: dapi-test-pod
spec:
  containers:
    - name: test-container
      image: k8s.gcr.io/busybox
      command: [ "/bin/sh", "-c", "ls /etc/config/" ]
      volumeMounts:
        - name: config-volume
          mountPath: /etc/config
  volumes:
    - name: config-volume
      configMap:
        # Provide the name of the ConfigMap containing the
        # files you want
        # to add to the container
        name: special-config
  restartPolicy: Never
```

Create the Pod:

```
kubectl create -f https://kubernetes.io/examples/pods/pod-configmap-volume.yaml
```

When the pod runs, the command `ls /etc/config/` produces the output below:

```
SPECIAL_LEVEL  
SPECIAL_TYPE
```

**Caution:** If there are some files in the `/etc/config/` directory, they will be deleted.

## Add ConfigMap data to a specific path in the Volume

Use the `path` field to specify the desired file path for specific ConfigMap items. In this case, the `SPECIAL_LEVEL` item will be mounted in the `config-volume` volume at `/etc/config/keys`.

### [pods/pod-configmap-volume-specific-key.yaml](#)

```
apiVersion: v1
kind: Pod
metadata:
  name: dapi-test-pod
spec:
  containers:
    - name: test-container
      image: k8s.gcr.io/busybox
      command: [ "/bin/sh", "-c", "cat /etc/config/keys" ]
      volumeMounts:
        - name: config-volume
          mountPath: /etc/config
  volumes:
    - name: config-volume
      configMap:
        name: special-config
        items:
          - key: SPECIAL_LEVEL
            path: keys
  restartPolicy: Never
```

Create the Pod:

```
kubectl create -f https://kubernetes.io/examples/pods/pod-configmap-volume-specific-key.yaml
```

When the pod runs, the command `cat /etc/config/keys` produces the output below:

```
very
```

**Caution:** Like before, all previous files in the /etc/config/ directory will be deleted.

## Project keys to specific paths and file permissions

You can project keys to specific paths and specific permissions on a per-file basis. The [Secrets](#) user guide explains the syntax.

## Mounted ConfigMaps are updated automatically

When a ConfigMap already being consumed in a volume is updated, projected keys are eventually updated as well. Kubelet is checking whether the mounted ConfigMap is fresh on every periodic sync. However, it is using its local ttl-based cache for getting the current value of the ConfigMap. As a result, the total delay from the moment when the ConfigMap is updated to the moment when new keys are projected to the pod can be as long as kubelet sync period (1 minute by default) + ttl of ConfigMaps cache (1 minute by default) in kubelet. You can trigger an immediate refresh by updating one of the pod's annotations.

**Note:** A container using a ConfigMap as a [subPath](#) volume will not receive ConfigMap updates.

## Understanding ConfigMaps and Pods

The ConfigMap API resource stores configuration data as key-value pairs. The data can be consumed in pods or provide the configurations for system components such as controllers. ConfigMap is similar to [Secrets](#), but provides a means of working with strings that don't contain sensitive information. Users and system components alike can store configuration data in ConfigMap.

**Note:** ConfigMaps should reference properties files, not replace them. Think of the ConfigMap as representing something similar to the Linux /etc directory and its contents. For example, if you create a [Kubernetes Volume](#) from a ConfigMap, each data item in the ConfigMap is represented by an individual file in the volume.

The ConfigMap's data field contains the configuration data. As shown in the example below, this can be simple - like individual properties defined using --from-literal - or complex - like configuration files or JSON blobs defined using --from-file.

```
apiVersion: v1
kind: ConfigMap
metadata:
  creationTimestamp: 2016-02-18T19:14:38Z
  name: example-config
  namespace: default
data:
  # example of a simple property defined using --from-literal
```

```

example.property.1: hello
example.property.2: world
# example of a complex property defined using --from-file
example.property.file: | -
  property.1=value-1
  property.2=value-2
  property.3=value-3

```

## Restrictions

- You must create a ConfigMap before referencing it in a Pod specification (unless you mark the ConfigMap as "optional"). If you reference a ConfigMap that doesn't exist, the Pod won't start. Likewise, references to keys that don't exist in the ConfigMap will prevent the pod from starting.
- If you use `envFrom` to define environment variables from ConfigMaps, keys that are considered invalid will be skipped. The pod will be allowed to start, but the invalid names will be recorded in the event log (`InvalidVariableNames`). The log message lists each skipped key. For example:

```
kubectl get events
```

The output is similar to this:

	LASTSEEN	FIRSTSEEN	COUNT	NAME	KIND	SUBOBJECT
TYPE		REASON				
SOURCE			MESSAGE			
	0s	0s	1	dapi-test-pod	Pod	
Warning			InvalidEnvironmentVariableNames	{kubelet, 127.0.0.1}		Keys [1badkey, 2alsobad] from the EnvFrom configMap default/myconfig were skipped since they are considered invalid environment variable names.

- ConfigMaps reside in a specific [NamespaceAn abstraction used by Kubernetes to support multiple virtual clusters on the same physical cluster.](#). A ConfigMap can only be referenced by pods residing in the same namespace.
- You can't use ConfigMaps for [static podsA pod managed directly by the kubelet daemon on a specific node.](#), because the Kubelet does not support this.

## What's next

- Follow a real world example of [Configuring Redis using a ConfigMap](#).

## Feedback

Was this page helpful?

Yes No

Thanks for the feedback. If you have a specific, answerable question about how to use Kubernetes, ask it on [Stack Overflow](#). Open an issue in the GitHub repo if you want to [report a problem](#) or [suggest an improvement](#).

---

[Create an Issue](#) [Edit This Page](#)

Page last modified on February 06, 2020 at 4:07 PM PST by [Update kubectl create configmap section \(#18885\)](#) ([Page History](#))

[Edit This Page](#)

# Share Process Namespace between Containers in a Pod

**FEATURE STATE:** Kubernetes v1.17 [stable](#)

This feature is *stable*, meaning:

[Edit This Page](#)

## Create static Pods

*Static Pods* are managed directly by the kubelet daemon on a specific node, without the [API server](#)[Control plane component that serves the Kubernetes API](#). observing them. Unlike Pods that are managed by the control plane (for example, a [Deployment](#)[An API object that manages a replicated application.](#)); instead, the kubelet watches each static Pod (and restarts it if it crashes).

Static Pods are always bound to one [Kubelet](#)[An agent that runs on each node in the cluster. It makes sure that containers are running in a pod.](#) on a specific node.

The kubelet automatically tries to create a [mirror Pod](#)[An object in the API server that tracks a static pod on a kubelet.](#) on the Kubernetes API server for each static Pod. This means that the Pods running on a node are visible on the API server, but cannot be controlled from there.

**Note:** If you are running clustered Kubernetes and are using static Pods to run a Pod on every node, you should probably be using a [DaemonSet](#)[Ensures a copy of a Pod is running across a set of nodes in a cluster.](#) instead.

- [Before you begin](#)
- [Create a static pod](#)
- [Observe static pod behavior](#)
- [Dynamic addition and removal of static pods](#)

## Before you begin

You need to have a Kubernetes cluster, and the kubectl command-line tool must be configured to communicate with your cluster. 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 page assumes you're using [Docker](#)[Docker is a software technology providing operating-system-level virtualization also known as containers.](#) to

run Pods, and that your nodes are running the Fedora operating system. Instructions for other distributions or Kubernetes installations may vary.

## Create a static pod

You can configure a static Pod with either a [file system hosted configuration file](#) or a [web hosted configuration file](#).

### Filesystem-hosted static Pod manifest

Manifests are standard Pod definitions in JSON or YAML format in a specific directory. Use the `staticPodPath`: <the directory> field in the [kubelet configuration file](#), which periodically scans the directory and creates/deletes static Pods as YAML/JSON files appear/disappear there. Note that the kubelet will ignore files starting with dots when scanning the specified directory.

For example, this is how to start a simple web server as a static Pod:

1. Choose a node where you want to run the static Pod. In this example, it's `my-node1`.

```
ssh my-node1
```

2. Choose a directory, say `/etc/kubelet.d` and place a web server Pod definition there, e.g. `/etc/kubelet.d/static-web.yaml`:

```
# Run this command on the node where kubelet is running
mkdir /etc/kubelet.d/
cat <<EOF >/etc/kubelet.d/static-web.yaml
apiVersion: v1
kind: Pod
metadata:
  name: static-web
  labels:
    role: myrole
spec:
  containers:
    - name: web
      image: nginx
      ports:
        - name: web
          containerPort: 80
          protocol: TCP
EOF
```

3. Configure your kubelet on the node to use this directory by running it with `--pod-manifest-path=/etc/kubelet.d/` argument. On Fedora edit `/etc/kubernetes/kubelet` to include this line:

```
KUBELET_ARGS="--cluster-dns=10.254.0.10 --cluster-
domain=kube.local --pod-manifest-path=/etc/kubelet.d/"
```

or add the `staticPodPath: <the directory>` field in the [kubelet configuration file](#).

4. Restart the kubelet. On Fedora, you would run:

```
# Run this command on the node where the kubelet is running
systemctl restart kubelet
```

## Web-hosted static pod manifest

Kubelet periodically downloads a file specified by `--manifest-url=<URL>` argument and interprets it as a JSON/YAML file that contains Pod definitions. Similar to how [filesystem-hosted manifests](#) work, the kubelet refetches the manifest on a schedule. If there are changes to the list of static Pods, the kubelet applies them.

To use this approach:

1. Create a YAML file and store it on a web server so that you can pass the URL of that file to the kubelet.

```
apiVersion: v1
kind: Pod
metadata:
  name: static-web
  labels:
    role: myrole
spec:
  containers:
    - name: web
      image: nginx
      ports:
        - name: web
          containerPort: 80
          protocol: TCP
```

2. Configure the kubelet on your selected node to use this web manifest by running it with `--manifest-url=<manifest-url>`. On Fedora, edit `/etc/kubernetes/kubelet` to include this line:

```
KUBELET_ARGS="--cluster-dns=10.254.0.10 --cluster-
domain=kube.local --manifest-url=<manifest-url>"
```

3. Restart the kubelet. On Fedora, you would run:

```
# Run this command on the node where the kubelet is running
systemctl restart kubelet
```

# Observe static pod behavior

When the kubelet starts, it automatically starts all defined static Pods. As you have defined a static Pod and restarted the kubelet, the new static Pod should already be running.

You can view running containers (including static Pods) by running (on the node):

```
# Run this command on the node where the kubelet is running
docker ps
```

The output might be something like:

CONTAINER ID	IMAGE	COMMAND	CREATED
STATUS	PORTS	NAMES	
f6d05272b57e	nginx:latest	"nginx"	8 minutes ago
minutes		k8s_web.6f802af4_static-web-fk-	Up 8
		node1_default_67e24ed9466ba55986d120c867395f3c_378e5f3c	

You can see the mirror Pod on the API server:

```
kubectl get pods
```

NAME	READY	STATUS	RESTARTS	AGE
static-web-my-node1	1/1	Running	0	2m

**Note:** Make sure the kubelet has permission to create the mirror Pod in the API server. If not, the creation request is rejected by the API server. See [PodSecurityPolicy](#).

[Labels](#)[Tags](#) objects with identifying attributes that are meaningful and relevant to users. from the static Pod are propagated into the mirror Pod. You can use those labels as normal via [selectors](#)[Allows users to filter a list of resources based on labels.](#), etc.

If you try to use kubectl to delete the mirror Pod from the API server, the kubelet *doesn't* remove the static Pod:

```
kubectl delete pod static-web-my-node1
```

```
pod "static-web-my-node1" deleted
```

You can see that the Pod is still running:

```
kubectl get pods
```

NAME	READY	STATUS	RESTARTS	AGE
static-web-my-node1	1/1	Running	0	12s

Back on your node where the kubelet is running, you can try to stop the Docker container manually. You'll see that, after a time, the kubelet will notice and will restart the Pod automatically:

```
# Run these commands on the node where the kubelet is running
docker stop f6d05272b57e # replace with the ID of your container
sleep 20
docker ps
```

CONTAINER ID	IMAGE	COMMAND	
CREATED	...		
5b920cbaf8b1 seconds ago ...	nginx:latest	"nginx -g 'daemon off'; sleep 20"	2

## Dynamic addition and removal of static pods

The running kubelet periodically scans the configured directory (`/etc/kubelet.d` in our example) for changes and adds/removes Pods as files appear/disappear in this directory.

```
# This assumes you are using filesystem-hosted static Pod
# configuration
# Run these commands on the node where the kubelet is running
#
mv /etc/kubelet.d/static-web.yaml /tmp
sleep 20
docker ps
# You see that no nginx container is running
mv /tmp/static-web.yaml /etc/kubelet.d/
sleep 20
docker ps
```

CONTAINER ID	IMAGE	COMMAND	
CREATED	...		
e7a62e3427f1 seconds ago	nginx:latest	"nginx -g 'daemon off'; sleep 20"	27

## Feedback

Was this page helpful?

Yes No

Thanks for the feedback. If you have a specific, answerable question about how to use Kubernetes, ask it on [Stack Overflow](#). Open an issue in the GitHub repo if you want to [report a problem](#) or [suggest an improvement](#).

---

[Create an Issue](#) [Edit This Page](#)

Page last modified on January 05, 2020 at 9:57 AM PST by [fix words issue \(#18312\)](#) ([Page History](#))

[Edit This Page](#)

# Translate a Docker Compose File to Kubernetes Resources

What's Kompose? It's a conversion tool for all things compose (namely Docker Compose) to container orchestrators (Kubernetes or OpenShift).

More information can be found on the Kompose website at <http://kompose.io>.

- [Before you begin](#)
- [Install Kompose](#)
- [Use Kompose](#)
- [User Guide](#)
- [kompose convert](#)
- [kompose up](#)
- [kompose down](#)
- [Build and Push Docker Images](#)
- [Alternative Conversions](#)
- [Labels](#)
- [Restart](#)
- [Docker Compose Versions](#)

## Before you begin

You need to have a Kubernetes cluster, and the `kubectl` command-line tool must be configured to communicate with your cluster. 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`.

## Install Kompose

We have multiple ways to install Kompose. Our preferred method is downloading the binary from the latest GitHub release.

- [GitHub download](#)
- [Build from source](#)
- [CentOS package](#)
- [Fedora package](#)
- [Homebrew \(macOS\)](#)

Kompose is released via GitHub on a three-week cycle, you can see all current releases on the [GitHub release page](#).

```
# Linux
curl -L https://github.com/kubernetes/kompose/releases/download/v1.16.0/kompose-linux-amd64 -o kompose

# macOS
curl -L https://github.com/kubernetes/kompose/releases/download/v1.16.0/kompose-darwin-amd64 -o kompose

# Windows
```

```
curl -L https://github.com/kubernetes/kompose/releases/download/v1.16.0/kompose-windows-amd64.exe -o kompose.exe
```

```
chmod +x kompose
sudo mv ./kompose /usr/local/bin/kompose
```

Alternatively, you can download the [tarball](#).

Installing using `go get` pulls from the master branch with the latest development changes.

```
go get -u github.com/kubernetes/kompose
```

[Edit This Page](#)

# Declarative Management of Kubernetes Objects Using Configuration Files

Kubernetes objects can be created, updated, and deleted by storing multiple object configuration files in a directory and using `kubectl apply` to recursively create and update those objects as needed. This method retains writes made to live objects without merging the changes back into the object configuration files. `kubectl diff` also gives you a preview of what changes `apply` will make.

- [Before you begin](#)
- [Trade-offs](#)
- [Overview](#)
- [How to create objects](#)
- [How to update objects](#)
- [How to delete objects](#)
- [How to view an object](#)
- [How apply calculates differences and merges changes](#)
- [Default field values](#)
- [How to change ownership of a field between the configuration file and direct imperative writers](#)
- [Changing management methods](#)
- [Defining controller selectors and PodTemplate labels](#)
- [What's next](#)

## Before you begin

Install [kubectl](#).

You need to have a Kubernetes cluster, and the `kubectl` command-line tool must be configured to communicate with your cluster. 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`.

## Trade-offs

The `kubectl` tool supports three kinds of object management:

- Imperative commands
- Imperative object configuration
- Declarative object configuration

See [Kubernetes Object Management](#) for a discussion of the advantages and disadvantage of each kind of object management.

## Overview

Declarative object configuration requires a firm understanding of the Kubernetes object definitions and configuration. Read and complete the following documents if you have not already:

- [Managing Kubernetes Objects Using Imperative Commands](#)
- [Imperative Management of Kubernetes Objects Using Configuration Files](#)

Following are definitions for terms used in this document:

- *object configuration file / configuration file*: A file that defines the configuration for a Kubernetes object. This topic shows how to pass configuration files to `kubectl apply`. Configuration files are typically stored in source control, such as Git.
- *live object configuration / live configuration*: The live configuration values of an object, as observed by the Kubernetes cluster. These are kept in the Kubernetes cluster storage, typically etcd.
- *declarative configuration writer / declarative writer*: A person or software component that makes updates to a live object. The live writers referred to in this topic make changes to object configuration files and run `kubectl apply` to write the changes.

## How to create objects

Use `kubectl apply` to create all objects, except those that already exist, defined by configuration files in a specified directory:

```
kubectl apply -f <directory>/
```

This sets the `kubectl.kubernetes.io/last-applied-configuration: '{...}'` annotation on each object. The annotation contains the contents of the object configuration file that was used to create the object.

**Note:** Add the `-R` flag to recursively process directories.

Here's an example of an object configuration file:

#### [application/simple\\_deployment.yaml](#)

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deployment
spec:
  selector:
    matchLabels:
      app: nginx
  minReadySeconds: 5
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
        - name: nginx
          image: nginx:1.7.9
          ports:
            - containerPort: 80
```

Run `kubectl diff` to print the object that will be created:

```
kubectl diff -f https://k8s.io/examples/application/
simple_deployment.yaml
```

**Note:**

`diff` uses [server-side dry-run](#), which needs to be enabled on `kube-apiserver`.

Since `diff` performs a server-side apply request in dry-run mode, it requires granting PATCH, CREATE, and UPDATE permissions. See [Dry-Run Authorization](#) for details.

Create the object using `kubectl apply`:

```
kubectl apply -f https://k8s.io/examples/application/
simple_deployment.yaml
```

Print the live configuration using `kubectl get`:

```
kubectl get -f https://k8s.io/examples/application/simple_deployment.yaml -o yaml
```

The output shows that the `kubectl.kubernetes.io/last-applied-configuration` annotation was written to the live configuration, and it matches the configuration file:

```
kind: Deployment
metadata:
  annotations:
    # ...
    # This is the json representation of simple_deployment.yaml
    # It was written by kubectl apply when the object was created
    kubectl.kubernetes.io/last-applied-configuration: /
      {"apiVersion": "apps/v1", "kind": "Deployment",
       "metadata": {"annotations": {}, "name": "nginx-deployment", "namespace": "default"},
       "spec": {"minReadySeconds": 5, "selector": {"matchLabels": {"app": "nginx"}}, "template": {"metadata": {"labels": {"app": "nginx"}}, "spec": {"containers": [{"image": "nginx:1.7.9", "name": "nginx", "ports": [{"containerPort": 80}]}]}}}}
    # ...
spec:
  # ...
  minReadySeconds: 5
  selector:
    matchLabels:
      # ...
      app: nginx
  template:
    metadata:
      # ...
      labels:
        app: nginx
  spec:
    containers:
      - image: nginx:1.7.9
        # ...
        name: nginx
        ports:
          - containerPort: 80
        # ...
      # ...
    # ...
# ...
```

# How to update objects

You can also use `kubectl apply` to update all objects defined in a directory, even if those objects already exist. This approach accomplishes the following:

1. Sets fields that appear in the configuration file in the live configuration.
2. Clears fields removed from the configuration file in the live configuration.

```
kubectl diff -f <directory>/  
kubectl apply -f <directory>/
```

**Note:** Add the `-R` flag to recursively process directories.

Here's an example configuration file:

## [application/simple\\_deployment.yaml](#)

```
apiVersion: apps/v1  
kind: Deployment  
metadata:  
  name: nginx-deployment  
spec:  
  selector:  
    matchLabels:  
      app: nginx  
  minReadySeconds: 5  
  template:  
    metadata:  
      labels:  
        app: nginx  
    spec:  
      containers:  
        - name: nginx  
          image: nginx:1.7.9  
          ports:  
            - containerPort: 80
```

Create the object using `kubectl apply`:

```
kubectl apply -f https://k8s.io/examples/application/  
simple_deployment.yaml
```

**Note:** For purposes of illustration, the preceding command refers to a single configuration file instead of a directory.

Print the live configuration using `kubectl get`:

```
kubectl get -f https://k8s.io/examples/application/simple_deployment.yaml -o yaml
```

The output shows that the `kubectl.kubernetes.io/last-applied-configuration` annotation was written to the live configuration, and it matches the configuration file:

```
kind: Deployment
metadata:
  annotations:
    # ...
    # This is the json representation of simple_deployment.yaml
    # It was written by kubectl apply when the object was created
    kubectl.kubernetes.io/last-applied-configuration: /
      {"apiVersion": "apps/v1", "kind": "Deployment",
       "metadata": {"annotations": {}, "name": "nginx-deployment", "namespace": "default"},
       "spec": {"minReadySeconds": 5, "selector": {"matchLabels": {"app": "nginx"}}, "template": {"metadata": {"labels": {"app": "nginx"}}, "spec": {"containers": [{"image": "nginx:1.7.9", "name": "nginx", "ports": [{"containerPort": 80}]}]}}}}
    # ...
spec:
  # ...
  minReadySeconds: 5
  selector:
    matchLabels:
      # ...
      app: nginx
  template:
    metadata:
      # ...
      labels:
        app: nginx
  spec:
    containers:
      - image: nginx:1.7.9
        # ...
        name: nginx
        ports:
          - containerPort: 80
        # ...
      # ...
    # ...
# ...
```

Directly update the `replicas` field in the live configuration by using `kubectl scale`. This does not use `kubectl apply`:

```
kubectl scale deployment/nginx-deployment --replicas=2
```

Print the live configuration using `kubectl get`:

```
kubectl get -f https://k8s.io/examples/application/simple_deployment.yaml -o yaml
```

The output shows that the `replicas` field has been set to 2, and the `last-applied-configuration` annotation does not contain a `replicas` field:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  annotations:
    # ...
    # note that the annotation does not contain replicas
    # because it was not updated through apply
    kubectl.kubernetes.io/last-applied-configuration: /
      {"apiVersion":"apps/v1","kind":"Deployment",
       "metadata": {"annotations":{},"name":"nginx-deployment","namespace":"default"},
       "spec": {"minReadySeconds":5,"selector": {"matchLabels": {"app": "nginx"}}, "template": {"metadata": {"labels": {"app": "nginx"}}, "spec": {"containers": [{"image": "nginx:1.7.9", "name": "nginx", "ports": [{"containerPort": 80}]}]}}}}
    # ...
spec:
  replicas: 2 # written by scale
  # ...
  minReadySeconds: 5
  selector:
    matchLabels:
      # ...
      app: nginx
  template:
    metadata:
      # ...
      labels:
        app: nginx
  spec:
    containers:
      - image: nginx:1.7.9
        # ...
        name: nginx
        ports:
          - containerPort: 80
    # ...
```

Update the `simple_deployment.yaml` configuration file to change the image from `nginx:1.7.9` to `nginx:1.11.9`, and delete the `minReadySeconds` field:

### [application/update\\_deployment.yaml](#)

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deployment
spec:
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
        - name: nginx
          image: nginx:1.11.9 # update the image
          ports:
            - containerPort: 80
```

Apply the changes made to the configuration file:

```
kubectl diff -f https://k8s.io/examples/application/
update_deployment.yaml
kubectl apply -f https://k8s.io/examples/application/
update_deployment.yaml
```

Print the live configuration using kubectl get:

```
kubectl get -f https://k8s.io/examples/application/
simple_deployment.yaml -o yaml
```

The output shows the following changes to the live configuration:

- The replicas field retains the value of 2 set by kubectl scale. This is possible because it is omitted from the configuration file.
- The image field has been updated to nginx:1.11.9 from nginx:1.7.9.
- The last-applied-configuration annotation has been updated with the new image.
- The minReadySeconds field has been cleared.
- The last-applied-configuration annotation no longer contains the minReadySeconds field.

```
apiVersion: apps/v1
kind: Deployment
metadata:
  annotations:
# ...
# The annotation contains the updated image to nginx 1.11.9,
```

```

# but does not contain the updated replicas to 2
kubectl.kubernetes.io/last-applied-configuration: /
  {"apiVersion": "apps/v1", "kind": "Deployment",
    "metadata": {"annotations": {}, "name": "nginx-deployment", "namespace": "default"}, 
    "spec": {"selector": {"matchLabels": {"app": "nginx"}}, "template": {"metadata": {"labels": {"app": "nginx"}}, "spec": {"containers": [{"image": "nginx:1.11.9", "name": "nginx"}], "ports": [{"containerPort": 80}]}]}}
# ...
spec:
replicas: 2 # Set by `kubectl scale`. Ignored by `kubectl apply`.
# minReadySeconds cleared by `kubectl apply`
# ...
selector:
matchLabels:
# ...
  app: nginx
template:
metadata:
# ...
labels:
  app: nginx
spec:
  containers:
    - image: nginx:1.11.9 # Set by `kubectl apply`
      # ...
      name: nginx
      ports:
        - containerPort: 80
      # ...
# ...
# ...
# ...

```

**Warning:** Mixing `kubectl apply` with the imperative object configuration commands `create` and `replace` is not supported. This is because `create` and `replace` do not retain the `kubectl.kubernetes.io/last-applied-configuration` that `kubectl apply` uses to compute updates.

## How to delete objects

There are two approaches to delete objects managed by `kubectl apply`.

## **Recommended: `kubectl delete -f <filename>`**

Manually deleting objects using the imperative command is the recommended approach, as it is more explicit about what is being deleted, and less likely to result in the user deleting something unintentionally:

```
kubectl delete -f <filename>
```

## **Alternative: `kubectl apply -f <directory/> --prune -l your=label`**

Only use this if you know what you are doing.

**Warning:** `kubectl apply --prune` is in alpha, and backwards incompatible changes might be introduced in subsequent releases.

**Warning:** You must be careful when using this command, so that you do not delete objects unintentionally.

As an alternative to `kubectl delete`, you can use `kubectl apply` to identify objects to be deleted after their configuration files have been removed from the directory. Apply with `--prune` queries the API server for all objects matching a set of labels, and attempts to match the returned live object configurations against the object configuration files. If an object matches the query, and it does not have a configuration file in the directory, and it has a `last-applied-configuration` annotation, it is deleted.

```
kubectl apply -f <directory/> --prune -l <labels>
```

**Warning:** Apply with `prune` should only be run against the root directory containing the object configuration files. Running against sub-directories can cause objects to be unintentionally deleted if they are returned by the label selector query specified with `-l <labels>` and do not appear in the subdirectory.

## **How to view an object**

You can use `kubectl get` with `-o yaml` to view the configuration of a live object:

```
kubectl get -f <filename|url> -o yaml
```

## **How `apply` calculates differences and merges changes**

**Caution:** A *patch* is an update operation that is scoped to specific fields of an object instead of the entire object. This enables updating only a specific set of fields on an object without reading the object first.

When `kubectl apply` updates the live configuration for an object, it does so by sending a patch request to the API server. The patch defines updates scoped to specific fields of the live object configuration. The `kubectl apply` command calculates this patch request using the configuration file, the live configuration, and the `last-applied-configuration` annotation stored in the live configuration.

## Merge patch calculation

The `kubectl apply` command writes the contents of the configuration file to the `kubectl.kubernetes.io/last-applied-configuration` annotation. This is used to identify fields that have been removed from the configuration file and need to be cleared from the live configuration. Here are the steps used to calculate which fields should be deleted or set:

1. Calculate the fields to delete. These are the fields present in `last-applied-configuration` and missing from the configuration file.
2. Calculate the fields to add or set. These are the fields present in the configuration file whose values don't match the live configuration.

Here's an example. Suppose this is the configuration file for a Deployment object:

```
application/update\_deployment.yaml  
  
apiVersion: apps/v1  
kind: Deployment  
metadata:  
  name: nginx-deployment  
spec:  
  selector:  
    matchLabels:  
      app: nginx  
  template:  
    metadata:  
      labels:  
        app: nginx  
    spec:  
      containers:  
      - name: nginx  
        image: nginx:1.11.9 # update the image  
        ports:  
        - containerPort: 80
```

Also, suppose this is the live configuration for the same Deployment object:

```
apiVersion: apps/v1  
kind: Deployment  
metadata:  
  annotations:
```

```

# ...
# note that the annotation does not contain replicas
# because it was not updated through apply
kubectl.kubernetes.io/last-applied-configuration: /
  {"apiVersion": "apps/v1", "kind": "Deployment",
    "metadata": {"annotations": {}, "name": "nginx-deployment", "namespace": "default"},
    "spec": {"minReadySeconds": 5, "selector": {"matchLabels": {"app": "nginx"}}, "template": {"metadata": {"labels": {"app": "nginx"}}, "spec": {"containers": [{"image": "nginx:1.7.9", "name": "nginx", "ports": [{"containerPort": 80}]}]}}}}
# ...
spec:
  replicas: 2 # written by scale
# ...
  minReadySeconds: 5
  selector:
    matchLabels:
      # ...
      app: nginx
  template:
    metadata:
      # ...
      labels:
        app: nginx
  spec:
    containers:
      - image: nginx:1.7.9
        # ...
        name: nginx
        ports:
          - containerPort: 80
# ...

```

Here are the merge calculations that would be performed by `kubectl apply`:

1. Calculate the fields to delete by reading values from `last-applied-configuration` and comparing them to values in the configuration file. Clear fields explicitly set to null in the local object configuration file regardless of whether they appear in the `last-applied-configuration`. In this example, `minReadySeconds` appears in the `last-applied-configuration` annotation, but does not appear in the configuration file. **Action:** Clear `minReadySeconds` from the live configuration.
2. Calculate the fields to set by reading values from the configuration file and comparing them to values in the live configuration. In this example, the value of `image` in the configuration file does not match the value in the live configuration. **Action:** Set the value of `image` in the live configuration.

3. Set the `last-applied-configuration` annotation to match the value of the configuration file.
4. Merge the results from 1, 2, 3 into a single patch request to the API server.

Here is the live configuration that is the result of the merge:

```

apiVersion: apps/v1
kind: Deployment
metadata:
  annotations:
    # ...
    # The annotation contains the updated image to nginx 1.11.9,
    # but does not contain the updated replicas to 2
    kubectl.kubernetes.io/last-applied-configuration: /
      {"apiVersion":"apps/v1","kind":"Deployment",
       "metadata": {"annotations":{}, "name": "nginx-deployment", "namespace": "default"},
       "spec": {"selector": {"matchLabels": {"app": "nginx"}}, "template": {"metadata": {"labels": {"app": "nginx"}}, "spec": {"containers": [{"image": "nginx:1.11.9", "name": "nginx"}]}, "ports": [{"containerPort": 80}]}]}
    # ...
spec:
  selector:
    matchLabels:
      # ...
      app: nginx
  replicas: 2 # Set by `kubectl scale`. Ignored by `kubectl apply`.
  # minReadySeconds cleared by `kubectl apply`
  # ...
  template:
    metadata:
      # ...
      labels:
        app: nginx
  spec:
    containers:
      - image: nginx:1.11.9 # Set by `kubectl apply`
        # ...
        name: nginx
        ports:
          - containerPort: 80
            # ...
        # ...
      # ...
    # ...
# ...

```

## How different types of fields are merged

How a particular field in a configuration file is merged with the live configuration depends on the type of the field. There are several types of fields:

- *primitive*: A field of type string, integer, or boolean. For example, `image` and `replicas` are primitive fields. **Action**: Replace.
- *map*, also called *object*: A field of type map or a complex type that contains subfields. For example, `labels`, `annotations`, `spec` and `meta-data` are all maps. **Action**: Merge elements or subfields.
- *list*: A field containing a list of items that can be either primitive types or maps. For example, `containers`, `ports`, and `args` are lists. **Action**: Varies.

When `kubectl apply` updates a map or list field, it typically does not replace the entire field, but instead updates the individual subelements. For instance, when merging the `spec` on a Deployment, the entire `spec` is not replaced. Instead the subfields of `spec`, such as `replicas`, are compared and merged.

### Merging changes to primitive fields

Primitive fields are replaced or cleared.

**Note:** - is used for "not applicable" because the value is not used.

Field in object configuration file	Field in live object configuration	Field in last-applied-configuration	Action
Yes	Yes	-	Set live to configuration file value.
Yes	No	-	Set live to local configuration.
No	-	Yes	Clear from live configuration.
No	-	No	Do nothing. Keep live value.

### Merging changes to map fields

Fields that represent maps are merged by comparing each of the subfields or elements of the map:

**Note:** - is used for "not applicable" because the value is not used.

Yes	Yes	-	Compare sub fields values.
-----	-----	---	----------------------------

<b>Key in object configuration file</b>	<b>Key in live object configuration</b>	<b>Field in last-applied-configuration</b>	<b>Action</b>
Yes	No	-	Set live to local configuration.
No	-	Yes	Delete from live configuration.
No	-	No	Do nothing. Keep live value.

## Merging changes for fields of type list

Merging changes to a list uses one of three strategies:

- Replace the list if all its elements are primitives.
- Merge individual elements in a list of complex elements.
- Merge a list of primitive elements.

The choice of strategy is made on a per-field basis.

### Replace the list if all its elements are primitives

Treat the list the same as a primitive field. Replace or delete the entire list. This preserves ordering.

**Example:** Use kubectl apply to update the args field of a Container in a Pod. This sets the value of args in the live configuration to the value in the configuration file. Any args elements that had previously been added to the live configuration are lost. The order of the args elements defined in the configuration file is retained in the live configuration.

```
# last-applied-configuration value
  args: ["a", "b"]

# configuration file value
  args: ["a", "c"]

# live configuration
  args: ["a", "b", "d"]

# result after merge
  args: ["a", "c"]
```

**Explanation:** The merge used the configuration file value as the new list value.

### Merge individual elements of a list of complex elements:

Treat the list as a map, and treat a specific field of each element as a key. Add, delete, or update individual elements. This does not preserve ordering.

This merge strategy uses a special tag on each field called a `patchMergeKey`. The `patchMergeKey` is defined for each field in the Kubernetes source code: [types.go](#) When merging a list of maps, the field specified as the `patchMergeKey` for a given element is used like a map key for that element.

**Example:** Use `kubectl apply` to update the `containers` field of a PodSpec. This merges the list as though it was a map where each element is keyed by name.

```
# last-applied-configuration value
  containers:
    - name: nginx
      image: nginx:1.10
    - name: nginx-helper-a # key: nginx-helper-a; will be
      deleted in result
      image: helper:1.3
    - name: nginx-helper-b # key: nginx-helper-b; will be
      retained
      image: helper:1.3

# configuration file value
  containers:
    - name: nginx
      image: nginx:1.10
    - name: nginx-helper-b
      image: helper:1.3
    - name: nginx-helper-c # key: nginx-helper-c; will be added
      in result
      image: helper:1.3

# live configuration
  containers:
    - name: nginx
      image: nginx:1.10
    - name: nginx-helper-a
      image: helper:1.3
    - name: nginx-helper-b
      image: helper:1.3
      args: ["run"] # Field will be retained
    - name: nginx-helper-d # key: nginx-helper-d; will be
      retained
      image: helper:1.3

# result after merge
  containers:
    - name: nginx
      image: nginx:1.10
      # Element nginx-helper-a was deleted
    - name: nginx-helper-b
      image: helper:1.3
      args: ["run"] # Field was retained
    - name: nginx-helper-c # Element was added
```

```
image: helper:1.3
- name: nginx-helper-d # Element was ignored
  image: helper:1.3
```

### Explanation:

- The container named "nginx-helper-a" was deleted because no container named "nginx-helper-a" appeared in the configuration file.
- The container named "nginx-helper-b" retained the changes to args in the live configuration. kubectl apply was able to identify that "nginx-helper-b" in the live configuration was the same "nginx-helper-b" as in the configuration file, even though their fields had different values (no args in the configuration file). This is because the patchMergeKey field value (name) was identical in both.
- The container named "nginx-helper-c" was added because no container with that name appeared in the live configuration, but one with that name appeared in the configuration file.
- The container named "nginx-helper-d" was retained because no element with that name appeared in the last-applied-configuration.

### Merge a list of primitive elements

As of Kubernetes 1.5, merging lists of primitive elements is not supported.

**Note:** Which of the above strategies is chosen for a given field is controlled by the patchStrategy tag in [types.go](#). If no patchStrategy is specified for a field of type list, then the list is replaced.

### Default field values

The API server sets certain fields to default values in the live configuration if they are not specified when the object is created.

Here's a configuration file for a Deployment. The file does not specify strategy:

### [application/simple\\_deployment.yaml](#)

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deployment
spec:
  selector:
    matchLabels:
      app: nginx
  minReadySeconds: 5
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
        - name: nginx
          image: nginx:1.7.9
          ports:
            - containerPort: 80
```

Create the object using `kubectl apply`:

```
kubectl apply -f https://k8s.io/examples/application/
simple_deployment.yaml
```

Print the live configuration using `kubectl get`:

```
kubectl get -f https://k8s.io/examples/application/
simple_deployment.yaml -o yaml
```

The output shows that the API server set several fields to default values in the live configuration. These fields were not specified in the configuration file.

```
apiVersion: apps/v1
kind: Deployment
# ...
spec:
  selector:
    matchLabels:
      app: nginx
  minReadySeconds: 5
  replicas: 1 # defaulted by apiserver
  strategy:
    rollingUpdate: # defaulted by apiserver - derived from
strategy.type
    maxSurge: 1
    maxUnavailable: 1
    type: RollingUpdate # defaulted by apiserver
```

```

template:
  metadata:
    creationTimestamp: null
    labels:
      app: nginx
  spec:
    containers:
      - image: nginx:1.7.9
        imagePullPolicy: IfNotPresent # defaulted by apiserver
        name: nginx
        ports:
          - containerPort: 80
            protocol: TCP # defaulted by apiserver
        resources: {} # defaulted by apiserver
        terminationMessagePath: /dev/termination-log # defaulted by apiserver
      dnsPolicy: ClusterFirst # defaulted by apiserver
      restartPolicy: Always # defaulted by apiserver
      securityContext: {} # defaulted by apiserver
      terminationGracePeriodSeconds: 30 # defaulted by apiserver
# ...

```

In a patch request, defaulted fields are not re-defaulted unless they are explicitly cleared as part of a patch request. This can cause unexpected behavior for fields that are defaulted based on the values of other fields. When the other fields are later changed, the values defaulted from them will not be updated unless they are explicitly cleared.

For this reason, it is recommended that certain fields defaulted by the server are explicitly defined in the configuration file, even if the desired values match the server defaults. This makes it easier to recognize conflicting values that will not be re-defaulted by the server.

### Example:

```

# last-applied-configuration
spec:
  template:
    metadata:
      labels:
        app: nginx
  spec:
    containers:
      - name: nginx
        image: nginx:1.7.9
        ports:
          - containerPort: 80

# configuration file
spec:
  strategy:
    type: Recreate # updated value

```

```

template:
  metadata:
    labels:
      app: nginx
spec:
  containers:
    - name: nginx
      image: nginx:1.7.9
      ports:
        - containerPort: 80

# live configuration
spec:
  strategy:
    type: RollingUpdate # defaulted value
    rollingUpdate: # defaulted value derived from type
      maxSurge : 1
      maxUnavailable: 1
  template:
    metadata:
      labels:
        app: nginx
  spec:
    containers:
      - name: nginx
        image: nginx:1.7.9
        ports:
          - containerPort: 80

# result after merge - ERROR!
spec:
  strategy:
    type: Recreate # updated value: incompatible with
rollingUpdate
    rollingUpdate: # defaulted value: incompatible with "type:
Recreate"
      maxSurge : 1
      maxUnavailable: 1
  template:
    metadata:
      labels:
        app: nginx
  spec:
    containers:
      - name: nginx
        image: nginx:1.7.9
        ports:
          - containerPort: 80

```

### **Explanation:**

1. The user creates a Deployment without defining `strategy.type`.

2. The server defaults `strategy.type` to `RollingUpdate` and defaults the `strategy.rollingUpdate` values.
3. The user changes `strategy.type` to `Recreate`. The `strategy.rollingUpdate` values remain at their defaulted values, though the server expects them to be cleared. If the `strategy.rollingUpdate` values had been defined initially in the configuration file, it would have been more clear that they needed to be deleted.
4. `Apply` fails because `strategy.rollingUpdate` is not cleared. The `strategy.rollingUpdate` field cannot be defined with a `strategy.type` of `Recreate`.

**Recommendation:** These fields should be explicitly defined in the object configuration file:

- Selectors and PodTemplate labels on workloads, such as Deployment, StatefulSet, Job, DaemonSet, ReplicaSet, and ReplicationController
- Deployment rollout strategy

## **How to clear server-defaulted fields or fields set by other writers**

Fields that do not appear in the configuration file can be cleared by setting their values to `null` and then applying the configuration file. For fields defaulted by the server, this triggers re-defaulting the values.

## **How to change ownership of a field between the configuration file and direct imperative writers**

These are the only methods you should use to change an individual object field:

- Use `kubectl apply`.
- Write directly to the live configuration without modifying the configuration file: for example, use `kubectl scale`.

### **Changing the owner from a direct imperative writer to a configuration file**

Add the field to the configuration file. For the field, discontinue direct updates to the live configuration that do not go through `kubectl apply`.

### **Changing the owner from a configuration file to a direct imperative writer**

As of Kubernetes 1.5, changing ownership of a field from a configuration file to an imperative writer requires manual steps:

- Remove the field from the configuration file.

- Remove the field from the `kubectl.kubernetes.io/last-applied-configuration` annotation on the live object.

## Changing management methods

Kubernetes objects should be managed using only one method at a time. Switching from one method to another is possible, but is a manual process.

**Note:** It is OK to use imperative deletion with declarative management.

### Migrating from imperative command management to declarative object configuration

Migrating from imperative command management to declarative object configuration involves several manual steps:

1. Export the live object to a local configuration file:

```
kubectl get <kind>/<name> -o yaml > <kind>_<name>.yaml
```

2. Manually remove the `status` field from the configuration file.

**Note:** This step is optional, as `kubectl apply` does not update the `status` field even if it is present in the configuration file.

3. Set the `kubectl.kubernetes.io/last-applied-configuration` annotation on the object:

```
kubectl replace --save-config -f <kind>_<name>.yaml
```

4. Change processes to use `kubectl apply` for managing the object exclusively.

### Migrating from imperative object configuration to declarative object configuration

1. Set the `kubectl.kubernetes.io/last-applied-configuration` annotation on the object:

```
kubectl replace --save-config -f <kind>_<name>.yaml
```

2. Change processes to use `kubectl apply` for managing the object exclusively.

## Defining controller selectors and PodTemplate labels

**Warning:** Updating selectors on controllers is strongly discouraged.

The recommended approach is to define a single, immutable PodTemplate label used only by the controller selector with no other semantic meaning.

### Example:

```
selector:  
  matchLabels:  
    controller-selector: "apps/v1/deployment/nginx"  
template:  
  metadata:  
    labels:  
      controller-selector: "apps/v1/deployment/nginx"
```

## What's next

- [Managing Kubernetes Objects Using Imperative Commands](#)
- [Imperative Management of Kubernetes Objects Using Configuration Files](#)
- [Kubectl Command Reference](#)
- [Kubernetes API Reference](#)

## Feedback

Was this page helpful?

Yes No

Thanks for the feedback. If you have a specific, answerable question about how to use Kubernetes, ask it on [Stack Overflow](#). Open an issue in the GitHub repo if you want to [report a problem](#) or [suggest an improvement](#).

---

[Create an Issue](#) [Edit This Page](#)

Page last modified on January 31, 2020 at 4:52 AM PST by [Document dry-run authorization requirements \(#18235\)](#) ([Page History](#))

[Edit This Page](#)

# Declarative Management of Kubernetes Objects Using Kustomize

[Kustomize](#) is a standalone tool to customize Kubernetes objects through a [kustomization file](#).

Since 1.14, Kubectl also supports the management of Kubernetes objects using a kustomization file. To view Resources found in a directory containing a kustomization file, run the following command:

```
kubectl kustomize <kustomization_directory>
```

To apply those Resources, run `kubectl apply` with `--kustomize` or `-k` flag:

```
kubectl apply -k <kustomization_directory>
```

- [Before you begin](#)
- [Overview of Kustomize](#)
- [Bases and Overlays](#)
- [How to apply/view/delete objects using Kustomize](#)
- [Kustomize Feature List](#)
- [What's next](#)

## Before you begin

Install [kubectl](#).

You need to have a Kubernetes cluster, and the `kubectl` command-line tool must be configured to communicate with your cluster. 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`.

## Overview of Kustomize

Kustomize is a tool for customizing Kubernetes configurations. It has the following features to manage application configuration files:

- generating resources from other sources
- setting cross-cutting fields for resources
- composing and customizing collections of resources

## Generating Resources

ConfigMap and Secret hold config or sensitive data that are used by other Kubernetes objects, such as Pods. The source of truth of ConfigMap or Secret are usually from somewhere else, such as a `.properties` file or a ssh key file. Kustomize has `secretGenerator` and `configMapGenerator`, which generate Secret and ConfigMap from files or literals.

## configMapGenerator

To generate a ConfigMap from a file, add an entry to `files` list in `configMapGenerator`. Here is an example of generating a ConfigMap with a data item from a file content.

```
# Create a application.properties file
cat <<EOF >application.properties
FOO=Bar
EOF

cat <<EOF >./kustomization.yaml
configMapGenerator:
- name: example-configmap-1
  files:
    - application.properties
EOF
```

The generated ConfigMap can be checked by the following command:

```
kubectl kustomize ./
```

The generated ConfigMap is:

```
apiVersion: v1
data:
  application.properties: |
    FOO=Bar
kind: ConfigMap
metadata:
  name: example-configmap-1-8mbdf7882g
```

ConfigMap can also be generated from literal key-value pairs. To generate a ConfigMap from a literal key-value pair, add an entry to `literals` list in `configMapGenerator`. Here is an example of generating a ConfigMap with a data item from a key-value pair.

```
cat <<EOF >./kustomization.yaml
configMapGenerator:
- name: example-configmap-2
  literals:
    - FOO=Bar
EOF
```

The generated ConfigMap can be checked by the following command:

```
kubectl kustomize ./
```

The generated ConfigMap is

```
apiVersion: v1
data:
  FOO: Bar
```

```
kind: ConfigMap
metadata:
  name: example-configmap-2-g2hdhfc6tk
```

## secretGenerator

You can generate Secrets from files or literal key-value pairs. To generate a Secret from a file, add an entry to `files` list in `secretGenerator`. Here is an example of generating a Secret with a data item from a file.

```
# Create a password.txt file
cat <<EOF >./password.txt
username=admin
password=secret
EOF

cat <<EOF >./kustomization.yaml
secretGenerator:
- name: example-secret-1
  files:
    - password.txt
EOF
```

The generated Secret is as follows:

```
apiVersion: v1
data:
  password.txt: dXNlcm5hbWU9YWRtaW4KcGFzc3dvcmQ9c2VjcmV0Cg==
kind: Secret
metadata:
  name: example-secret-1-t2kt65hgtb
type: Opaque
```

To generate a Secret from a literal key-value pair, add an entry to `literals` list in `secretGenerator`. Here is an example of generating a Secret with a data item from a key-value pair.

```
cat <<EOF >./kustomization.yaml
secretGenerator:
- name: example-secret-2
  literals:
    - username=admin
    - password=secret
EOF
```

The generated Secret is as follows:

```
apiVersion: v1
data:
  password: c2VjcmV0
  username: YWRtaW4=
kind: Secret
```

```
metadata:  
  name: example-secret-2-t52t6g96d8  
type: Opaque
```

## generatorOptions

The generated ConfigMaps and Secrets have a suffix appended by hashing the contents. This ensures that a new ConfigMap or Secret is generated when the content is changed. To disable the behavior of appending a suffix, one can use `generatorOptions`. Besides that, it is also possible to specify cross-cutting options for generated ConfigMaps and Secrets.

```
cat <<EOF >./kustomization.yaml  
configMapGenerator:  
- name: example-configmap-3  
  literals:  
  - FOO=Bar  
generatorOptions:  
  disableNameSuffixHash: true  
  labels:  
    type: generated  
  annotations:  
    note: generated  
EOF
```

Run `kubectl kustomize ./` to view the generated ConfigMap:

```
apiVersion: v1  
data:  
  FOO: Bar  
kind: ConfigMap  
metadata:  
  annotations:  
    note: generated  
  labels:  
    type: generated  
  name: example-configmap-3
```

## Setting cross-cutting fields

It is quite common to set cross-cutting fields for all Kubernetes resources in a project. Some use cases for setting cross-cutting fields:

- setting the same namespace for all Resource
- adding the same name prefix or suffix
- adding the same set of labels
- adding the same set of annotations

Here is an example:

```
# Create a deployment.yaml  
cat <<EOF >./deployment.yaml
```

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deployment
  labels:
    app: nginx
spec:
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
        - name: nginx
          image: nginx
EOF
```

```
cat <<EOF >./kustomization.yaml
namespace: my-namespace
namePrefix: dev-
nameSuffix: "-001"
commonLabels:
  app: bingo
commonAnnotations:
  oncallPager: 800-555-1212
resources:
- deployment.yaml
EOF
```

Run `kubectl kustomize ./` to view those fields are all set in the Deployment Resource:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  annotations:
    oncallPager: 800-555-1212
  labels:
    app: bingo
  name: dev-nginx-deployment-001
  namespace: my-namespace
spec:
  selector:
    matchLabels:
      app: bingo
  template:
    metadata:
      annotations:
        oncallPager: 800-555-1212
```

```
  labels:
    app: bingo
  spec:
    containers:
      - image: nginx
        name: nginx
```

## Composing and Customizing Resources

It is common to compose a set of Resources in a project and manage them inside the same file or directory. Kustomize offers composing Resources from different files and applying patches or other customization to them.

### Composing

Kustomize supports composition of different resources. The `resources` field, in the `kustomization.yaml` file, defines the list of resources to include in a configuration. Set the path to a resource's configuration file in the `resources` list. Here is an example for an nginx application with a Deployment and a Service.

```
# Create a deployment.yaml file
cat <<EOF > deployment.yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: my-nginx
spec:
  selector:
    matchLabels:
      run: my-nginx
  replicas: 2
  template:
    metadata:
      labels:
        run: my-nginx
    spec:
      containers:
        - name: my-nginx
          image: nginx
          ports:
            - containerPort: 80
EOF

# Create a service.yaml file
cat <<EOF > service.yaml
apiVersion: v1
kind: Service
metadata:
  name: my-nginx
  labels:
```

```

    run: my-nginx
spec:
  ports:
    - port: 80
      protocol: TCP
  selector:
    run: my-nginx
EOF

# Create a kustomization.yaml composing them
cat <<EOF >./kustomization.yaml
resources:
- deployment.yaml
- service.yaml
EOF

```

The Resources from `kubectl kustomize ./` contains both the Deployment and the Service objects.

## Customizing

On top of Resources, one can apply different customizations by applying patches. Kustomize supports different patching mechanisms through `patchesStrategicMerge` and `patchesJson6902`. `patchesStrategicMerge` is a list of file paths. Each file should be resolved to a [strategic merge patch](#). The names inside the patches must match Resource names that are already loaded. Small patches that do one thing are recommended. For example, create one patch for increasing the deployment replica number and another patch for setting the memory limit.

```

# Create a deployment.yaml file
cat <<EOF > deployment.yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: my-nginx
spec:
  selector:
    matchLabels:
      run: my-nginx
  replicas: 2
  template:
    metadata:
      labels:
        run: my-nginx
  spec:
    containers:
      - name: my-nginx
        image: nginx
        ports:
          - containerPort: 80
EOF

```

```

# Create a patch increase_replicas.yaml
cat <<EOF > increase_replicas.yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: my-nginx
spec:
  replicas: 3
EOF

# Create another patch set_memory.yaml
cat <<EOF > set_memory.yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: my-nginx
spec:
  template:
    spec:
      containers:
        - name: my-nginx
          resources:
            limits:
              memory: 512Mi
EOF

cat <<EOF >./kustomization.yaml
resources:
- deployment.yaml
patchesStrategicMerge:
- increase_replicas.yaml
- set_memory.yaml
EOF

```

Run `kubectl kustomize ./` to view the Deployment:

```

apiVersion: apps/v1
kind: Deployment
metadata:
  name: my-nginx
spec:
  replicas: 3
  selector:
    matchLabels:
      run: my-nginx
  template:
    metadata:
      labels:
        run: my-nginx
    spec:
      containers:

```

```
- image: nginx
limits:
  memory: 512Mi
name: my-nginx
ports:
- containerPort: 80
```

Not all Resources or fields support strategic merge patches. To support modifying arbitrary fields in arbitrary Resources, Kustomize offers applying [JSON patch](#) through `patchesJson6902`. To find the correct Resource for a Json patch, the group, version, kind and name of that Resource need to be specified in `kustomization.yaml`. For example, increasing the replica number of a Deployment object can also be done through `patchesJson6902`.

```
# Create a deployment.yaml file
cat <<EOF > deployment.yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: my-nginx
spec:
  selector:
    matchLabels:
      run: my-nginx
  replicas: 2
  template:
    metadata:
      labels:
        run: my-nginx
  spec:
    containers:
    - name: my-nginx
      image: nginx
      ports:
      - containerPort: 80
EOF
```

```
# Create a json patch
cat <<EOF > patch.yaml
- op: replace
  path: /spec/relicas
  value: 3
EOF

# Create a kustomization.yaml
cat <<EOF >./kustomization.yaml
resources:
- deployment.yaml

patchesJson6902:
- target:
    group: apps
```

```
version: v1
kind: Deployment
name: my-nginx
path: patch.yaml
EOF
```

Run `kubectl kustomize ./` to see the `replicas` field is updated:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: my-nginx
spec:
  replicas: 3
  selector:
    matchLabels:
      run: my-nginx
  template:
    metadata:
      labels:
        run: my-nginx
    spec:
      containers:
        - image: nginx
          name: my-nginx
          ports:
            - containerPort: 80
```

In addition to patches, Kustomize also offers customizing container images or injecting field values from other objects into containers without creating patches. For example, you can change the image used inside containers by specifying the new image in `images` field in `kustomization.yaml`.

```
cat <<EOF > deployment.yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: my-nginx
spec:
  selector:
    matchLabels:
      run: my-nginx
  replicas: 2
  template:
    metadata:
      labels:
        run: my-nginx
    spec:
      containers:
        - name: my-nginx
          image: nginx
          ports:
```

```

        - containerPort: 80
EOF

cat <<EOF >./kustomization.yaml
resources:
- deployment.yaml
images:
- name: nginx
  newName: my.image.registry/nginx
  newTag: 1.4.0
EOF

```

Run `kubectl kustomize ./` to see that the image being used is updated:

```

apiVersion: apps/v1
kind: Deployment
metadata:
  name: my-nginx
spec:
  replicas: 2
  selector:
    matchLabels:
      run: my-nginx
  template:
    metadata:
      labels:
        run: my-nginx
    spec:
      containers:
        - image: my.image.registry/nginx:1.4.0
          name: my-nginx
          ports:
            - containerPort: 80

```

Sometimes, the application running in a Pod may need to use configuration values from other objects. For example, a Pod from a Deployment object need to read the corresponding Service name from Env or as a command argument. Since the Service name may change as `namePrefix` or `nameSuffix` is added in the `kustomization.yaml` file. It is not recommended to hard code the Service name in the command argument. For this usage, Kustomize can inject the Service name into containers through `vars`.

```

# Create a deployment.yaml file
cat <<EOF > deployment.yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: my-nginx
spec:
  selector:
    matchLabels:
      run: my-nginx

```

```

replicas: 2
template:
  metadata:
    labels:
      run: my-nginx
spec:
  containers:
    - name: my-nginx
      image: nginx
      command: ["start", "--host", "\$(MY_SERVICE_NAME)"]
EOF

# Create a service.yaml file
cat <<EOF > service.yaml
apiVersion: v1
kind: Service
metadata:
  name: my-nginx
  labels:
    run: my-nginx
spec:
  ports:
    - port: 80
      protocol: TCP
  selector:
    run: my-nginx
EOF

cat <<EOF >./kustomization.yaml
namePrefix: dev-
nameSuffix: "-001"

resources:
- deployment.yaml
- service.yaml

vars:
- name: MY_SERVICE_NAME
  objref:
    kind: Service
    name: my-nginx
    apiVersion: v1
EOF

```

Run `kubectl kustomize ./` to see that the Service name injected into containers is `dev-my-nginx-001`:

```

apiVersion: apps/v1
kind: Deployment
metadata:
  name: dev-my-nginx-001
spec:

```

```
replicas: 2
selector:
  matchLabels:
    run: my-nginx
template:
  metadata:
    labels:
      run: my-nginx
spec:
  containers:
    - command:
      - start
      - --host
      - dev-my-nginx-001
    image: nginx
    name: my-nginx
```

## Bases and Overlays

Kustomize has the concepts of **bases** and **overlays**. A **base** is a directory with a `kustomization.yaml`, which contains a set of resources and associated customization. A base could be either a local directory or a directory from a remote repo, as long as a `kustomization.yaml` is present inside. An **overlay** is a directory with a `kustomization.yaml` that refers to other kustomization directories as its bases. A **base** has no knowledge of an overlay and can be used in multiple overlays. An overlay may have multiple bases and it composes all resources from bases and may also have customization on top of them.

Here is an example of a base:

```
# Create a directory to hold the base
mkdir base
# Create a base/deployment.yaml
cat <<EOF > base/deployment.yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: my-nginx
spec:
  selector:
    matchLabels:
      run: my-nginx
  replicas: 2
  template:
    metadata:
      labels:
        run: my-nginx
  spec:
    containers:
      - name: my-nginx
```

```

        image: nginx
EOF

# Create a base/service.yaml file
cat <<EOF > base/service.yaml
apiVersion: v1
kind: Service
metadata:
  name: my-nginx
  labels:
    run: my-nginx
spec:
  ports:
  - port: 80
    protocol: TCP
  selector:
    run: my-nginx
EOF
# Create a base/kustomization.yaml
cat <<EOF > base/kustomization.yaml
resources:
- deployment.yaml
- service.yaml
EOF

```

This base can be used in multiple overlays. You can add different namePrefix or other cross-cutting fields in different overlays. Here are two overlays using the same base.

```

mkdir dev
cat <<EOF > dev/kustomization.yaml
bases:
- ../base
namePrefix: dev-
EOF

mkdir prod
cat <<EOF > prod/kustomization.yaml
bases:
- ../base
namePrefix: prod-
EOF

```

## How to apply/view/delete objects using Kustomize

Use --kustomize or -k in kubectl commands to recognize Resources managed by kustomization.yaml. Note that -k should point to a kustomization directory, such as

```
kubectl apply -k <kustomization directory>/
```

Given the following `kustomization.yaml`,

```
# Create a deployment.yaml file
cat <<EOF > deployment.yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: my-nginx
spec:
  selector:
    matchLabels:
      run: my-nginx
  replicas: 2
  template:
    metadata:
      labels:
        run: my-nginx
    spec:
      containers:
        - name: my-nginx
          image: nginx
          ports:
            - containerPort: 80
EOF
```

```
# Create a kustomization.yaml
cat <<EOF >./kustomization.yaml
namePrefix: dev-
commonLabels:
  app: my-nginx
resources:
- deployment.yaml
EOF
```

Run the following command to apply the Deployment object `dev-my-nginx`:

```
> kubectl apply -k ./  
deployment.apps/dev-my-nginx created
```

Run one of the following commands to view the Deployment object `dev-my-nginx`:

```
kubectl get -k ./
```

```
kubectl describe -k ./
```

Run the following command to delete the Deployment object `dev-my-nginx`:

```
> kubectl delete -k ./  
deployment.apps "dev-my-nginx" deleted
```

# Kustomize Feature List

Field	Type	Explanation
namespace	string	add namespace to all resources
namePrefix	string	value of this field is prepended to the names of all resources
nameSuffix	string	value of this field is appended to the names of all resources
commonLabels	map[string]string	labels to add to all resources and selectors
commonAnnotations	map[string]string	annotations to add to all resources
resources	[]string	each entry in this list must resolve to an existing resource configuration file
configmapGenerator	[] <a href="#">ConfigMapArgs</a>	Each entry in this list generates a ConfigMap
secretGenerator	[] <a href="#">SecretArgs</a>	Each entry in this list generates a Secret
generatorOptions	<a href="#">GeneratorOptions</a>	Modify behaviors of all ConfigMap and Secret generator
bases	[]string	Each entry in this list should resolve to a directory containing a kustomization.yaml file
patchesStrategicMerge	[]string	Each entry in this list should resolve a strategic merge patch of a Kubernetes object
patchesJson6902	[] <a href="#">Json6902</a>	Each entry in this list should resolve to a Kubernetes object and a Json Patch
vars	[] <a href="#">Var</a>	Each entry is to capture text from one resource's field
images	[] <a href="#">Image</a>	Each entry is to modify the name, tags and/or digest for one image without creating patches
configurations	[]string	Each entry in this list should resolve to a file containing <a href="#">Kustomize transformer configurations</a>
crds	[]string	Each entry in this list should resolve to an OpenAPI definition file for Kubernetes types

## What's next

- [Kustomize](#)
- [Kubectl Book](#)
- [Kubectl Command Reference](#)

- [Kubernetes API Reference](#)

## Feedback

Was this page helpful?

Yes No

Thanks for the feedback. If you have a specific, answerable question about how to use Kubernetes, ask it on [Stack Overflow](#). Open an issue in the GitHub repo if you want to [report a problem](#) or [suggest an improvement](#).

---

[Create an Issue](#) [Edit This Page](#)

Page last modified on December 06, 2019 at 7:24 PM PST by [fix-up 404 urls \(#17955\)](#) ([Page History](#))

[Edit This Page](#)

# Managing Kubernetes Objects Using Imperative Commands

Kubernetes objects can quickly be created, updated, and deleted directly using imperative commands built into the `kubectl` command-line tool. This document explains how those commands are organized and how to use them to manage live objects.

- [Before you begin](#)
- [Trade-offs](#)
- [How to create objects](#)
- [How to update objects](#)
- [How to delete objects](#)
- [How to view an object](#)
- [Using `set` commands to modify objects before creation](#)
- [Using `--edit` to modify objects before creation](#)
- [What's next](#)

## Before you begin

Install [kubectl](#).

You need to have a Kubernetes cluster, and the `kubectl` command-line tool must be configured to communicate with your cluster. 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`.

## Trade-offs

The `kubectl` tool supports three kinds of object management:

- Imperative commands
- Imperative object configuration
- Declarative object configuration

See [Kubernetes Object Management](#) for a discussion of the advantages and disadvantage of each kind of object management.

# How to create objects

The `kubectl` tool supports verb-driven commands for creating some of the most common object types. The commands are named to be recognizable to users unfamiliar with the Kubernetes object types.

- `run`: Create a new Deployment object to run Containers in one or more Pods.
- `expose`: Create a new Service object to load balance traffic across Pods.
- `autoscale`: Create a new Autoscaler object to automatically horizontally scale a controller, such as a Deployment.

The `kubectl` tool also supports creation commands driven by object type. These commands support more object types and are more explicit about their intent, but require users to know the type of objects they intend to create.

- `create <objecttype> [<subtype>] <instancename>`

Some objects types have subtypes that you can specify in the `create` command. For example, the Service object has several subtypes including ClusterIP, LoadBalancer, and NodePort. Here's an example that creates a Service with subtype NodePort:

```
kubectl create service nodeport <myservicename>
```

In the preceding example, the `create service nodeport` command is called a subcommand of the `create service` command.

You can use the `-h` flag to find the arguments and flags supported by a subcommand:

```
kubectl create service nodeport -h
```

# How to update objects

The `kubectl` command supports verb-driven commands for some common update operations. These commands are named to enable users unfamiliar with Kubernetes objects to perform updates without knowing the specific fields that must be set:

- `scale`: Horizontally scale a controller to add or remove Pods by updating the replica count of the controller.
- `annotate`: Add or remove an annotation from an object.
- `label`: Add or remove a label from an object.

The `kubectl` command also supports update commands driven by an aspect of the object. Setting this aspect may set different fields for different object types:

- `set <field>`: Set an aspect of an object.

**Note:** In Kubernetes version 1.5, not every verb-driven command has an associated aspect-driven command.

The `kubectl` tool supports these additional ways to update a live object directly, however they require a better understanding of the Kubernetes object schema.

- `edit`: Directly edit the raw configuration of a live object by opening its configuration in an editor.
- `patch`: Directly modify specific fields of a live object by using a patch string. For more details on patch strings, see the patch section in [API Conventions](#).

## How to delete objects

You can use the `delete` command to delete an object from a cluster:

- `delete <type>/<name>`

**Note:** You can use `kubectl delete` for both imperative commands and imperative object configuration. The difference is in the arguments passed to the command. To use `kubectl delete` as an imperative command, pass the object to be deleted as an argument. Here's an example that passes a Deployment object named `nginx`:

```
kubectl delete deployment/nginx
```

## How to view an object

There are several commands for printing information about an object:

- `get`: Prints basic information about matching objects. Use `get -h` to see a list of options.
- `describe`: Prints aggregated detailed information about matching objects.
- `logs`: Prints the `stdout` and `stderr` for a container running in a Pod.

## Using set commands to modify objects before creation

There are some object fields that don't have a flag you can use in a `create` command. In some of those cases, you can use a combination of `set` and `create` to specify a value for the field before object creation. This is done by piping the output of the `create` command to the `set` command, and then back to the `create` command. Here's an example:

```
kubectl create service clusterip my-svc --clusterip="None" -o yaml --dry-run | kubectl set selector --local -f - 'environment=q a' -o yaml | kubectl create -f -
```

1. The `kubectl create service -o yaml --dry-run` command creates the configuration for the Service, but prints it to stdout as YAML instead of sending it to the Kubernetes API server.
2. The `kubectl set selector --local -f - -o yaml` command reads the configuration from stdin, and writes the updated configuration to stdout as YAML.
3. The `kubectl create -f -` command creates the object using the configuration provided via stdin.

## Using `--edit` to modify objects before creation

You can use `kubectl create --edit` to make arbitrary changes to an object before it is created. Here's an example:

```
kubectl create service clusterip my-svc --clusterip="None" -o yaml --dry-run > /tmp/srv.yaml  
kubectl create --edit -f /tmp/srv.yaml
```

1. The `kubectl create service` command creates the configuration for the Service and saves it to `/tmp/srv.yaml`.
2. The `kubectl create --edit` command opens the configuration file for editing before it creates the object.

## What's next

- [Managing Kubernetes Objects Using Object Configuration \(Imperative\)](#)
- [Managing Kubernetes Objects Using Object Configuration \(Declarative\)](#)
- [Kubectl Command Reference](#)
- [Kubernetes API Reference](#)

## Feedback

Was this page helpful?

Yes No

Thanks for the feedback. If you have a specific, answerable question about how to use Kubernetes, ask it on [Stack Overflow](#). Open an issue in the GitHub repo if you want to [report a problem](#) or [suggest an improvement](#).

---

[Create an Issue](#) [Edit This Page](#)

Page last modified on May 21, 2019 at 3:48 AM PST by [fix links to tool install \(#14418\)](#) ([Page History](#))

[Edit This Page](#)

# Imperative Management of Kubernetes Objects Using Configuration Files

Kubernetes objects can be created, updated, and deleted by using the `kubectl` command-line tool along with an object configuration file written in YAML

or JSON. This document explains how to define and manage objects using configuration files.

- [Before you begin](#)
- [Trade-offs](#)
- [How to create objects](#)
- [How to update objects](#)
- [How to delete objects](#)
- [How to view an object](#)
- [Limitations](#)
- [Creating and editing an object from a URL without saving the configuration](#)
- [Migrating from imperative commands to imperative object configuration](#)
- [Defining controller selectors and PodTemplate labels](#)
- [What's next](#)

## Before you begin

Install [kubectl](#).

You need to have a Kubernetes cluster, and the kubectl command-line tool must be configured to communicate with your cluster. 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`.

## Trade-offs

The kubectl tool supports three kinds of object management:

- Imperative commands
- Imperative object configuration
- Declarative object configuration

See [Kubernetes Object Management](#) for a discussion of the advantages and disadvantage of each kind of object management.

## How to create objects

You can use `kubectl create -f` to create an object from a configuration file. Refer to the [kubernetes API reference](#) for details.

- `kubectl create -f <filename|url>`

# How to update objects

**Warning:** Updating objects with the `replace` command drops all parts of the spec not specified in the configuration file. This should not be used with objects whose specs are partially managed by the cluster, such as Services of type `LoadBalancer`, where the `externalIPs` field is managed independently from the configuration file. Independently managed fields must be copied to the configuration file to prevent `replace` from dropping them.

You can use `kubectl replace -f` to update a live object according to a configuration file.

- `kubectl replace -f <filename|url>`

# How to delete objects

You can use `kubectl delete -f` to delete an object that is described in a configuration file.

- `kubectl delete -f <filename|url>`

# How to view an object

You can use `kubectl get -f` to view information about an object that is described in a configuration file.

- `kubectl get -f <filename|url> -o yaml`

The `-o yaml` flag specifies that the full object configuration is printed. Use `kubectl get -h` to see a list of options.

# Limitations

The `create`, `replace`, and `delete` commands work well when each object's configuration is fully defined and recorded in its configuration file. However when a live object is updated, and the updates are not merged into its configuration file, the updates will be lost the next time a `replace` is executed. This can happen if a controller, such as a `HorizontalPodAutoscaler`, makes updates directly to a live object. Here's an example:

1. You create an object from a configuration file.
2. Another source updates the object by changing some field.
3. You replace the object from the configuration file. Changes made by the other source in step 2 are lost.

If you need to support multiple writers to the same object, you can use `kubectl apply` to manage the object.

# Creating and editing an object from a URL without saving the configuration

Suppose you have the URL of an object configuration file. You can use `kubectl create --edit` to make changes to the configuration before the object is created. This is particularly useful for tutorials and tasks that point to a configuration file that could be modified by the reader.

```
kubectl create -f <url> --edit
```

# Migrating from imperative commands to imperative object configuration

Migrating from imperative commands to imperative object configuration involves several manual steps.

1. Export the live object to a local object configuration file:

```
kubectl get <kind>/<name> -o yaml > <kind>_<name>.yaml
```

2. Manually remove the status field from the object configuration file.

3. For subsequent object management, use `replace` exclusively.

```
kubectl replace -f <kind>_<name>.yaml
```

# Defining controller selectors and PodTemplate labels

**Warning:** Updating selectors on controllers is strongly discouraged.

The recommended approach is to define a single, immutable PodTemplate label used only by the controller selector with no other semantic meaning.

Example label:

```
selector:
  matchLabels:
    controller-selector: "apps/v1/deployment/nginx"
template:
  metadata:
    labels:
      controller-selector: "apps/v1/deployment/nginx"
```

## What's next

- [Managing Kubernetes Objects Using Imperative Commands](#)
- [Managing Kubernetes Objects Using Object Configuration \(Declarative\)](#)

- [Kubectl Command Reference](#)
- [Kubernetes API Reference](#)

## Feedback

Was this page helpful?

Yes No

Thanks for the feedback. If you have a specific, answerable question about how to use Kubernetes, ask it on [Stack Overflow](#). Open an issue in the GitHub repo if you want to [report a problem](#) or [suggest an improvement](#).

---

[Create an Issue](#) [Edit This Page](#)

Page last modified on January 27, 2020 at 7:23 AM PST by [Clean up extensions/v1beta1 in docs \(#18838\)](#) ([Page History](#))

[Edit This Page](#)

# Define a Command and Arguments for a Container

This page shows how to define commands and arguments when you run a container in a [Pod](#)[The smallest and simplest Kubernetes object. A Pod represents a set of running containers on your cluster.](#).

- [Before you begin](#)
- [Define a command and arguments when you create a Pod](#)
- [Use environment variables to define arguments](#)
- [Run a command in a shell](#)
- [Notes](#)
- [What's next](#)

## Before you begin

You need to have a Kubernetes cluster, and the kubectl command-line tool must be configured to communicate with your cluster. 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`.

## Define a command and arguments when you create a Pod

When you create a Pod, you can define a command and arguments for the containers that run in the Pod. To define a command, include the `command` field in the configuration file. To define arguments for the command, include the `args` field in the configuration file. The command and arguments that you define cannot be changed after the Pod is created.

The command and arguments that you define in the configuration file override the default command and arguments provided by the container image. If you define `args`, but do not define a `command`, the default command is used with your new arguments.

**Note:** The `command` field corresponds to `entrypoint` in some container runtimes. Refer to the [Notes](#) below.

In this exercise, you create a Pod that runs one container. The configuration file for the Pod defines a command and two arguments:

### [pods/commands.yaml](#)

```
apiVersion: v1
kind: Pod
metadata:
  name: command-demo
  labels:
    purpose: demonstrate-command
spec:
  containers:
    - name: command-demo-container
      image: debian
      command: ["printenv"]
      args: ["HOSTNAME", "KUBERNETES_PORT"]
  restartPolicy: OnFailure
```

1. Create a Pod based on the YAML configuration file:

```
kubectl apply -f https://k8s.io/examples/pods/commands.yaml
```

2. List the running Pods:

```
kubectl get pods
```

The output shows that the container that ran in the command-demo Pod has completed.

3. To see the output of the command that ran in the container, view the logs from the Pod:

```
kubectl logs command-demo
```

The output shows the values of the HOSTNAME and KUBERNETES\_PORT environment variables:

```
command-demo
tcp://10.3.240.1:443
```

## Use environment variables to define arguments

In the preceding example, you defined the arguments directly by providing strings. As an alternative to providing strings directly, you can define arguments by using environment variables:

```
env:
- name: MESSAGE
  value: "hello world"
command: ["/bin/echo"]
args: ["$(MESSAGE)"]
```

This means you can define an argument for a Pod using any of the techniques available for defining environment variables, including [ConfigMaps](#) and [Secrets](#).

**Note:** The environment variable appears in parentheses, "\$ (VAR)". This is required for the variable to be expanded in the command or args field.

## Run a command in a shell

In some cases, you need your command to run in a shell. For example, your command might consist of several commands piped together, or it might be a shell script. To run your command in a shell, wrap it like this:

```
command: ["/bin/sh"]
args: ["-c", "while true; do echo hello; sleep 10;done"]
```

## Notes

This table summarizes the field names used by Docker and Kubernetes.

Description	Docker field name	Kubernetes field name
The command run by the container	Entrypoint	command
The arguments passed to the command	Cmd	args

When you override the default Entrypoint and Cmd, these rules apply:

- If you do not supply command or args for a Container, the defaults defined in the Docker image are used.
- If you supply a command but no args for a Container, only the supplied command is used. The default EntryPoint and the default Cmd defined in the Docker image are ignored.
- If you supply only args for a Container, the default Entrypoint defined in the Docker image is run with the args that you supplied.
- If you supply a command and args, the default Entrypoint and the default Cmd defined in the Docker image are ignored. Your command is run with your args.

Here are some examples:

Image Entrypoint	Image Cmd	Container command	Container args	Command run
[/ep-1]	[foo bar]	<not set>	<not set>	[ep-1 foo bar]
[/ep-1]	[foo bar]	[/ep-2]	<not set>	[ep-2]

<b>Image Entrypoint</b>	<b>Image Cmd</b>	<b>Container command</b>	<b>Container args</b>	<b>Command run</b>
[/ep-1]	[foo bar]	<not set>	[zoo boo]	[ep-1 zoo boo]
[/ep-1]	[foo bar]	[/ep-2]	[zoo boo]	[ep-2 zoo boo]

## What's next

- Learn more about [configuring pods and containers](#).
- Learn more about [running commands in a container](#).
- See [Container](#).

## Feedback

Was this page helpful?

Yes No

Thanks for the feedback. If you have a specific, answerable question about how to use Kubernetes, ask it on [Stack Overflow](#). Open an issue in the GitHub repo if you want to [report a problem](#) or [suggest an improvement](#).

---

[Create an Issue](#) [Edit This Page](#)

Page last modified on March 25, 2019 at 5:06 PM PST by [Official 1.14](#)  
[Release Docs \(#13174\)](#) ([Page History](#))

[Edit This Page](#)

# Define Environment Variables for a Container

This page shows how to define environment variables for a container in a Kubernetes Pod.

- [Before you begin](#)
- [Define an environment variable for a container](#)
- [Using environment variables inside of your config](#)

- [What's next](#)

## Before you begin

You need to have a Kubernetes cluster, and the `kubectl` command-line tool must be configured to communicate with your cluster. 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`.

## Define an environment variable for a container

When you create a Pod, you can set environment variables for the containers that run in the Pod. To set environment variables, include the `env` or `envFrom` field in the configuration file.

In this exercise, you create a Pod that runs one container. The configuration file for the Pod defines an environment variable with name `DEMO_GREETING` and value "Hello from the environment". Here is the configuration file for the Pod:

### [pods/inject/envvars.yaml](#)

```
apiVersion: v1
kind: Pod
metadata:
  name: envar-demo
  labels:
    purpose: demonstrate-envvars
spec:
  containers:
    - name: envar-demo-container
      image: gcr.io/google-samples/node-hello:1.0
      env:
        - name: DEMO_GREETING
          value: "Hello from the environment"
        - name: DEMO_FAREWELL
          value: "Such a sweet sorrow"
```

1. Create a Pod based on the YAML configuration file:

```
kubectl apply -f https://k8s.io/examples/pods/inject/
envvars.yaml
```

2. List the running Pods:

```
kubectl get pods -l purpose=demonstrate-envars
```

The output is similar to this:

NAME	READY	STATUS	RESTARTS	AGE
envar-demo	1/1	Running	0	9s

3. Get a shell to the container running in your Pod:

```
kubectl exec -it envar-demo -- /bin/bash
```

4. In your shell, run the `printenv` command to list the environment variables.

```
root@envar-demo:/# printenv
```

The output is similar to this:

```
NODE_VERSION=4.4.2
EXAMPLE_SERVICE_PORT_8080_TCP_ADDR=10.3.245.237
HOSTNAME=envar-demo
...
DEMO_GREETING=Hello from the environment
DEMO_FAREWELL=Such a sweet sorrow
```

5. To exit the shell, enter `exit`.

**Note:** The environment variables set using the `env` or `envFrom` field will override any environment variables specified in the container image.

## Using environment variables inside of your config

Environment variables that you define in a Pod's configuration can be used elsewhere in the configuration, for example in commands and arguments that you set for the Pod's containers. In the example configuration below, the `GREETING`, `HONORIFIC`, and `NAME` environment variables are set to `Warm greetings to, The Most Honorable, and Kubernetes`, respectively. Those environment variables are then used in the CLI arguments passed to the `env-print-demo` container.

```
apiVersion: v1
kind: Pod
metadata:
  name: print-greeting
spec:
  containers:
    - name: env-print-demo
      image: bash
```

```
env:  
  - name: GREETING  
    value: "Warm greetings to"  
  - name: HONORIFIC  
    value: "The Most Honorable"  
  - name: NAME  
    value: "Kubernetes"  
  command: ["echo"]  
  args: ["$(GREETING) $(HONORIFIC) $(NAME)"]
```

Upon creation, the command echo Warm greetings to The Most Honorable Kubernetes is run on the container.

## What's next

- Learn more about [environment variables](#).
- Learn about [using secrets as environment variables](#).
- See [EnvVarSource](#).

## Feedback

Was this page helpful?

Yes No

Thanks for the feedback. If you have a specific, answerable question about how to use Kubernetes, ask it on [Stack Overflow](#). Open an issue in the GitHub repo if you want to [report a problem](#) or [suggest an improvement](#).

---

[Create an Issue](#) [Edit This Page](#)

Page last modified on March 25, 2019 at 5:06 PM PST by [Official 1.14](#)  
[Release Docs \(#13174\)](#) ([Page History](#))

[Edit This Page](#)

# Expose Pod Information to Containers Through Environment Variables

This page shows how a Pod can use environment variables to expose information about itself to Containers running in the Pod. Environment variables can expose Pod fields and Container fields.

- [Before you begin](#)
- [The Downward API](#)
- [Use Pod fields as values for environment variables](#)
- [Use Container fields as values for environment variables](#)
- [What's next](#)

## Before you begin

You need to have a Kubernetes cluster, and the `kubectl` command-line tool must be configured to communicate with your cluster. 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`.

## The Downward API

There are two ways to expose Pod and Container fields to a running Container:

- Environment variables
- [Volume Files](#)

Together, these two ways of exposing Pod and Container fields are called the *Downward API*.

## Use Pod fields as values for environment variables

In this exercise, you create a Pod that has one Container. Here is the configuration file for the Pod:

## [pods/inject/dapi-envars-pod.yaml](#)

```
apiVersion: v1
kind: Pod
metadata:
  name: dapi-envars-fieldref
spec:
  containers:
    - name: test-container
      image: k8s.gcr.io/busybox
      command: [ "sh", "-c" ]
      args:
        - while true; do
            echo -en '\n';
            printenv MY_NODE_NAME MY_POD_NAME MY_POD_NAMESPACE;
            printenv MY_POD_IP MY_POD_SERVICE_ACCOUNT;
            sleep 10;
        done;
  env:
    - name: MY_NODE_NAME
      valueFrom:
        fieldRef:
          fieldPath: spec.nodeName
    - name: MY_POD_NAME
      valueFrom:
        fieldRef:
          fieldPath: metadata.name
    - name: MY_POD_NAMESPACE
      valueFrom:
        fieldRef:
          fieldPath: metadata.namespace
    - name: MY_POD_IP
      valueFrom:
        fieldRef:
          fieldPath: status.podIP
    - name: MY_POD_SERVICE_ACCOUNT
      valueFrom:
        fieldRef:
          fieldPath: spec.serviceAccountName
  restartPolicy: Never
```

In the configuration file, you can see five environment variables. The `env` field is an array of [EnvVars](#). The first element in the array specifies that the `MY_NODE_NAME` environment variable gets its value from the Pod's `spec.nodeName` field. Similarly, the other environment variables get their names from Pod fields.

**Note:** The fields in this example are Pod fields. They are not fields of the Container in the Pod.

Create the Pod:

```
kubectl apply -f https://k8s.io/examples/pods/inject/dapi-envvars-pod.yaml
```

Verify that the Container in the Pod is running:

```
kubectl get pods
```

View the Container's logs:

```
kubectl logs dapi-envvars-fieldref
```

The output shows the values of selected environment variables:

```
minikube
dapi-envvars-fieldref
default
172.17.0.4
default
```

To see why these values are in the log, look at the `command` and `args` fields in the configuration file. When the Container starts, it writes the values of five environment variables to stdout. It repeats this every ten seconds.

Next, get a shell into the Container that is running in your Pod:

```
kubectl exec -it dapi-envvars-fieldref -- sh
```

In your shell, view the environment variables:

```
/# printenv
```

The output shows that certain environment variables have been assigned the values of Pod fields:

```
MY_POD_SERVICE_ACCOUNT=default
...
MY_POD_NAMESPACE=default
MY_POD_IP=172.17.0.4
...
MY_NODE_NAME=minikube
...
MY_POD_NAME=dapi-envvars-fieldref
```

## Use Container fields as values for environment variables

In the preceding exercise, you used Pod fields as the values for environment variables. In this next exercise, you use Container fields as the values for environment variables. Here is the configuration file for a Pod that has one container:

## [`pods/inject/dapi-envars-container.yaml`](#)

```
apiVersion: v1
kind: Pod
metadata:
  name: dapi-envars-resourcefieldref
spec:
  containers:
    - name: test-container
      image: k8s.gcr.io/busybox:1.24
      command: [ "sh", "-c"]
      args:
        - while true; do
          echo -en '\n';
          printenv MY_CPU_REQUEST MY_CPU_LIMIT;
          printenv MY_MEM_REQUEST MY_MEM_LIMIT;
          sleep 10;
        done;
      resources:
        requests:
          memory: "32Mi"
          cpu: "125m"
        limits:
          memory: "64Mi"
          cpu: "250m"
    env:
      - name: MY_CPU_REQUEST
        valueFrom:
          resourceFieldRef:
            containerName: test-container
            resource: requests.cpu
      - name: MY_CPU_LIMIT
        valueFrom:
          resourceFieldRef:
            containerName: test-container
            resource: limits.cpu
      - name: MY_MEM_REQUEST
        valueFrom:
          resourceFieldRef:
            containerName: test-container
            resource: requests.memory
      - name: MY_MEM_LIMIT
        valueFrom:
          resourceFieldRef:
            containerName: test-container
            resource: limits.memory
  restartPolicy: Never
```

In the configuration file, you can see four environment variables. The `env` field is an array of [EnvVars](#). The first element in the array specifies that the

`MY_CPU_REQUEST` environment variable gets its value from the `requests.cpu` field of a Container named `test-container`. Similarly, the other environment variables get their values from Container fields.

Create the Pod:

```
kubectl apply -f https://k8s.io/examples/pods/inject/dapi-envvars-container.yaml
```

Verify that the Container in the Pod is running:

```
kubectl get pods
```

View the Container's logs:

```
kubectl logs dapi-envvars-resourcefieldref
```

The output shows the values of selected environment variables:

```
1  
1  
33554432  
67108864
```

## What's next

- [Defining Environment Variables for a Container](#)
- [PodSpec](#)
- [Container](#)
- [EnvVar](#)
- [EnvVarSource](#)
- [ObjectFieldSelector](#)
- [ResourceFieldSelector](#)

## Feedback

Was this page helpful?

Yes No

Thanks for the feedback. If you have a specific, answerable question about how to use Kubernetes, ask it on [Stack Overflow](#). Open an issue in the GitHub repo if you want to [report a problem](#) or [suggest an improvement](#).

---

[Create an Issue](#) [Edit This Page](#)

Page last modified on October 07, 2019 at 1:01 PM PST by [Documentation update \(#16693\)](#) ([Page History](#))

[Edit This Page](#)

# Expose Pod Information to Containers Through Files

This page shows how a Pod can use a DownwardAPIVolumeFile to expose information about itself to Containers running in the Pod. A DownwardAPIVolumeFile can expose Pod fields and Container fields.

- [Before you begin](#)
- [The Downward API](#)

- [Store Pod fields](#)
- [Store Container fields](#)
- [Capabilities of the Downward API](#)
- [Project keys to specific paths and file permissions](#)
- [Motivation for the Downward API](#)
- [What's next](#)

## Before you begin

You need to have a Kubernetes cluster, and the `kubectl` command-line tool must be configured to communicate with your cluster. 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`.

## The Downward API

There are two ways to expose Pod and Container fields to a running Container:

- [Environment variables](#)
- Volume Files

Together, these two ways of exposing Pod and Container fields are called the *Downward API*.

## Store Pod fields

In this exercise, you create a Pod that has one Container. Here is the configuration file for the Pod:

## [pods/inject/dapi-volume.yaml](#)

```
apiVersion: v1
kind: Pod
metadata:
  name: kubernetes-downwardapi-volume-example
  labels:
    zone: us-est-coast
    cluster: test-cluster1
    rack: rack-22
  annotations:
    build: two
    builder: john-doe
spec:
  containers:
    - name: client-container
      image: k8s.gcr.io/busybox
      command: ["sh", "-c"]
      args:
        - while true; do
            if [[ -e /etc/podinfo/labels ]]; then
              echo -en '\n\n'; cat /etc/podinfo/labels; fi;
            if [[ -e /etc/podinfo/annotations ]]; then
              echo -en '\n\n'; cat /etc/podinfo/annotations; fi;
            sleep 5;
          done;
      volumeMounts:
        - name: podinfo
          mountPath: /etc/podinfo
  volumes:
    - name: podinfo
      downwardAPI:
        items:
          - path: "labels"
            fieldRef:
              fieldPath: metadata.labels
          - path: "annotations"
            fieldRef:
              fieldPath: metadata.annotations
```

In the configuration file, you can see that the Pod has a `downwardAPI` Volume, and the Container mounts the Volume at `/etc/podinfo`.

Look at the `items` array under `downwardAPI`. Each element of the array is a [DownwardAPIVolumeFile](#). The first element specifies that the value of the Pod's `metadata.labels` field should be stored in a file named `labels`. The second element specifies that the value of the Pod's `annotations` field should be stored in a file named `annotations`.

**Note:** The fields in this example are Pod fields. They are not fields of the Container in the Pod.

Create the Pod:

```
kubectl apply -f https://k8s.io/examples/pods/inject/dapi-volume.yaml
```

Verify that the Container in the Pod is running:

```
kubectl get pods
```

View the Container's logs:

```
kubectl logs kubernetes-downwardapi-volume-example
```

The output shows the contents of the `labels` file and the `annotations` file:

```
cluster="test-cluster1"
rack="rack-22"
zone="us-est-coast"

build="two"
builder="john-doe"
```

Get a shell into the Container that is running in your Pod:

```
kubectl exec -it kubernetes-downwardapi-volume-example -- sh
```

In your shell, view the `labels` file:

```
/# cat /etc/podinfo/labels
```

The output shows that all of the Pod's labels have been written to the `labels` file:

```
cluster="test-cluster1"
rack="rack-22"
zone="us-est-coast"
```

Similarly, view the `annotations` file:

```
/# cat /etc/podinfo/annotations
```

View the files in the `/etc/podinfo` directory:

```
/# ls -laR /etc/podinfo
```

In the output, you can see that the `labels` and `annotations` files are in a temporary subdirectory: in this example, `..2982_06_02_21_47_53.299460680`. In the `/etc/podinfo` directory, `..data` is a symbolic link to the temporary subdirectory. Also in the `/etc/podinfo` directory, `labels` and `annotations` are symbolic links.

```
drwxr-xr-x  ... Feb 6 21:47 ..2982_06_02_21_47_53.299460680
lrwxrwxrwx  ... Feb 6 21:47 ..data -> ..
2982_06_02_21_47_53.299460680
lrwxrwxrwx  ... Feb 6 21:47 annotations -> ..data/annotations
lrwxrwxrwx  ... Feb 6 21:47 labels -> ..data/labels

/etc/..2982_06_02_21_47_53.299460680:
total 8
-rw-r--r--  ... Feb 6 21:47 annotations
-rw-r--r--  ... Feb 6 21:47 labels
```

Using symbolic links enables dynamic atomic refresh of the metadata; updates are written to a new temporary directory, and the `..data` symlink is updated atomically using [rename\(2\)](#).

**Note:** A container using Downward API as a [subPath](#) volume mount will not receive Downward API updates.

Exit the shell:

```
/# exit
```

## Store Container fields

The preceding exercise, you stored Pod fields in a `DownwardAPIVolumeFile`. In this next exercise, you store Container fields. Here is the configuration file for a Pod that has one Container:

```
apiVersion: v1
kind: Pod
metadata:
  name: kubernetes-downwardapi-volume-example-2
spec:
  containers:
    - name: client-container
      image: k8s.gcr.io/busybox:1.24
      command: ["sh", "-c"]
      args:
        - while true; do
            echo -en '\n';
            if [[ -e /etc/podinfo/cpu_limit ]]; then
              echo -en '\n'; cat /etc/podinfo/cpu_limit; fi;
            if [[ -e /etc/podinfo/cpu_request ]]; then
              echo -en '\n'; cat /etc/podinfo/cpu_request; fi;
            if [[ -e /etc/podinfo/mem_limit ]]; then
              echo -en '\n'; cat /etc/podinfo/mem_limit; fi;
            if [[ -e /etc/podinfo/mem_request ]]; then
              echo -en '\n'; cat /etc/podinfo/mem_request; fi;
            sleep 5;
          done;
      resources:
        requests:
          memory: "32Mi"
          cpu: "125m"
        limits:
          memory: "64Mi"
          cpu: "250m"
      volumeMounts:
        - name: podinfo
          mountPath: /etc/podinfo
    volumes:
      - name: podinfo
        downwardAPI:
          items:
            - path: "cpu_limit"
              resourceFieldRef:
                containerName: client-container
                resource: limits.cpu
                divisor: 1m
            - path: "cpu_request"
              resourceFieldRef:
                containerName: client-container
                resource: requests.cpu
                divisor: 1m
            - path: "mem_limit"
              resourceFieldRef:
                containerName: client-container
                resource: limits.memory
                divisor: 1Mi
            - path: "mem_request"
              resourceFieldRef:
                containerName: client-container
                resource: requests.memory
                divisor: 1Mi
```

In the configuration file, you can see that the Pod has a downwardAPI Volume, and the Container mounts the Volume at /etc/podinfo.

Look at the `items` array under `downwardAPI`. Each element of the array is a `DownwardAPIVolumeFile`.

The first element specifies that in the Container named `client-container`, the value of the `limits.cpu` field in the format specified by `1m` should be stored in a file named `cpu_limit`. The `divisor` field is optional and has the default value of 1 which means cores for cpu and bytes for memory.

Create the Pod:

```
kubectl apply -f https://k8s.io/examples/pods/inject/dapi-volume-resources.yaml
```

Get a shell into the Container that is running in your Pod:

```
kubectl exec -it kubernetes-downwardapi-volume-example-2 -- sh
```

In your shell, view the `cpu_limit` file:

```
/# cat /etc/podinfo/cpu_limit
```

You can use similar commands to view the `cpu_request`, `mem_limit` and `mem_request` files.

## Capabilities of the Downward API

The following information is available to containers through environment variables and downwardAPI volumes:

- Information available via `fieldRef`:
  - `metadata.name` - the pod's name
  - `metadata.namespace` - the pod's namespace
  - `metadata.uid` - the pod's UID, available since v1.8.0-alpha.2
  - `metadata.labels['<KEY>']` - the value of the pod's label `<KEY>` (for example, `metadata.labels['mylabel']`); available in Kubernetes 1.9+
  - `metadata.annotations['<KEY>']` - the value of the pod's annotation `<KEY>` (for example, `metadata.annotations['myannotation']`); available in Kubernetes 1.9+
- Information available via `resourceFieldRef`:
  - A Container's CPU limit
  - A Container's CPU request
  - A Container's memory limit
  - A Container's memory request
  - A Container's ephemeral-storage limit, available since v1.8.0-beta.0
  - A Container's ephemeral-storage request, available since v1.8.0-beta.0

In addition, the following information is available through downwardAPI volume `fieldRef`:

- `metadata.labels` - all of the pod's labels, formatted as `label-key="escaped-label-value"` with one label per line
- `metadata.annotations` - all of the pod's annotations, formatted as `annotation-key="escaped-annotation-value"` with one annotation per line

The following information is available through environment variables:

- `status.podIP` - the pod's IP address
- `spec.serviceAccountName` - the pod's service account name, available since v1.4.0-alpha.3
- `spec.nodeName` - the node's name, available since v1.4.0-alpha.3
- `status.hostIP` - the node's IP, available since v1.7.0-alpha.1

**Note:** If CPU and memory limits are not specified for a Container, the Downward API defaults to the node allocatable value for CPU and memory.

## Project keys to specific paths and file permissions

You can project keys to specific paths and specific permissions on a per-file basis. For more information, see [Secrets](#).

## Motivation for the Downward API

It is sometimes useful for a Container to have information about itself, without being overly coupled to Kubernetes. The Downward API allows containers to consume information about themselves or the cluster without using the Kubernetes client or API server.

An example is an existing application that assumes a particular well-known environment variable holds a unique identifier. One possibility is to wrap the application, but that is tedious and error prone, and it violates the goal of low coupling. A better option would be to use the Pod's name as an identifier, and inject the Pod's name into the well-known environment variable.

## What's next

- [PodSpec](#)
- [Volume](#)
- [DownwardAPIVolumeSource](#)
- [DownwardAPIVolumeFile](#)
- [ResourceFieldSelector](#)

# Feedback

Was this page helpful?

Yes No

Thanks for the feedback. If you have a specific, answerable question about how to use Kubernetes, ask it on [Stack Overflow](#). Open an issue in the GitHub repo if you want to [report a problem](#) or [suggest an improvement](#).

---

[Create an Issue](#) [Edit This Page](#)

Page last modified on January 15, 2020 at 2:49 PM PST by [Add missing article \(#18707\)](#) ([Page History](#))

[Edit This Page](#)

# Distribute Credentials Securely Using Secrets

This page shows how to securely inject sensitive data, such as passwords and encryption keys, into Pods.

- [Before you begin](#)
- [Convert your secret data to a base-64 representation](#)
- [Create a Secret](#)
- [Create a Pod that has access to the secret data through a Volume](#)
- [Define container environment variables using Secret data](#)
- [Configure all key-value pairs in a Secret as container environment variables](#)
- [What's next](#)

## Before you begin

You need to have a Kubernetes cluster, and the `kubectl` command-line tool must be configured to communicate with your cluster. 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](#)

## Convert your secret data to a base-64 representation

Suppose you want to have two pieces of secret data: a username `my-app` and a password `39528$vdg7Jb`. First, use online Base 64 Encoding Tool [Base64 encoding](#), [Base64 encode](#) to convert your username and password to a base-64 representation. Here's a Linux example:

```
echo -n 'my-app' | base64  
echo -n '39528$vdg7Jb' | base64
```

The output shows that the base-64 representation of your username is `bXktYXBw`, and the base-64 representation of your password is `Mzk1MjgkdmRnN0pi`.

**Caution:** Use a local tool trusted by your OS to decrease the security risks of external tools.

## Create a Secret

Here is a configuration file you can use to create a Secret that holds your username and password:

### [pods/inject/secret.yaml](#)

```
apiVersion: v1
kind: Secret
metadata:
  name: test-secret
data:
  username: bXktYXBw
  password: Mzk1MjgkdmRnN0pi
```

#### 1. Create the Secret

```
kubectl apply -f https://k8s.io/examples/pods/inject/
secret.yaml
```

#### 2. View information about the Secret:

```
kubectl get secret test-secret
```

Output:

NAME	TYPE	DATA	AGE
test-secret	Opaque	2	1m

#### 3. View more detailed information about the Secret:

```
kubectl describe secret test-secret
```

Output:

```
Name:      test-secret
Namespace: default
Labels:    <none>
Annotations: <none>

Type:     Opaque

Data
=====
password: 13 bytes
username: 7 bytes
```

**Note:** If you want to skip the Base64 encoding step, you can create a Secret by using the `kubectl create secret` command:

```
kubectl create secret generic test-secret --from-
literal=username='my-app' --from-literal=password='39528$vdg7Jb'
```

# Create a Pod that has access to the secret data through a Volume

Here is a configuration file you can use to create a Pod:

## [pods/inject/secret-pod.yaml](#)

```
apiVersion: v1
kind: Pod
metadata:
  name: secret-test-pod
spec:
  containers:
    - name: test-container
      image: nginx
      volumeMounts:
        # name must match the volume name below
        - name: secret-volume
          mountPath: /etc/secret-volume
    # The secret data is exposed to Containers in the Pod through
    # a Volume.
  volumes:
    - name: secret-volume
      secret:
        secretName: test-secret
```

1. Create the Pod:

```
kubectl apply -f https://k8s.io/examples/pods/inject/secret-
pod.yaml
```

2. Verify that your Pod is running:

```
kubectl get pod secret-test-pod
```

Output:

NAME	READY	STATUS	RESTARTS	AGE
secret-test-pod	1/1	Running	0	42m

3. Get a shell into the Container that is running in your Pod:

```
kubectl exec -it secret-test-pod -- /bin/bash
```

4. The secret data is exposed to the Container through a Volume mounted under /etc/secret-volume. In your shell, go to the directory where the secret data is exposed:

```
root@secret-test-pod:/# cd /etc/secret-volume
```

5. In your shell, list the files in the /etc/secret-volume directory:

```
root@secret-test-pod:/etc/secret-volume# ls
```

The output shows two files, one for each piece of secret data:

```
password username
```

6. In your shell, display the contents of the username and password files:

```
root@secret-test-pod:/etc/secret-volume# cat username; echo;  
cat password; echo
```

The output is your username and password:

```
my-app  
39528$vdg7Jb
```

## Define container environment variables using Secret data

### Define a container environment variable with data from a single Secret

- Define an environment variable as a key-value pair in a Secret:

```
kubectl create secret generic backend-user --from-literal=backend-username='backend-admin'
```

- Assign the backend-username value defined in the Secret to the SECRET\_USERNAME environment variable in the Pod specification.

[pods/inject/pod-single-secret-env-variable.yaml](#)

```
apiVersion: v1  
kind: Pod  
metadata:  
  name: env-single-secret  
spec:  
  containers:  
    - name: envvars-test-container  
      image: nginx  
      env:  
        - name: SECRET_USERNAME  
          valueFrom:  
            secretKeyRef:  
              name: backend-user  
              key: backend-username
```

- Create the Pod:

```
kubectl create -f https://k8s.io/examples/pods/inject/pod-single-secret-env-variable.yaml
```

- Now, the Pod's output includes environment variable SECRET\_USERNAME=backend-admin

## Define container environment variables with data from multiple Secrets

- As with the previous example, create the Secrets first.

```
kubectl create secret generic backend-user --from-literal=backend-username='backend-admin'
```

```
kubectl create secret generic db-user --from-literal=db-username='db-admin'
```

- Define the environment variables in the Pod specification.

### [pods/inject/pod-multiple-secret-env-variable.yaml](#)

```
apiVersion: v1
kind: Pod
metadata:
  name: envvars-multiple-secrets
spec:
  containers:
    - name: envvars-test-container
      image: nginx
      env:
        - name: BACKEND_USERNAME
          valueFrom:
            secretKeyRef:
              name: backend-user
              key: backend-username
        - name: DB_USERNAME
          valueFrom:
            secretKeyRef:
              name: db-user
              key: db-username
```

- Create the Pod:

```
kubectl create -f https://k8s.io/examples/pods/inject/pod-multiple-secret-env-variable.yaml
```

- Now, the Pod's output includes BACKEND\_USERNAME=backend-admin and DB\_USERNAME=db-admin environment variables.

# Configure all key-value pairs in a Secret as container environment variables

**Note:** This functionality is available in Kubernetes v1.6 and later.

- Create a Secret containing multiple key-value pairs

```
kubectl create secret generic test-secret --from-literal=user  
name='my-app' --from-literal=password='39528$vdg7Jb'
```

- Use envFrom to define all of the Secret's data as container environment variables. The key from the Secret becomes the environment variable name in the Pod.

## [pods/inject/pod-secret-envFrom.yaml](#)

```
apiVersion: v1  
kind: Pod  
metadata:  
  name: envfrom-secret  
spec:  
  containers:  
    - name: envvars-test-container  
      image: nginx  
      envFrom:  
        - secretRef:  
            name: test-secret
```

- Create the Pod:

```
kubectl create -f https://k8s.io/examples/pods/inject/pod-  
secret-envFrom.yaml
```

- Now, the Pod's output includes `username=my-app` and `password=39528$vdg7Jb` environment variables.

## What's next

- Learn more about [Secrets](#).
- Learn about [Volumes](#).

## Reference

- [Secret](#)
- [Volume](#)
- [Pod](#)

# Feedback

Was this page helpful?

Yes No

Thanks for the feedback. If you have a specific, answerable question about how to use Kubernetes, ask it on [Stack Overflow](#). Open an issue in the GitHub repo if you want to [report a problem](#) or [suggest an improvement](#).

---

[Create an Issue](#) [Edit This Page](#)

Page last modified on January 29, 2020 at 11:02 PM PST by [Revise version requirements \(#18688\)](#) ([Page History](#))

[Edit This Page](#)

# Inject Information into Pods Using a PodPreset

You can use a PodPreset object to inject information like secrets, volume mounts, and environment variables etc into pods at creation time. This task shows some examples on using the PodPreset resource.

- [Before you begin](#)
- [Simple Pod Spec Example](#)
- [Pod Spec with ConfigMap Example](#)
- [ReplicaSet with Pod Spec Example](#)
- [Multiple PodPreset Example](#)
- [Conflict Example](#)
- [Deleting a Pod Preset](#)

## Before you begin

Get an overview of PodPresets at [Understanding Pod Presets](#).

You need to have a Kubernetes cluster, and the kubectl command-line tool must be configured to communicate with your cluster. 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`.

## Simple Pod Spec Example

This is a simple example to show how a Pod spec is modified by the Pod Preset.

### podpreset/preset.yaml

```
apiVersion: settings.k8s.io/v1alpha1
kind: PodPreset
metadata:
  name: allow-database
spec:
  selector:
    matchLabels:
      role: frontend
  env:
    - name: DB_PORT
      value: "6379"
  volumeMounts:
    - mountPath: /cache
      name: cache-volume
  volumes:
    - name: cache-volume
      emptyDir: {}
```

Create the PodPreset:

```
kubectl apply -f https://k8s.io/examples/podpreset/preset.yaml
```

Examine the created PodPreset:

```
kubectl get podpreset
```

NAME	AGE
allow-database	1m

The new PodPreset will act upon any pod that has label `role: frontend`.

### podpreset/pod.yaml

```
apiVersion: v1
kind: Pod
metadata:
  name: website
  labels:
    app: website
    role: frontend
spec:
  containers:
    - name: website
      image: nginx
      ports:
        - containerPort: 80
```

Create a pod:

```
kubectl create -f https://k8s.io/examples/podpreset/pod.yaml
```

List the running Pods:

```
kubectl get pods
```

NAME	READY	STATUS	RESTARTS	AGE
website	1/1	Running	0	4m

### Pod spec after admission controller:

#### [podpreset/merged.yaml](#)

```
apiVersion: v1
kind: Pod
metadata:
  name: website
  labels:
    app: website
    role: frontend
  annotations:
    podpreset.admission.kubernetes.io/podpreset-allow-database:
"resource version"
spec:
  containers:
    - name: website
      image: nginx
      volumeMounts:
        - mountPath: /cache
          name: cache-volume
      ports:
        - containerPort: 80
      env:
        - name: DB_PORT
          value: "6379"
  volumes:
    - name: cache-volume
      emptyDir: {}
```

To see above output, run the following command:

```
kubectl get pod website -o yaml
```

## Pod Spec with ConfigMap Example

This is an example to show how a Pod spec is modified by the Pod Preset that defines a ConfigMap for Environment Variables.

## User submitted pod spec:

### [podpreset/pod.yaml](#)

```
apiVersion: v1
kind: Pod
metadata:
  name: website
  labels:
    app: website
    role: frontend
spec:
  containers:
    - name: website
      image: nginx
      ports:
        - containerPort: 80
```

## User submitted ConfigMap:

### [podpreset/configmap.yaml](#)

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: etcd-env-config
data:
  number_of_members: "1"
  initial_cluster_state: new
  initial_cluster_token: DUMMY_ETCD_INITIAL_CLUSTER_TOKEN
  discovery_token: DUMMY_ETCD_DISCOVERY_TOKEN
  discovery_url: http://etcd_discovery:2379
  etcdctl_peers: http://etcd:2379
  duplicate_key: FROM_CONFIG_MAP
  REPLACE_ME: "a value"
```

## Example Pod Preset:

### [podpreset/allow-db.yaml](#)

```
apiVersion: settings.k8s.io/v1alpha1
kind: PodPreset
metadata:
  name: allow-database
spec:
  selector:
    matchLabels:
      role: frontend
  env:
    - name: DB_PORT
      value: "6379"
    - name: duplicate_key
      value: FROM_ENV
    - name: expansion
      value: $(REPLACE_ME)
  envFrom:
    - configMapRef:
        name: etcd-env-config
  volumeMounts:
    - mountPath: /cache
      name: cache-volume
    - mountPath: /etc/app/config.json
      readOnly: true
      name: secret-volume
  volumes:
    - name: cache-volume
      emptyDir: {}
    - name: secret-volume
      secret:
        secretName: config-details
```

### **Pod spec after admission controller:**

### [podpreset/allow-db-merged.yaml](#)

```
apiVersion: v1
kind: Pod
metadata:
  name: website
  labels:
    app: website
    role: frontend
  annotations:
    podpreset.admission.kubernetes.io/podpreset-allow-database:
      "resource version"
spec:
  containers:
    - name: website
      image: nginx
      volumeMounts:
        - mountPath: /cache
          name: cache-volume
        - mountPath: /etc/app/config.json
          readOnly: true
          name: secret-volume
      ports:
        - containerPort: 80
    env:
      - name: DB_PORT
        value: "6379"
      - name: duplicate_key
        value: FROM_ENV
      - name: expansion
        value: $(REPLACE_ME)
    envFrom:
      - configMapRef:
          name: etcd-env-config
  volumes:
    - name: cache-volume
      emptyDir: {}
    - name: secret-volume
      secret:
        secretName: config-details
```

## **ReplicaSet with Pod Spec Example**

The following example shows that only the pod spec is modified by the Pod Preset.

### **User submitted ReplicaSet:**

### [podpreset/replicaset.yaml](#)

```
apiVersion: apps/v1
kind: ReplicaSet
metadata:
  name: frontend
spec:
  replicas: 3
  selector:
    matchLabels:
      role: frontend
    matchExpressions:
      - {key: role, operator: In, values: [frontend]}
  template:
    metadata:
      labels:
        app: guestbook
        role: frontend
    spec:
      containers:
        - name: php-redis
          image: gcr.io/google_samples/gb-frontend:v3
          resources:
            requests:
              cpu: 100m
              memory: 100Mi
          env:
            - name: GET_HOSTS_FROM
              value: dns
          ports:
            - containerPort: 80
```

### **Example Pod Preset:**

### [podpreset/preset.yaml](#)

```
apiVersion: settings.k8s.io/v1alpha1
kind: PodPreset
metadata:
  name: allow-database
spec:
  selector:
    matchLabels:
      role: frontend
  env:
    - name: DB_PORT
      value: "6379"
  volumeMounts:
    - mountPath: /cache
      name: cache-volume
  volumes:
    - name: cache-volume
      emptyDir: {}
```

### **Pod spec after admission controller:**

Note that the ReplicaSet spec was not changed, users have to check individual pods to validate that the PodPreset has been applied.

## [`podpreset/replicaset-merged.yaml`](#)

```
apiVersion: v1
kind: Pod
metadata:
  name: frontend
  labels:
    app: guestbook
    role: frontend
  annotations:
    podpreset.admission.kubernetes.io/podpreset-allow-database:
"resource version"
spec:
  containers:
    - name: php-redis
      image: gcr.io/google_samples/gb-frontend:v3
      resources:
        requests:
          cpu: 100m
          memory: 100Mi
      volumeMounts:
        - mountPath: /cache
          name: cache-volume
      env:
        - name: GET_HOSTS_FROM
          value: dns
        - name: DB_PORT
          value: "6379"
      ports:
        - containerPort: 80
    volumes:
      - name: cache-volume
        emptyDir: {}
```

## Multiple PodPreset Example

This is an example to show how a Pod spec is modified by multiple Pod Injection Policies.

### User submitted pod spec:

### [podpreset/pod.yaml](#)

```
apiVersion: v1
kind: Pod
metadata:
  name: website
  labels:
    app: website
    role: frontend
spec:
  containers:
    - name: website
      image: nginx
      ports:
        - containerPort: 80
```

### **Example Pod Preset:**

### [podpreset/preset.yaml](#)

```
apiVersion: settings.k8s.io/v1alpha1
kind: PodPreset
metadata:
  name: allow-database
spec:
  selector:
    matchLabels:
      role: frontend
  env:
    - name: DB_PORT
      value: "6379"
  volumeMounts:
    - mountPath: /cache
      name: cache-volume
  volumes:
    - name: cache-volume
      emptyDir: {}
```

### **Another Pod Preset:**

### [podpreset/proxy.yaml](#)

```
apiVersion: settings.k8s.io/v1alpha1
kind: PodPreset
metadata:
  name: proxy
spec:
  selector:
    matchLabels:
      role: frontend
  volumeMounts:
    - mountPath: /etc/proxy/configs
      name: proxy-volume
  volumes:
    - name: proxy-volume
      emptyDir: {}
```

### **Pod spec after admission controller:**

## [podpreset/multi-merged.yaml](#)

```
apiVersion: v1
kind: Pod
metadata:
  name: website
  labels:
    app: website
    role: frontend
  annotations:
    podpreset.admission.kubernetes.io/podpreset-allow-database:
"resource version"
    podpreset.admission.kubernetes.io/podpreset-proxy:
"resource version"
spec:
  containers:
    - name: website
      image: nginx
      volumeMounts:
        - mountPath: /cache
          name: cache-volume
        - mountPath: /etc/proxy/configs
          name: proxy-volume
      ports:
        - containerPort: 80
      env:
        - name: DB_PORT
          value: "6379"
  volumes:
    - name: cache-volume
      emptyDir: {}
    - name: proxy-volume
      emptyDir: {}
```

## **Conflict Example**

This is an example to show how a Pod spec is not modified by the Pod Preset when there is a conflict.

**User submitted pod spec:**

### [podpreset/conflict-pod.yaml](#)

```
apiVersion: v1
kind: Pod
metadata:
  name: website
  labels:
    app: website
    role: frontend
spec:
  containers:
    - name: website
      image: nginx
      volumeMounts:
        - mountPath: /cache
          name: cache-volume
      ports:
        - containerPort: 80
  volumes:
    - name: cache-volume
      emptyDir: {}
```

### **Example Pod Preset:**

### [podpreset/conflict-preset.yaml](#)

```
apiVersion: settings.k8s.io/v1alpha1
kind: PodPreset
metadata:
  name: allow-database
spec:
  selector:
    matchLabels:
      role: frontend
  env:
    - name: DB_PORT
      value: "6379"
  volumeMounts:
    - mountPath: /cache
      name: other-volume
  volumes:
    - name: other-volume
      emptyDir: {}
```

### **Pod spec after admission controller will not change because of the conflict:**

### [podpreset/conflict-pod.yaml](#)

```
apiVersion: v1
kind: Pod
metadata:
  name: website
  labels:
    app: website
    role: frontend
spec:
  containers:
    - name: website
      image: nginx
      volumeMounts:
        - mountPath: /cache
          name: cache-volume
      ports:
        - containerPort: 80
  volumes:
    - name: cache-volume
      emptyDir: {}
```

If we run **kubectl describe...** we can see the event:

```
kubectl describe ...
```

```
....  
Events:  
FirstSeen           LastSeen           Count  
From               SubobjectPath     Reason  
Message  
Tue, 07 Feb 2017 16:56:12 -0700  Tue, 07 Feb 2017 16:56:12  
-0700 1 {podpreset.admission.kubernetes.io/podpreset-allow-  
database } conflict Conflict on pod preset. Duplicate  
mountPath /cache.
```

## Deleting a Pod Preset

Once you don't need a pod preset anymore, you can delete it with **kubectl**:

```
kubectl delete podpreset allow-database
```

```
podpreset "allow-database" deleted
```

## Feedback

Was this page helpful?

Yes No

Thanks for the feedback. If you have a specific, answerable question about how to use Kubernetes, ask it on [Stack Overflow](#). Open an issue in the GitHub repo if you want to [report a problem](#) or [suggest an improvement](#).

---

[Create an Issue](#) [Edit This Page](#)

Page last modified on March 25, 2019 at 5:06 PM PST by [Official 1.14](#)  
[Release Docs \(#13174\)](#) ([Page History](#))

[Edit This Page](#)

# Run a Stateless Application Using a Deployment

This page shows how to run an application using a Kubernetes Deployment object.

- [Objectives](#)
- [Before you begin](#)
- [Creating and exploring an nginx deployment](#)
- [Updating the deployment](#)
- [Scaling the application by increasing the replica count](#)
- [Deleting a deployment](#)
- [ReplicationControllers - the Old Way](#)
- [What's next](#)

## Objectives

- Create an nginx deployment.
- Use kubectl to list information about the deployment.
- Update the deployment.

## Before you begin

You need to have a Kubernetes cluster, and the kubectl command-line tool must be configured to communicate with your cluster. 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`.

## Creating and exploring an nginx deployment

You can run an application by creating a Kubernetes Deployment object, and you can describe a Deployment in a YAML file. For example, this YAML file describes a Deployment that runs the nginx:1.7.9 Docker image:

## [application/deployment.yaml](#)

```
apiVersion: apps/v1 # for versions before 1.9.0 use apps/v1beta2
kind: Deployment
metadata:
  name: nginx-deployment
spec:
  selector:
    matchLabels:
      app: nginx
  replicas: 2 # tells deployment to run 2 pods matching the template
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
        - name: nginx
          image: nginx:1.7.9
          ports:
            - containerPort: 80
```

1. Create a Deployment based on the YAML file:

```
kubectl apply -f https://k8s.io/examples/application/deployment.yaml
```

2. Display information about the Deployment:

```
kubectl describe deployment nginx-deployment
```

The output is similar to this:

```
user@computer:~/website$ kubectl describe deployment nginx-deployment
Name:      nginx-deployment
Namespace:  default
CreationTimestamp: Tue, 30 Aug 2016 18:11:37 -0700
Labels:    app=nginx
Annotations: deployment.kubernetes.io/revision=1
Selector:  app=nginx
Replicas:  2 desired | 2 updated | 2 total | 2 available | 0 unavailable
StrategyType: RollingUpdate
MinReadySeconds: 0
RollingUpdateStrategy: 1 max unavailable, 1 max surge
Pod Template:
  Labels:      app=nginx
  Containers:
    nginx:
```

```
  Image:          nginx:1.7.9
  Port:          80/TCP
  Environment:    <none>
  Mounts:         <none>
  Volumes:        <none>
Conditions:
  Type     Status  Reason
  ----
  Available  True   MinimumReplicasAvailable
  Progressing True   NewReplicaSetAvailable
OldReplicaSets:  <none>
NewReplicaSet:   nginx-deployment-1771418926 (2/2 replicas created)
No events.
```

3. List the pods created by the deployment:

```
kubectl get pods -l app=nginx
```

The output is similar to this:

NAME	READY	STATUS
RESTARTS AGE		
nginx-deployment-1771418926-7o5ns 0 16h	1/1	Running
nginx-deployment-1771418926-r18az 0 16h	1/1	Running

4. Display information about a pod:

```
kubectl describe pod <pod-name>
```

where `<pod-name>` is the name of one of your pods.

## Updating the deployment

You can update the deployment by applying a new YAML file. This YAML file specifies that the deployment should be updated to use nginx 1.8.

## [application/deployment-update.yaml](#)

```
apiVersion: apps/v1 # for versions before 1.9.0 use apps/v1beta2
kind: Deployment
metadata:
  name: nginx-deployment
spec:
  selector:
    matchLabels:
      app: nginx
  replicas: 2
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
        - name: nginx
          image: nginx:1.8 # Update the version of nginx from
1.7.9 to 1.8
          ports:
            - containerPort: 80
```

1. Apply the new YAML file:

```
kubectl apply -f https://k8s.io/examples/application/
deployment-update.yaml
```

2. Watch the deployment create pods with new names and delete the old pods:

```
kubectl get pods -l app=nginx
```

## **Scaling the application by increasing the replica count**

You can increase the number of pods in your Deployment by applying a new YAML file. This YAML file sets `replicas` to 4, which specifies that the Deployment should have four pods:

## [application/deployment-scale.yaml](#)

```
apiVersion: apps/v1 # for versions before 1.9.0 use apps/v1beta2
kind: Deployment
metadata:
  name: nginx-deployment
spec:
  selector:
    matchLabels:
      app: nginx
  replicas: 4 # Update the replicas from 2 to 4
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
        - name: nginx
          image: nginx:1.8
          ports:
            - containerPort: 80
```

1. Apply the new YAML file:

```
kubectl apply -f https://k8s.io/examples/application/
deployment-scale.yaml
```

2. Verify that the Deployment has four pods:

```
kubectl get pods -l app=nginx
```

The output is similar to this:

NAME	READY	STATUS
RESTARTS	AGE	
nginx-deployment-148880595-4zdqq	1/1	Running
0	25s	
nginx-deployment-148880595-6zgil	1/1	Running
0	25s	
nginx-deployment-148880595-fxcez	1/1	Running
0	2m	
nginx-deployment-148880595-rwovn	1/1	Running
0	2m	

## **Deleting a deployment**

Delete the deployment by name:

```
kubectl delete deployment nginx-deployment
```

# ReplicationControllers - the Old Way

The preferred way to create a replicated application is to use a Deployment, which in turn uses a ReplicaSet. Before the Deployment and ReplicaSet were added to Kubernetes, replicated applications were configured using a [ReplicationController](#).

## What's next

- Learn more about [Deployment objects](#).

## Feedback

Was this page helpful?

Yes No

Thanks for the feedback. If you have a specific, answerable question about how to use Kubernetes, ask it on [Stack Overflow](#). Open an issue in the GitHub repo if you want to [report a problem](#) or [suggest an improvement](#).

---

[Create an Issue](#) [Edit This Page](#)

Page last modified on August 21, 2018 at 1:09 AM PST by [Minor grammar edit \(#9921\)](#) ([Page History](#))

[Edit This Page](#)

# Run a Single-Instance Stateful Application

This page shows you how to run a single-instance stateful application in Kubernetes using a PersistentVolume and a Deployment. The application is MySQL.

- [Objectives](#)
- [Before you begin](#)

- [Deploy MySQL](#)
- [Accessing the MySQL instance](#)
- [Updating](#)
- [Deleting a deployment](#)
- [What's next](#)

## Objectives

- Create a PersistentVolume referencing a disk in your environment.
- Create a MySQL Deployment.
- Expose MySQL to other pods in the cluster at a known DNS name.

## Before you begin

- You need to have a Kubernetes cluster, and the kubectl command-line tool must be configured to communicate with your cluster. 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`.

- You need to either have a dynamic PersistentVolume provisioner with a default [StorageClass](#), or [statically provision PersistentVolumes](#) yourself to satisfy the [PersistentVolumeClaims](#) used here.

## Deploy MySQL

You can run a stateful application by creating a Kubernetes Deployment and connecting it to an existing PersistentVolume using a PersistentVolumeClaim. For example, this YAML file describes a Deployment that runs MySQL and references the PersistentVolumeClaim. The file defines a volume mount for `/var/lib/mysql`, and then creates a PersistentVolumeClaim that looks for a 20G volume. This claim is satisfied by any existing volume that meets the requirements, or by a dynamic provisioner.

Note: The password is defined in the config yaml, and this is insecure. See [Kubernetes Secrets](#) for a secure solution.

## [application/mysql/mysql-deployment.yaml](#)

```
apiVersion: v1
kind: Service
metadata:
  name: mysql
spec:
  ports:
  - port: 3306
  selector:
    app: mysql
  clusterIP: None
---
apiVersion: apps/v1 # for versions before 1.9.0 use apps/v1beta2
kind: Deployment
metadata:
  name: mysql
spec:
  selector:
    matchLabels:
      app: mysql
  strategy:
    type: Recreate
  template:
    metadata:
      labels:
        app: mysql
    spec:
      containers:
      - image: mysql:5.6
        name: mysql
        env:
          # Use secret in real usage
          - name: MYSQL_ROOT_PASSWORD
            value: password
        ports:
        - containerPort: 3306
          name: mysql
        volumeMounts:
        - name: mysql-persistent-storage
          mountPath: /var/lib/mysql
      volumes:
      - name: mysql-persistent-storage
        persistentVolumeClaim:
          claimName: mysql-pv-claim
```

## [application/mysql/mysql-pv.yaml](#)

```
apiVersion: v1
kind: PersistentVolume
metadata:
  name: mysql-pv-volume
  labels:
    type: local
spec:
  storageClassName: manual
  capacity:
    storage: 20Gi
  accessModes:
    - ReadWriteOnce
  hostPath:
    path: "/mnt/data"
---
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: mysql-pv-claim
spec:
  storageClassName: manual
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: 20Gi
```

1. Deploy the PV and PVC of the YAML file:

```
kubectl apply -f https://k8s.io/examples/application/mysql/mysql-pv.yaml
```

2. Deploy the contents of the YAML file:

```
kubectl apply -f https://k8s.io/examples/application/mysql/mysql-deployment.yaml
```

3. Display information about the Deployment:

```
kubectl describe deployment mysql

Name:                   mysql
Namespace:              default
CreationTimestamp:      Tue, 01 Nov 2016 11:18:45 -0700
Labels:                 app=mysql
Annotations:            deployment.kubernetes.io/revision=1
Selector:               app=mysql
Replicas:               1 desired | 1 updated | 1 total | 0
available | 1 unavailable
```

```

StrategyType:            Recreate
MinReadySeconds:         0
Pod Template:
  Labels:      app=mysql
  Containers:
    mysql:
      Image:      mysql:5.6
      Port:       3306/TCP
      Environment:
        MYSQL_ROOT_PASSWORD:      password
      Mounts:
        /var/lib/mysql from mysql-persistent-storage (rw)
  Volumes:
    mysql-persistent-storage:
      Type:      PersistentVolumeClaim (a reference to a
PersistentVolumeClaim in the same namespace)
        ClaimName:  mysql-pv-claim
        ReadOnly:   false
Conditions:
  Type        Status  Reason
  ----        ----   -----
  Available   False   MinimumReplicasUnavailable
  Progressing True    ReplicaSetUpdated
OldReplicaSets: <none>
NewReplicaSet:  mysql-63082529 (1/1 replicas created)
Events:
  FirstSeen  LastSeen  Count  From
SubobjectPath  Type      Reason           Message
  -----  -----  -----  -----
  33s       33s       1     {deployment-
controller }          Normal   ScalingReplicaSet
Scaled up replica set mysql-63082529 to 1

```

4. List the pods created by the Deployment:

```
kubectl get pods -l app=mysql
```

NAME	READY	STATUS	RESTARTS	AGE
mysql-63082529-2z3ki	1/1	Running	0	3m

5. Inspect the PersistentVolumeClaim:

```
kubectl describe pvc mysql-pv-claim
```

Name:	Namespace:	Status:	Volume:	Labels:	Annotations:
mysql-pv-claim	default	Bound	mysql-pv-volume	<none>	pv.kubernetes.io/bind-completed=yes

```
pv.kubernetes.io/bound-by-controller=yes  
Capacity: 20Gi  
Access Modes: RWO  
Events: <none>
```

## Accessing the MySQL instance

The preceding YAML file creates a service that allows other Pods in the cluster to access the database. The Service option `clusterIP: None` lets the Service DNS name resolve directly to the Pod's IP address. This is optimal when you have only one Pod behind a Service and you don't intend to increase the number of Pods.

Run a MySQL client to connect to the server:

```
kubectl run -it --rm --image=mysql:5.6 --restart=Never mysql-client -- mysql -h mysql -ppassword
```

This command creates a new Pod in the cluster running a MySQL client and connects it to the server through the Service. If it connects, you know your stateful MySQL database is up and running.

```
Waiting for pod default/mysql-client-274442439-zyp6i to be  
running, status is Pending, pod ready: false  
If you don't see a command prompt, try pressing enter.
```

```
mysql>
```

## Updating

The image or any other part of the Deployment can be updated as usual with the `kubectl apply` command. Here are some precautions that are specific to stateful apps:

- Don't scale the app. This setup is for single-instance apps only. The underlying PersistentVolume can only be mounted to one Pod. For clustered stateful apps, see the [StatefulSet documentation](#).
- Use `strategy: type: Recreate` in the Deployment configuration YAML file. This instructs Kubernetes to *not* use rolling updates. Rolling updates will not work, as you cannot have more than one Pod running at a time. The Recreate strategy will stop the first pod before creating a new one with the updated configuration.

## Deleting a deployment

Delete the deployed objects by name:

```
kubectl delete deployment,svc mysql  
kubectl delete pvc mysql-pv-claim  
kubectl delete pv mysql-pv-volume
```

If you manually provisioned a PersistentVolume, you also need to manually delete it, as well as release the underlying resource. If you used a dynamic provisioner, it automatically deletes the PersistentVolume when it sees that you deleted the PersistentVolumeClaim. Some dynamic provisioners (such as those for EBS and PD) also release the underlying resource upon deleting the PersistentVolume.

## What's next

- Learn more about [Deployment objects](#).
- Learn more about [Deploying applications](#)
- [kubectl run documentation](#)
- [Volumes](#) and [Persistent Volumes](#)

## Feedback

Was this page helpful?

Yes No

Thanks for the feedback. If you have a specific, answerable question about how to use Kubernetes, ask it on [Stack Overflow](#). Open an issue in the GitHub repo if you want to [report a problem](#) or [suggest an improvement](#).

---

[Create an Issue](#) [Edit This Page](#)

Page last modified on February 11, 2020 at 11:18 AM PST by [Replace links with redirect destination \(#19038\)](#) ([Page History](#))

[Edit This Page](#)

# Run a Replicated Stateful Application

This page shows how to run a replicated stateful application using a [StatefulSet](#) controller. The example is a MySQL single-master topology with multiple slaves running asynchronous replication.

**Note: This is not a production configuration.** MySQL settings remain on insecure defaults to keep the focus on general patterns for running stateful applications in Kubernetes.

- [Objectives](#)
- [Before you begin](#)
- [Deploy MySQL](#)
- [Understanding stateful Pod initialization](#)
- [Sending client traffic](#)
- [Simulating Pod and Node downtime](#)
- [Scaling the number of slaves](#)
- [Cleaning up](#)
- [What's next](#)

## Objectives

- Deploy a replicated MySQL topology with a StatefulSet controller.
- Send MySQL client traffic.
- Observe resistance to downtime.
- Scale the StatefulSet up and down.

## Before you begin

- You need to have a Kubernetes cluster, and the kubectl command-line tool must be configured to communicate with your cluster. 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`.

- You need to either have a dynamic PersistentVolume provisioner with a default [StorageClass](#), or [statically provision PersistentVolumes](#) yourself to satisfy the [PersistentVolumeClaims](#) used here.
- This tutorial assumes you are familiar with [PersistentVolumes](#) and [StatefulSets](#), as well as other core concepts like [Pods](#), [Services](#), and [ConfigMaps](#).
- Some familiarity with MySQL helps, but this tutorial aims to present general patterns that should be useful for other systems.

## Deploy MySQL

The example MySQL deployment consists of a ConfigMap, two Services, and a StatefulSet.

## ConfigMap

Create the ConfigMap from the following YAML configuration file:

[application/mysql/mysql-configmap.yaml](https://k8s.io/examples/application/mysql/mysql-configmap.yaml)

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: mysql
  labels:
    app: mysql
data:
  master.cnf: |
    # Apply this config only on the master.
    [mysqld]
    log-bin
  slave.cnf: |
    # Apply this config only on slaves.
    [mysqld]
    super-read-only
```

```
kubectl apply -f https://k8s.io/examples/application/mysql/mysql-
configmap.yaml
```

This ConfigMap provides `my.cnf` overrides that let you independently control configuration on the MySQL master and slaves. In this case, you want the master to be able to serve replication logs to slaves and you want slaves to reject any writes that don't come via replication.

There's nothing special about the ConfigMap itself that causes different portions to apply to different Pods. Each Pod decides which portion to look at as it's initializing, based on information provided by the StatefulSet controller.

## Services

Create the Services from the following YAML configuration file:

## [application/mysql/mysql-services.yaml](#)

```
# Headless service for stable DNS entries of StatefulSet members.
apiVersion: v1
kind: Service
metadata:
  name: mysql
  labels:
    app: mysql
spec:
  ports:
  - name: mysql
    port: 3306
  clusterIP: None
  selector:
    app: mysql
---
# Client service for connecting to any MySQL instance for reads.
# For writes, you must instead connect to the master:
mysql-read.mysql.
apiVersion: v1
kind: Service
metadata:
  name: mysql-read
  labels:
    app: mysql
spec:
  ports:
  - name: mysql
    port: 3306
  selector:
    app: mysql
```

```
kubectl apply -f https://k8s.io/examples/application/mysql/mysql-services.yaml
```

The Headless Service provides a home for the DNS entries that the StatefulSet controller creates for each Pod that's part of the set. Because the Headless Service is named `mysql`, the Pods are accessible by resolving `<pod-name>.mysql` from within any other Pod in the same Kubernetes cluster and namespace.

The Client Service, called `mysql-read`, is a normal Service with its own cluster IP that distributes connections across all MySQL Pods that report being Ready. The set of potential endpoints includes the MySQL master and all slaves.

Note that only read queries can use the load-balanced Client Service. Because there is only one MySQL master, clients should connect directly to the MySQL master Pod (through its DNS entry within the Headless Service) to execute writes.

## **StatefulSet**

Finally, create the StatefulSet from the following YAML configuration file:

```

apiVersion: apps/v1
kind: StatefulSet
metadata:
  name: mysql
spec:
  selector:
    matchLabels:
      app: mysql
  serviceName: mysql
  replicas: 3
  template:
    metadata:
      labels:
        app: mysql
  spec:
    initContainers:
      - name: init-mysql
        image: mysql:5.7
        command:
          - bash
          - "-c"
          - |
            set -ex
            # Generate mysql server-id from pod ordinal index.
            [[ `hostname` =~ -([0-9]+)$ ]] || exit 1
            ordinal=${BASH_REMATCH[1]}
            echo [mysqld] > /mnt/conf.d/server-id.cnf
            # Add an offset to avoid reserved server-id=0 value.
            echo server-id=$((100 + $ordinal)) >> /mnt/conf.d/
server-id.cnf
            # Copy appropriate conf.d files from config-map to
emptyDir.
            if [[ $ordinal -eq 0 ]]; then
              cp /mnt/config-map/master.cnf /mnt/conf.d/
            else
              cp /mnt/config-map/slave.cnf /mnt/conf.d/
            fi
    volumeMounts:
      - name: conf
        mountPath: /mnt/conf.d
      - name: config-map
        mountPath: /mnt/config-map
    - name: clone-mysql
      image: gcr.io/google-samples/xtrabackup:1.0
      command:
        - bash
        - "-c"
        - |
          set -ex
          # Skip the clone if data already exists.
          [[ -d /var/lib/mysql/mysql ]] && exit 0
          # Skip the clone on master (ordinal index 0).
          [[ `hostname` =~ -([0-9]+)$ ]] || exit 1
          ordinal=${BASH_REMATCH[1]}
          [[ $ordinal -eq 0 ]] && exit 0
          # Clone data from previous peer.

```

```
kubectl apply -f https://k8s.io/examples/application/mysql/mysql-statefulset.yaml
```

You can watch the startup progress by running:

```
kubectl get pods -l app=mysql --watch
```

After a while, you should see all 3 Pods become Running:

NAME	READY	STATUS	RESTARTS	AGE
mysql-0	2/2	Running	0	2m
mysql-1	2/2	Running	0	1m
mysql-2	2/2	Running	0	1m

Press **Ctrl+C** to cancel the watch. If you don't see any progress, make sure you have a dynamic PersistentVolume provisioner enabled as mentioned in the [prerequisites](#).

This manifest uses a variety of techniques for managing stateful Pods as part of a StatefulSet. The next section highlights some of these techniques to explain what happens as the StatefulSet creates Pods.

## Understanding stateful Pod initialization

The StatefulSet controller starts Pods one at a time, in order by their ordinal index. It waits until each Pod reports being Ready before starting the next one.

In addition, the controller assigns each Pod a unique, stable name of the form <statefulset-name>-<ordinal-index>, which results in Pods named mysql-0, mysql-1, and mysql-2.

The Pod template in the above StatefulSet manifest takes advantage of these properties to perform orderly startup of MySQL replication.

## Generating configuration

Before starting any of the containers in the Pod spec, the Pod first runs any [Init Containers](#) in the order defined.

The first Init Container, named `init-mysql`, generates special MySQL config files based on the ordinal index.

The script determines its own ordinal index by extracting it from the end of the Pod name, which is returned by the `hostname` command. Then it saves the ordinal (with a numeric offset to avoid reserved values) into a file called `server-id.cnf` in the MySQL `conf.d` directory. This translates the unique, stable identity provided by the StatefulSet controller into the domain of MySQL server IDs, which require the same properties.

The script in the `init-mysql` container also applies either `master.cnf` or `slave.cnf` from the ConfigMap by copying the contents into `conf.d`. Because

the example topology consists of a single MySQL master and any number of slaves, the script simply assigns ordinal 0 to be the master, and everyone else to be slaves. Combined with the StatefulSet controller's [deployment order guarantee](#), this ensures the MySQL master is Ready before creating slaves, so they can begin replicating.

## Cloning existing data

In general, when a new Pod joins the set as a slave, it must assume the MySQL master might already have data on it. It also must assume that the replication logs might not go all the way back to the beginning of time. These conservative assumptions are the key to allow a running StatefulSet to scale up and down over time, rather than being fixed at its initial size.

The second Init Container, named `clone-mysql`, performs a clone operation on a slave Pod the first time it starts up on an empty PersistentVolume. That means it copies all existing data from another running Pod, so its local state is consistent enough to begin replicating from the master.

MySQL itself does not provide a mechanism to do this, so the example uses a popular open-source tool called Percona XtraBackup. During the clone, the source MySQL server might suffer reduced performance. To minimize impact on the MySQL master, the script instructs each Pod to clone from the Pod whose ordinal index is one lower. This works because the StatefulSet controller always ensures Pod N is Ready before starting Pod N+1.

## Starting replication

After the Init Containers complete successfully, the regular containers run. The MySQL Pods consist of a `mysql` container that runs the actual `mysqld` server, and an `xtrabackup` container that acts as a [sidecar](#).

The `xtrabackup` sidecar looks at the cloned data files and determines if it's necessary to initialize MySQL replication on the slave. If so, it waits for `mysqld` to be ready and then executes the `CHANGE MASTER TO` and `START SLAVE` commands with replication parameters extracted from the XtraBackup clone files.

Once a slave begins replication, it remembers its MySQL master and reconnects automatically if the server restarts or the connection dies. Also, because slaves look for the master at its stable DNS name (`mysql-0.mysql`), they automatically find the master even if it gets a new Pod IP due to being rescheduled.

Lastly, after starting replication, the `xtrabackup` container listens for connections from other Pods requesting a data clone. This server remains up indefinitely in case the StatefulSet scales up, or in case the next Pod loses its PersistentVolumeClaim and needs to redo the clone.

# Sending client traffic

You can send test queries to the MySQL master (hostname `mysql-0.mysql`) by running a temporary container with the `mysql:5.7` image and running the `mysql` client binary.

```
kubectl run mysql-client --image=mysql:5.7 -i --rm --restart=Never -- \
  mysql -h mysql-0.mysql <<EOF
CREATE DATABASE test;
CREATE TABLE test.messages (message VARCHAR(250));
INSERT INTO test.messages VALUES ('hello');
EOF
```

Use the hostname `mysql-read` to send test queries to any server that reports being Ready:

```
kubectl run mysql-client --image=mysql:5.7 -i -t --rm --restart=Never -- \
  mysql -h mysql-read -e "SELECT * FROM test.messages"
```

You should get output like this:

```
Waiting for pod default/mysql-client to be running, status is
Pending, pod ready: false
+-----+
| message |
+-----+
| hello   |
+-----+
pod "mysql-client" deleted
```

To demonstrate that the `mysql-read` Service distributes connections across servers, you can run `SELECT @@server_id` in a loop:

```
kubectl run mysql-client-loop --image=mysql:5.7 -i -t --rm --
restart=Never -- \
  bash -ic "while sleep 1; do mysql -h mysql-read -e 'SELECT
@@server_id,NOW()'; done"
```

You should see the reported `@@server_id` change randomly, because a different endpoint might be selected upon each connection attempt:

```
+-----+-----+
| @@server_id | NOW()           |
+-----+-----+
|      100  | 2006-01-02 15:04:05 |
+-----+-----+
+-----+-----+
| @@server_id | NOW()           |
+-----+-----+
|      102  | 2006-01-02 15:04:06 |
```

```
+-----+-----+
+-----+-----+
| @server_id | NOW()           |
+-----+-----+
|      101 | 2006-01-02 15:04:07 |
+-----+-----+
```

You can press **Ctrl+C** when you want to stop the loop, but it's useful to keep it running in another window so you can see the effects of the following steps.

## Simulating Pod and Node downtime

To demonstrate the increased availability of reading from the pool of slaves instead of a single server, keep the `SELECT @@server_id` loop from above running while you force a Pod out of the Ready state.

### Break the Readiness Probe

The [readiness probe](#) for the `mysql` container runs the command `mysql -h 127.0.0.1 -e 'SELECT 1'` to make sure the server is up and able to execute queries.

One way to force this readiness probe to fail is to break that command:

```
kubectl exec mysql-2 -c mysql -- mv /usr/bin/mysql /usr/bin/mysql.off
```

This reaches into the actual container's filesystem for Pod `mysql-2` and renames the `mysql` command so the readiness probe can't find it. After a few seconds, the Pod should report one of its containers as not Ready, which you can check by running:

```
kubectl get pod mysql-2
```

Look for 1/2 in the READY column:

NAME	READY	STATUS	RESTARTS	AGE
mysql-2	1/2	Running	0	3m

At this point, you should see your `SELECT @@server_id` loop continue to run, although it never reports 102 anymore. Recall that the `init-mysql` script defined `server-id` as `100 + $ordinal`, so server ID 102 corresponds to Pod `mysql-2`.

Now repair the Pod and it should reappear in the loop output after a few seconds:

```
kubectl exec mysql-2 -c mysql -- mv /usr/bin/mysql.off /usr/bin/mysql
```

## Delete Pods

The StatefulSet also recreates Pods if they're deleted, similar to what a ReplicaSet does for stateless Pods.

```
kubectl delete pod mysql-2
```

The StatefulSet controller notices that no `mysql-2` Pod exists anymore, and creates a new one with the same name and linked to the same PersistentVolumeClaim. You should see server ID 102 disappear from the loop output for a while and then return on its own.

## Drain a Node

If your Kubernetes cluster has multiple Nodes, you can simulate Node downtime (such as when Nodes are upgraded) by issuing a [drain](#).

First determine which Node one of the MySQL Pods is on:

```
kubectl get pod mysql-2 -o wide
```

The Node name should show up in the last column:

NAME	READY	STATUS	RESTARTS	AGE	IP
NODE					
mysql-2	2/2	Running	0	15m	10.244.5.27
kubernetes-node-9l2t					

Then drain the Node by running the following command, which cordons it so no new Pods may schedule there, and then evicts any existing Pods. Replace `<node-name>` with the name of the Node you found in the last step.

This might impact other applications on the Node, so it's best to **only do this in a test cluster**.

```
kubectl drain <node-name> --force --delete-local-data --ignore-daemonsets
```

Now you can watch as the Pod reschedules on a different Node:

```
kubectl get pod mysql-2 -o wide --watch
```

It should look something like this:

NAME	READY	STATUS	RESTARTS	AGE
IP		NODE		
mysql-2	2/2	Terminating	0	15m
10.244.1.56		kubernetes-node-9l2t		
[...]				
mysql-2	0/2	Pending	0	0s
<none>		kubernetes-node-fjlm		
mysql-2	0/2	Init:0/2	0	0s
<none>		kubernetes-node-fjlm		

```
mysql-2  0/2    Init:1/2      0          20s
10.244.5.32  kubernetes-node-fjlm
mysql-2  0/2    PodInitializing 0        21s
10.244.5.32  kubernetes-node-fjlm
mysql-2  1/2    Running      0          22s
10.244.5.32  kubernetes-node-fjlm
mysql-2  2/2    Running      0          30s
10.244.5.32  kubernetes-node-fjlm
```

And again, you should see server ID 102 disappear from the SELECT @`server_id` loop output for a while and then return.

Now uncordon the Node to return it to a normal state:

```
kubectl uncordon <node-name>
```

## Scaling the number of slaves

With MySQL replication, you can scale your read query capacity by adding slaves. With StatefulSet, you can do this with a single command:

```
kubectl scale statefulset mysql --replicas=5
```

Watch the new Pods come up by running:

```
kubectl get pods -l app=mysql --watch
```

Once they're up, you should see server IDs 103 and 104 start appearing in the SELECT @`server_id` loop output.

You can also verify that these new servers have the data you added before they existed:

```
kubectl run mysql-client --image=mysql:5.7 -i -t --rm --restart=Never -- \
  mysql -h mysql-3.mysql -e "SELECT * FROM test.messages"
```

```
Waiting for pod default/mysql-client to be running, status is
Pending, pod ready: false
+-----+
| message |
+-----+
| hello   |
+-----+
pod "mysql-client" deleted
```

Scaling back down is also seamless:

```
kubectl scale statefulset mysql --replicas=3
```

Note, however, that while scaling up creates new PersistentVolumeClaims automatically, scaling down does not automatically delete these PVCs. This

gives you the choice to keep those initialized PVCs around to make scaling back up quicker, or to extract data before deleting them.

You can see this by running:

```
kubectl get pvc -l app=mysql
```

Which shows that all 5 PVCs still exist, despite having scaled the StatefulSet down to 3:

NAME	STATUS	VOLUME	CAPACITY
ACCESSMODES	AGE		
data-mysql-0	Bound	pvc-8acb5dc-b103-11e6-93fa-42010a800002	10Gi RW0 20m
data-mysql-1	Bound	pvc-8ad39820-b103-11e6-93fa-42010a800002	10Gi RW0 20m
data-mysql-2	Bound	pvc-8ad69a6d-b103-11e6-93fa-42010a800002	10Gi RW0 20m
data-mysql-3	Bound	pvc-50043c45-b1c5-11e6-93fa-42010a800002	10Gi RW0 2m
data-mysql-4	Bound	pvc-500a9957-b1c5-11e6-93fa-42010a800002	10Gi RW0 2m

If you don't intend to reuse the extra PVCs, you can delete them:

```
kubectl delete pvc data-mysql-3  
kubectl delete pvc data-mysql-4
```

## Cleaning up

1. Cancel the `SELECT @@server_id` loop by pressing **Ctrl+C** in its terminal, or running the following from another terminal:

```
kubectl delete pod mysql-client-loop --now
```

2. Delete the StatefulSet. This also begins terminating the Pods.

```
kubectl delete statefulset mysql
```

3. Verify that the Pods disappear. They might take some time to finish terminating.

```
kubectl get pods -l app=mysql
```

You'll know the Pods have terminated when the above returns:

```
No resources found.
```

1. Delete the ConfigMap, Services, and PersistentVolumeClaims.

```
kubectl delete configmap,service,pvc -l app=mysql
```

2. If you manually provisioned PersistentVolumes, you also need to manually delete them, as well as release the underlying resources. If you used a dynamic provisioner, it automatically deletes the PersistentVolumes when it sees that you deleted the PersistentVolumeClaims. Some dynamic provisioners (such as those for EBS and PD) also release the underlying resources upon deleting the PersistentVolumes.

## What's next

- Learn more about [scaling a StatefulSet](#).
- Learn more about [debugging a StatefulSet](#).
- Learn more about [deleting a StatefulSet](#).
- Learn more about [force deleting StatefulSet Pods](#).
- Look in the [Helm Charts repository](#) for other stateful application examples.

## Feedback

Was this page helpful?

Yes No

Thanks for the feedback. If you have a specific, answerable question about how to use Kubernetes, ask it on [Stack Overflow](#). Open an issue in the GitHub repo if you want to [report a problem](#) or [suggest an improvement](#).

---

[Create an Issue](#) [Edit This Page](#)

Page last modified on February 15, 2020 at 3:27 PM PST by [Correct note shortcode for run-replicated-stateful-application.md \(#19132\)](#) ([Page History](#))

[Edit This Page](#)

# Update API Objects in Place Using kubectl patch

This task shows how to use `kubectl patch` to update an API object in place. The exercises in this task demonstrate a strategic merge patch and a JSON merge patch.

- [Before you begin](#)
- [Use a strategic merge patch to update a Deployment](#)

- [Use a JSON merge patch to update a Deployment](#)
- [Alternate forms of the kubectl patch command](#)
- [Summary](#)
- [What's next](#)

## Before you begin

You need to have a Kubernetes cluster, and the kubectl command-line tool must be configured to communicate with your cluster. 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`.

## Use a strategic merge patch to update a Deployment

Here's the configuration file for a Deployment that has two replicas. Each replica is a Pod that has one container:

### [application/deployment-patch.yaml](#)

```
apiVersion: apps/v1 # for versions before 1.9.0 use apps/v1beta2
kind: Deployment
metadata:
  name: patch-demo
spec:
  replicas: 2
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
        - name: patch-demo-ctr
          image: nginx
      tolerations:
        - effect: NoSchedule
          key: dedicated
          value: test-team
```

Create the Deployment:

```
kubectl apply -f https://k8s.io/examples/application/deployment-patch.yaml
```

View the Pods associated with your Deployment:

```
kubectl get pods
```

The output shows that the Deployment has two Pods. The 1/1 indicates that each Pod has one container:

NAME	READY	STATUS	RESTARTS	AGE
patch-demo-28633765-670qr	1/1	Running	0	23s
patch-demo-28633765-j5qs3	1/1	Running	0	23s

Make a note of the names of the running Pods. Later, you will see that these Pods get terminated and replaced by new ones.

At this point, each Pod has one Container that runs the nginx image. Now suppose you want each Pod to have two containers: one that runs nginx and one that runs redis.

Create a file named `patch-file.yaml` that has this content:

```
spec:
  template:
    spec:
      containers:
        - name: patch-demo-ctr-2
          image: redis
```

Patch your Deployment:

- [Bash](#)
- [PowerShell](#)

```
kubectl patch deployment patch-demo --patch "$(cat patch-file.yaml)"
```

[Edit This Page](#)

## Scale a StatefulSet

This task shows how to scale a StatefulSet. Scaling a StatefulSet refers to increasing or decreasing the number of replicas.

- [Before you begin](#)
- [Scaling StatefulSets](#)
- [Troubleshooting](#)
- [What's next](#)

# Before you begin

- StatefulSets are only available in Kubernetes version 1.5 or later. To check your version of Kubernetes, run `kubectl version`.
- Not all stateful applications scale nicely. If you are unsure about whether to scale your StatefulSets, see [StatefulSet concepts](#) or [StatefulSet tutorial](#) for further information.
- You should perform scaling only when you are confident that your stateful application cluster is completely healthy.

## Scaling StatefulSets

### Use `kubectl` to scale StatefulSets

First, find the StatefulSet you want to scale.

```
kubectl get statefulsets <stateful-set-name>
```

Change the number of replicas of your StatefulSet:

```
kubectl scale statefulsets <stateful-set-name> --replicas=<new-replicas>
```

### Make in-place updates on your StatefulSets

Alternatively, you can do [in-place updates](#) on your StatefulSets.

If your StatefulSet was initially created with `kubectl apply`, update `.spec.replicas` of the StatefulSet manifests, and then do a `kubectl apply`:

```
kubectl apply -f <stateful-set-file-updated>
```

Otherwise, edit that field with `kubectl edit`:

```
kubectl edit statefulsets <stateful-set-name>
```

Or use `kubectl patch`:

```
kubectl patch statefulsets <stateful-set-name> -p '{"spec": {"replicas":<new-replicas>}}'
```

## Troubleshooting

### Scaling down does not work right

You cannot scale down a StatefulSet when any of the stateful Pods it manages is unhealthy. Scaling down only takes place after those stateful Pods become running and ready.

If `spec.replicas > 1`, Kubernetes cannot determine the reason for an unhealthy Pod. It might be the result of a permanent fault or of a transient fault. A transient fault can be caused by a restart required by upgrading or maintenance.

If the Pod is unhealthy due to a permanent fault, scaling without correcting the fault may lead to a state where the StatefulSet membership drops below a certain minimum number of replicas that are needed to function correctly. This may cause your StatefulSet to become unavailable.

If the Pod is unhealthy due to a transient fault and the Pod might become available again, the transient error may interfere with your scale-up or scale-down operation. Some distributed databases have issues when nodes join and leave at the same time. It is better to reason about scaling operations at the application level in these cases, and perform scaling only when you are sure that your stateful application cluster is completely healthy.

## What's next

- Learn more about [deleting a StatefulSet](#).

## Feedback

Was this page helpful?

Yes No

Thanks for the feedback. If you have a specific, answerable question about how to use Kubernetes, ask it on [Stack Overflow](#). Open an issue in the GitHub repo if you want to [report a problem](#) or [suggest an improvement](#).

---

[Create an Issue](#) [Edit This Page](#)

Page last modified on March 25, 2019 at 5:06 PM PST by [Official 1.14](#)  
[Release Docs \(#13174\)](#) ([Page History](#))

[Edit This Page](#)

## Delete a StatefulSet

This task shows you how to delete a [StatefulSetManages the deployment and scaling of a set of Pods, and provides guarantees about the ordering and uniqueness of these Pods.](#).

- [Before you begin](#)
- [Deleting a StatefulSet](#)
- [What's next](#)

# Before you begin

- This task assumes you have an application running on your cluster represented by a StatefulSet.

## Deleting a StatefulSet

You can delete a StatefulSet in the same way you delete other resources in Kubernetes: use the `kubectl delete` command, and specify the StatefulSet either by file or by name.

```
kubectl delete -f <file.yaml>
```

```
kubectl delete statefulsets <statefulset-name>
```

You may need to delete the associated headless service separately after the StatefulSet itself is deleted.

```
kubectl delete service <service-name>
```

Deleting a StatefulSet through `kubectl` will scale it down to 0, thereby deleting all pods that are a part of it. If you want to delete just the StatefulSet and not the pods, use `--cascade=false`.

```
kubectl delete -f <file.yaml> --cascade=false
```

By passing `--cascade=false` to `kubectl delete`, the Pods managed by the StatefulSet are left behind even after the StatefulSet object itself is deleted. If the pods have a label `app=myapp`, you can then delete them as follows:

```
kubectl delete pods -l app=myapp
```

## Persistent Volumes

Deleting the Pods in a StatefulSet will not delete the associated volumes. This is to ensure that you have the chance to copy data off the volume before deleting it. Deleting the PVC after the pods have left the [terminating state](#) might trigger deletion of the backing Persistent Volumes depending on the storage class and reclaim policy. You should never assume ability to access a volume after claim deletion.

**Note:** Use caution when deleting a PVC, as it may lead to data loss.

## Complete deletion of a StatefulSet

To simply delete everything in a StatefulSet, including the associated pods, you can run a series of commands similar to the following:

```
grace=$(kubectl get pods <stateful-set-pod> --template '{{.spec.terminationGracePeriodSeconds}}')
```

```
kubectl delete statefulset -l app=myapp  
sleep $grace  
kubectl delete pvc -l app=myapp
```

In the example above, the Pods have the label `app=myapp`; substitute your own label as appropriate.

## Force deletion of StatefulSet pods

If you find that some pods in your StatefulSet are stuck in the ‘Terminating’ or ‘Unknown’ states for an extended period of time, you may need to manually intervene to forcefully delete the pods from the apiserver. This is a potentially dangerous task. Refer to [Force Delete StatefulSet Pods](#) for details.

## What's next

Learn more about [force deleting StatefulSet Pods](#).

## Feedback

Was this page helpful?

Yes No

Thanks for the feedback. If you have a specific, answerable question about how to use Kubernetes, ask it on [Stack Overflow](#). Open an issue in the GitHub repo if you want to [report a problem](#) or [suggest an improvement](#).

---

[Create an Issue](#) [Edit This Page](#)

Page last modified on May 31, 2019 at 1:42 AM PST by [Add relevant glossary tooltips \(#14632\)](#) ([Page History](#))

[Edit This Page](#)

## Force Delete StatefulSet Pods

This page shows how to delete Pods which are part of a [stateful setManages the deployment and scaling of a set of Pods, and provides guarantees about the ordering and uniqueness of these Pods.](#), and explains the considerations to keep in mind when doing so.

- [Before you begin](#)
- [StatefulSet considerations](#)
- [Delete Pods](#)

- [What's next](#)

## Before you begin

- This is a fairly advanced task and has the potential to violate some of the properties inherent to StatefulSet.
- Before proceeding, make yourself familiar with the considerations enumerated below.

## StatefulSet considerations

In normal operation of a StatefulSet, there is **never** a need to force delete a StatefulSet Pod. The [StatefulSet controller](#) is responsible for creating, scaling and deleting members of the StatefulSet. It tries to ensure that the specified number of Pods from ordinal 0 through N-1 are alive and ready. StatefulSet ensures that, at any time, there is at most one Pod with a given identity running in a cluster. This is referred to as *at most one* semantics provided by a StatefulSet.

Manual force deletion should be undertaken with caution, as it has the potential to violate the at most one semantics inherent to StatefulSet. StatefulSets may be used to run distributed and clustered applications which have a need for a stable network identity and stable storage. These applications often have configuration which relies on an ensemble of a fixed number of members with fixed identities. Having multiple members with the same identity can be disastrous and may lead to data loss (e.g. split brain scenario in quorum-based systems).

## Delete Pods

You can perform a graceful pod deletion with the following command:

```
kubectl delete pods <pod>
```

For the above to lead to graceful termination, the Pod **must not** specify a `pod.Spec.TerminationGracePeriodSeconds` of 0. The practice of setting a `pod.Spec.TerminationGracePeriodSeconds` of 0 seconds is unsafe and strongly discouraged for StatefulSet Pods. Graceful deletion is safe and will ensure that the [Pod shuts down gracefully](#) before the kubelet deletes the name from the apiserver.

Kubernetes (versions 1.5 or newer) will not delete Pods just because a Node is unreachable. The Pods running on an unreachable Node enter the ‘Terminating’ or ‘Unknown’ state after a [timeout](#). Pods may also enter these states when the user attempts graceful deletion of a Pod on an unreachable Node. The only ways in which a Pod in such a state can be removed from the apiserver are as follows:

- The Node object is deleted (either by you, or by the [Node Controller](#)).

- The kubelet on the unresponsive Node starts responding, kills the Pod and removes the entry from the apiserver.
- Force deletion of the Pod by the user.

The recommended best practice is to use the first or second approach. If a Node is confirmed to be dead (e.g. permanently disconnected from the network, powered down, etc), then delete the Node object. If the Node is suffering from a network partition, then try to resolve this or wait for it to resolve. When the partition heals, the kubelet will complete the deletion of the Pod and free up its name in the apiserver.

Normally, the system completes the deletion once the Pod is no longer running on a Node, or the Node is deleted by an administrator. You may override this by force deleting the Pod.

## Force Deletion

Force deletions **do not** wait for confirmation from the kubelet that the Pod has been terminated. Irrespective of whether a force deletion is successful in killing a Pod, it will immediately free up the name from the apiserver. This would let the StatefulSet controller create a replacement Pod with that same identity; this can lead to the duplication of a still-running Pod, and if said Pod can still communicate with the other members of the StatefulSet, will violate the at most one semantics that StatefulSet is designed to guarantee.

When you force delete a StatefulSet pod, you are asserting that the Pod in question will never again make contact with other Pods in the StatefulSet and its name can be safely freed up for a replacement to be created.

If you want to delete a Pod forcibly using kubectl version  $\geq 1.5$ , do the following:

```
kubectl delete pods <pod> --grace-period=0 --force
```

If you're using any version of kubectl  $\leq 1.4$ , you should omit the `--force` option and use:

```
kubectl delete pods <pod> --grace-period=0
```

If even after these commands the pod is stuck on Unknown state, use the following command to remove the pod from the cluster:

```
kubectl patch pod <pod> -p '{"metadata":{"finalizers":null}}'
```

Always perform force deletion of StatefulSet Pods carefully and with complete knowledge of the risks involved.

## What's next

Learn more about [debugging a StatefulSet](#).

# Feedback

Was this page helpful?

Yes No

Thanks for the feedback. If you have a specific, answerable question about how to use Kubernetes, ask it on [Stack Overflow](#). Open an issue in the GitHub repo if you want to [report a problem](#) or [suggest an improvement](#).

---

[Create an Issue](#) [Edit This Page](#)

Page last modified on October 15, 2019 at 6:39 AM PST by [fix url \(#16846\)](#)  
[\(Page History\)](#)

[Edit This Page](#)

# Perform Rolling Update Using a Replication Controller

**Note:** The preferred way to create a replicated application is to use a [Deployment](#), which in turn uses a [ReplicaSet](#). For more information, see [Running a Stateless Application Using a Deployment](#).

To update a service without an outage, `kubectl` supports what is called [rolling update](#), which updates one pod at a time, rather than taking down the entire service at the same time. See the [rolling update design document](#) for more information.

Note that `kubectl rolling-update` only supports Replication Controllers. However, if you deploy applications with Replication Controllers, consider switching them to [Deployments](#). A Deployment is a higher-level controller that automates rolling updates of applications declaratively, and therefore is recommended. If you still want to keep your Replication Controllers and use `kubectl rolling-update`, keep reading:

A rolling update applies changes to the configuration of pods being managed by a replication controller. The changes can be passed as a new replication controller configuration file; or, if only updating the image, a new container image can be specified directly.

A rolling update works by:

1. Creating a new replication controller with the updated configuration.
2. Increasing/decreasing the replica count on the new and old controllers until the correct number of replicas is reached.
3. Deleting the original replication controller.

Rolling updates are initiated with the `kubectl rolling-update` command:

```
kubectl rolling-update NAME NEW_NAME --image=IMAGE:TAG
```

```
# or read the configuration from a file  
kubectl rolling-update NAME -f FILE
```

- [Passing a configuration file](#)
- [Updating the container image](#)
- [Required and optional fields](#)
- [Walkthrough](#)
- [Troubleshooting](#)

## Passing a configuration file

To initiate a rolling update using a configuration file, pass the new file to `kubectl rolling-update`:

```
kubectl rolling-update NAME -f FILE
```

The configuration file must:

- Specify a different `metadata.name` value.
- Overwrite at least one common label in its `spec.selector` field.
- Use the same `metadata.namespace`.

Replication controller configuration files are described in [Creating Replication Controllers](#).

## Examples

```
# Update pods of frontend-v1 using new replication controller data in frontend-v2.json.
```

```
kubectl rolling-update frontend-v1 -f frontend-v2.json
```

```
# Update pods of frontend-v1 using JSON data passed into stdin.
```

```
cat frontend-v2.json | kubectl rolling-update frontend-v1 -f -
```

## Updating the container image

To update only the container image, pass a new image name and tag with the `--image` flag and (optionally) a new controller name:

```
kubectl rolling-update NAME NEW_NAME --image=IMAGE:TAG
```

The `--image` flag is only supported for single-container pods. Specifying `--image` with multi-container pods returns an error.

If you didn't specify a new name, this creates a new replication controller with a temporary name. Once the rollout is complete, the old controller is deleted, and the new controller is updated to use the original name.

The update will fail if `IMAGE:TAG` is identical to the current value. For this reason, we recommend the use of versioned tags as opposed to values such as `:latest`. Doing a rolling update from `image:latest` to a new `image:latest` will fail, even if the image at that tag has changed. Moreover, the use of `:latest` is not recommended, see [Best Practices for Configuration](#) for more information.

## Examples

```
# Update the pods of frontend-v1 to frontend-v2
```

```
kubectl rolling-update frontend-v1 frontend-v2 --image=image:v2
```

```
# Update the pods of frontend, keeping the replication controller name
```

```
kubectl rolling-update frontend --image=image:v2
```

# Required and optional fields

Required fields are:

- `NAME`: The name of the replication controller to update.

as well as either:

- `-f FILE`: A replication controller configuration file, in either JSON or YAML format. The configuration file must specify a new top-level `id` value and include at least one of the existing `spec.selector` key:value pairs. See the [Run Stateless AP Replication Controller](#) page for details.

or:

- `--image IMAGE:TAG`: The name and tag of the image to update to. Must be different than the current image:tag currently specified.

Optional fields are:

- `NEW_NAME`: Only used in conjunction with `--image` (not with `-f FILE`). The name to assign to the new replication controller.
- `--poll-interval DURATION`: The time between polling the controller status after update. Valid units are ns (nanoseconds), us or  $\mu$ s (microseconds), ms (milliseconds), s (seconds), m (minutes), or h (hours). Units can be combined (e.g. `1m30s`). The default is `3s`.
- `--timeout DURATION`: The maximum time to wait for the controller to update a pod before exiting. Default is `5m0s`. Valid units are as described for `--poll-interval` above.
- `--update-period DURATION`: The time to wait between updating pods. Default is `1m0s`. Valid units are as described for `--poll-interval` above.

Additional information about the `kubectl rolling-update` command is available from the [kubectl reference](#).

## Walkthrough

Let's say you were running version 1.7.9 of nginx:

## [controllers/replication-nginx-1.7.9.yaml](#)

```
apiVersion: v1
kind: ReplicationController
metadata:
  name: my-nginx
spec:
  replicas: 5
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
        - name: nginx
          image: nginx:1.7.9
          ports:
            - containerPort: 80
```

To update to version 1.9.1, you can use [kubectl rolling-update --image](#) to specify the new image:

```
kubectl rolling-update my-nginx --image=nginx:1.9.1
```

```
Created my-nginx-ccba8fdb8cc8160970f63f9a2696fc46
```

In another window, you can see that kubectl added a deployment label to the pods, whose value is a hash of the configuration, to distinguish the new pods from the old:

```
kubectl get pods -l app=nginx -L deployment
```

NAME	READY		
STATUS	RESTARTS	AGE	DEPLOYMENT
my-nginx-ccba8fdb8cc8160970f63f9a2696fc46-k156z	1/1		
Running 0	1m		ccba8fdb8cc8160970f63f9a2696fc46
my-nginx-ccba8fdb8cc8160970f63f9a2696fc46-v95yh	1/1		
Running 0	35s		ccba8fdb8cc8160970f63f9a2696fc46
my-nginx-divi2	1/1		
Running 0	2h		2d1d7a8f682934a254002b56404b813e
my-nginx-o0ef1	1/1		
Running 0	2h		2d1d7a8f682934a254002b56404b813e
my-nginx-q6all	1/1		
Running 0	8m		2d1d7a8f682934a254002b56404b813e

kubectl rolling-update reports progress as it progresses:

```
Scaling up my-nginx-ccba8fdb8cc8160970f63f9a2696fc46 from 0 to 3, scaling down my-nginx from 3 to 0 (keep 3 pods available, don't exceed 4 pods)
Scaling my-nginx-ccba8fdb8cc8160970f63f9a2696fc46 up to 1
```

```
Scaling my-nginx down to 2
Scaling my-nginx-ccba8fdb8cc8160970f63f9a2696fc46 up to 2
Scaling my-nginx down to 1
Scaling my-nginx-ccba8fdb8cc8160970f63f9a2696fc46 up to 3
Scaling my-nginx down to 0
Update succeeded. Deleting old controller: my-nginx
Renaming my-nginx-ccba8fdb8cc8160970f63f9a2696fc46 to my-nginx
replicationcontroller "my-nginx" rolling updated
```

If you encounter a problem, you can stop the rolling update midway and revert to the previous version using `--rollback`:

```
kubectl rolling-update my-nginx --rollback
```

```
Setting "my-nginx" replicas to 1
Continuing update with existing controller my-nginx.
Scaling up nginx from 1 to 1, scaling down my-nginx-
ccba8fdb8cc8160970f63f9a2696fc46 from 1 to 0 (keep 1 pods
available, don't exceed 2 pods)
Scaling my-nginx-ccba8fdb8cc8160970f63f9a2696fc46 down to 0
Update succeeded. Deleting my-nginx-
ccba8fdb8cc8160970f63f9a2696fc46
replicationcontroller "my-nginx" rolling updated
```

This is one example where the immutability of containers is a huge asset.

If you need to update more than just the image (e.g., command arguments, environment variables), you can create a new replication controller, with a new name and distinguishing label value, such as:

## [controllers/replication-nginx-1.9.2.yaml](#)

```
apiVersion: v1
kind: ReplicationController
metadata:
  name: my-nginx-v4
spec:
  replicas: 5
  selector:
    app: nginx
    deployment: v4
  template:
    metadata:
      labels:
        app: nginx
        deployment: v4
    spec:
      containers:
        - name: nginx
          image: nginx:1.9.2
          args: ["nginx", "-T"]
          ports:
            - containerPort: 80
```

and roll it out:

```
# Assuming you named the file "my-nginx.yaml"
kubectl rolling-update my-nginx -f ./my-nginx.yaml
```

```
Created my-nginx-v4
Scaling up my-nginx-v4 from 0 to 5, scaling down my-nginx from 4
to 0 (keep 4 pods available, don't exceed 5 pods)
Scaling my-nginx-v4 up to 1
Scaling my-nginx down to 3
Scaling my-nginx-v4 up to 2
Scaling my-nginx down to 2
Scaling my-nginx-v4 up to 3
Scaling my-nginx down to 1
Scaling my-nginx-v4 up to 4
Scaling my-nginx down to 0
Scaling my-nginx-v4 up to 5
Update succeeded. Deleting old controller: my-nginx
replicationcontroller "my-nginx-v4" rolling updated
```

## Troubleshooting

If the `timeout` duration is reached during a rolling update, the operation will fail with some pods belonging to the new replication controller, and some to the original controller.

To continue the update from where it failed, retry using the same command.

To roll back to the original state before the attempted update, append the `--rollback=true` flag to the original command. This will revert all changes.

## Feedback

Was this page helpful?

Yes No

Thanks for the feedback. If you have a specific, answerable question about how to use Kubernetes, ask it on [Stack Overflow](#). Open an issue in the GitHub repo if you want to [report a problem](#) or [suggest an improvement](#).

---

Page last modified on October 13, 2019 at 4:58 PM PST by [fix url to ReplicationController \(#16847\)](#) ([Page History](#))

[Edit This Page](#)

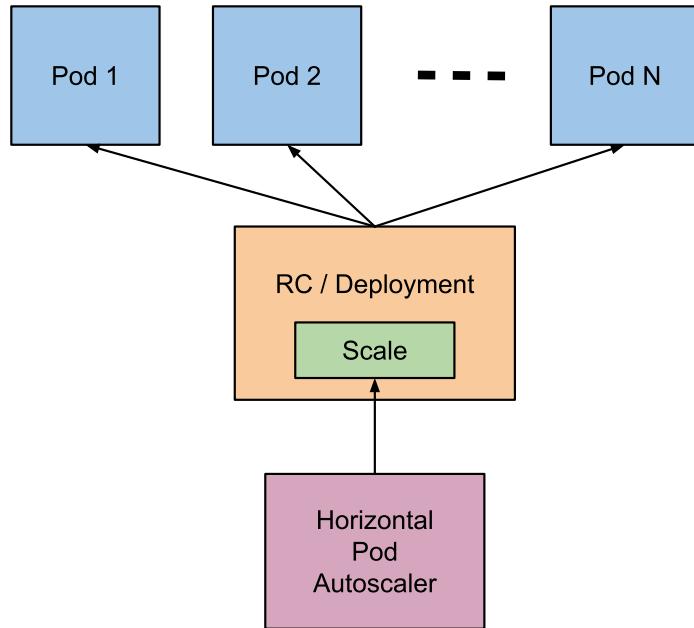
# Horizontal Pod Autoscaler

The Horizontal Pod Autoscaler automatically scales the number of pods in a replication controller, deployment, replica set or stateful set based on observed CPU utilization (or, with [custom metrics](#) support, on some other application-provided metrics). Note that Horizontal Pod Autoscaling does not apply to objects that can't be scaled, for example, DaemonSets.

The Horizontal Pod Autoscaler is implemented as a Kubernetes API resource and a controller. The resource determines the behavior of the controller. The controller periodically adjusts the number of replicas in a replication controller or deployment to match the observed average CPU utilization to the target specified by user.

- [How does the Horizontal Pod Autoscaler work?](#)
- [API Object](#)
- [Support for Horizontal Pod Autoscaler in kubectl](#)
- [Autoscaling during rolling update](#)
- [Support for cooldown/delay](#)
- [Support for multiple metrics](#)
- [Support for custom metrics](#)
- [Support for metrics APIs](#)
- [What's next](#)

# How does the Horizontal Pod Autoscaler work?



The Horizontal Pod Autoscaler is implemented as a control loop, with a period controlled by the controller manager's `--horizontal-pod-autoscaler-sync-period` flag (with a default value of 15 seconds).

During each period, the controller manager queries the resource utilization against the metrics specified in each `HorizontalPodAutoscaler` definition. The controller manager obtains the metrics from either the resource metrics API (for per-pod resource metrics), or the custom metrics API (for all other metrics).

- For per-pod resource metrics (like CPU), the controller fetches the metrics from the resource metrics API for each pod targeted by the `HorizontalPodAutoscaler`. Then, if a target utilization value is set, the controller calculates the utilization value as a percentage of the equivalent resource request on the containers in each pod. If a target raw value is set, the raw metric values are used directly. The controller then takes the mean of the utilization or the raw value (depending on the type of target specified) across all targeted pods, and produces a ratio used to scale the number of desired replicas.

Please note that if some of the pod's containers do not have the relevant resource request set, CPU utilization for the pod will not be defined and the

autoscaler will not take any action for that metric. See the [algorithm details](#) section below for more information about how the autoscaling algorithm works.

- For per-pod custom metrics, the controller functions similarly to per-pod resource metrics, except that it works with raw values, not utilization values.
- For object metrics and external metrics, a single metric is fetched, which describes the object in question. This metric is compared to the target value, to produce a ratio as above. In the `autoscaling/v2beta2` API version, this value can optionally be divided by the number of pods before the comparison is made.

The `HorizontalPodAutoscaler` normally fetches metrics from a series of aggregated APIs (`metrics.k8s.io`, `custom.metrics.k8s.io`, and `external.metrics.k8s.io`). The `metrics.k8s.io` API is usually provided by `metrics-server`, which needs to be launched separately. See [metrics-server](#) for instructions. The `HorizontalPodAutoscaler` can also fetch metrics directly from Heapster.

**Note:**

**FEATURE STATE:** Kubernetes 1.11 [deprecated](#)

This feature is *deprecated*. For more information on this state, see the [Kubernetes Deprecation Policy](#).

Fetching metrics from Heapster is deprecated as of Kubernetes 1.11.

See [Support for metrics APIs](#) for more details.

The autoscaler accesses corresponding scalable controllers (such as replication controllers, deployments, and replica sets) by using the `scale` sub-resource. `Scale` is an interface that allows you to dynamically set the number of replicas and examine each of their current states. More details on `scale` sub-resource can be found [here](#).

## Algorithm Details

From the most basic perspective, the Horizontal Pod Autoscaler controller operates on the ratio between desired metric value and current metric value:

```
desiredReplicas = ceil[currentReplicas * ( currentMetricValue /  
desiredMetricValue )]
```

For example, if the current metric value is `200m`, and the desired value is `100m`, the number of replicas will be doubled, since `200.0 / 100.0 == 2.0`. If the current value is instead `50m`, we'll halve the number of replicas, since `50.0 / 100.0 == 0.5`. We'll skip scaling if the ratio is sufficiently close to `1.0` (within a globally-configurable tolerance, from the `--horizontal-pod-autoscaler-tolerance` flag, which defaults to `0.1`).

When a `targetAverageValue` or `targetAverageUtilization` is specified, the `currentMetricValue` is computed by taking the average of the given metric across all Pods in the HorizontalPodAutoscaler's scale target. Before checking the tolerance and deciding on the final values, we take pod readiness and missing metrics into consideration, however.

All Pods with a deletion timestamp set (i.e. Pods in the process of being shut down) and all failed Pods are discarded.

If a particular Pod is missing metrics, it is set aside for later; Pods with missing metrics will be used to adjust the final scaling amount.

When scaling on CPU, if any pod has yet to become ready (i.e. it's still initializing) *or* the most recent metric point for the pod was before it became ready, that pod is set aside as well.

Due to technical constraints, the HorizontalPodAutoscaler controller cannot exactly determine the first time a pod becomes ready when determining whether to set aside certain CPU metrics. Instead, it considers a Pod "not yet ready" if it's unready and transitioned to unready within a short, configurable window of time since it started. This value is configured with the `--horizontal-pod-autoscaler-initial-readiness-delay` flag, and its default is 30 seconds. Once a pod has become ready, it considers any transition to ready to be the first if it occurred within a longer, configurable time since it started. This value is configured with the `--horizontal-pod-autoscaler-cpu-initialization-period` flag, and its default is 5 minutes.

The `currentMetricValue / desiredMetricValue` base scale ratio is then calculated using the remaining pods not set aside or discarded from above.

If there were any missing metrics, we recompute the average more conservatively, assuming those pods were consuming 100% of the desired value in case of a scale down, and 0% in case of a scale up. This dampens the magnitude of any potential scale.

Furthermore, if any not-yet-ready pods were present, and we would have scaled up without factoring in missing metrics or not-yet-ready pods, we conservatively assume the not-yet-ready pods are consuming 0% of the desired metric, further dampening the magnitude of a scale up.

After factoring in the not-yet-ready pods and missing metrics, we recalculate the usage ratio. If the new ratio reverses the scale direction, or is within the tolerance, we skip scaling. Otherwise, we use the new ratio to scale.

Note that the *original* value for the average utilization is reported back via the HorizontalPodAutoscaler status, without factoring in the not-yet-ready pods or missing metrics, even when the new usage ratio is used.

If multiple metrics are specified in a HorizontalPodAutoscaler, this calculation is done for each metric, and then the largest of the desired replica counts is chosen. If any of these metrics cannot be converted into a desired replica count (e.g. due to an error fetching the metrics from the metrics APIs) and a scale down is suggested by the metrics which can be

fetched, scaling is skipped. This means that the HPA is still capable of scaling up if one or more metrics give a `desiredReplicas` greater than the current value.

Finally, just before HPA scales the target, the scale recommendation is recorded. The controller considers all recommendations within a configurable window choosing the highest recommendation from within that window. This value can be configured using the `--horizontal-pod-autoscaler-downscale-stabilization` flag, which defaults to 5 minutes. This means that scaledowns will occur gradually, smoothing out the impact of rapidly fluctuating metric values.

## API Object

The Horizontal Pod Autoscaler is an API resource in the Kubernetes `autoscaling` API group. The current stable version, which only includes support for CPU autoscaling, can be found in the `autoscaling/v1` API version.

The beta version, which includes support for scaling on memory and custom metrics, can be found in `autoscaling/v2beta2`. The new fields introduced in `autoscaling/v2beta2` are preserved as annotations when working with `autoscaling/v1`.

More details about the API object can be found at [HorizontalPodAutoscaler Object](#).

## Support for Horizontal Pod Autoscaler in kubectl

Horizontal Pod Autoscaler, like every API resource, is supported in a standard way by `kubectl`. We can create a new autoscaler using `kubectl create` command. We can list autoscalers by `kubectl get hpa` and get detailed description by `kubectl describe hpa`. Finally, we can delete an autoscaler using `kubectl delete hpa`.

In addition, there is a special `kubectl autoscale` command for easy creation of a Horizontal Pod Autoscaler. For instance, executing `kubectl autoscale rs foo --min=2 --max=5 --cpu-percent=80` will create an autoscaler for replication set `foo`, with target CPU utilization set to 80% and the number of replicas between 2 and 5. The detailed documentation of `kubectl autoscale` can be found [here](#).

## Autoscaling during rolling update

Currently in Kubernetes, it is possible to perform a [rolling update](#) by managing replication controllers directly, or by using the deployment object, which manages the underlying replica sets for you. Horizontal Pod Autoscaler only supports the latter approach: the Horizontal Pod Autoscaler is bound to the deployment object, it sets the size for the deployment object,

and the deployment is responsible for setting sizes of underlying replica sets.

Horizontal Pod Autoscaler does not work with rolling update using direct manipulation of replication controllers, i.e. you cannot bind a Horizontal Pod Autoscaler to a replication controller and do rolling update (e.g. using `kubectl rolling-update`). The reason this doesn't work is that when rolling update creates a new replication controller, the Horizontal Pod Autoscaler will not be bound to the new replication controller.

## Support for cooldown/delay

When managing the scale of a group of replicas using the Horizontal Pod Autoscaler, it is possible that the number of replicas keeps fluctuating frequently due to the dynamic nature of the metrics evaluated. This is sometimes referred to as *thrashing*.

Starting from v1.6, a cluster operator can mitigate this problem by tuning the global HPA settings exposed as flags for the `kube-controller-manager` component:

Starting from v1.12, a new algorithmic update removes the need for the upscale delay.

- `--horizontal-pod-autoscaler-downscale-stabilization`: The value for this option is a duration that specifies how long the autoscaler has to wait before another downscale operation can be performed after the current one has completed. The default value is 5 minutes (5m0s).

**Note:** When tuning these parameter values, a cluster operator should be aware of the possible consequences. If the delay (cooldown) value is set too long, there could be complaints that the Horizontal Pod Autoscaler is not responsive to workload changes. However, if the delay value is set too short, the scale of the replicas set may keep thrashing as usual.

## Support for multiple metrics

Kubernetes 1.6 adds support for scaling based on multiple metrics. You can use the `autoscaling/v2beta2` API version to specify multiple metrics for the Horizontal Pod Autoscaler to scale on. Then, the Horizontal Pod Autoscaler controller will evaluate each metric, and propose a new scale based on that metric. The largest of the proposed scales will be used as the new scale.

## Support for custom metrics

**Note:** Kubernetes 1.2 added alpha support for scaling based on application-specific metrics using special annotations. Support for these annotations was removed in Kubernetes 1.6 in favor of the new autoscaling API. While the old method for collecting custom

metrics is still available, these metrics will not be available for use by the Horizontal Pod Autoscaler, and the former annotations for specifying which custom metrics to scale on are no longer honored by the Horizontal Pod Autoscaler controller.

Kubernetes 1.6 adds support for making use of custom metrics in the Horizontal Pod Autoscaler. You can add custom metrics for the Horizontal Pod Autoscaler to use in the `autoscaling/v2beta2` API. Kubernetes then queries the new custom metrics API to fetch the values of the appropriate custom metrics.

See [Support for metrics APIs](#) for the requirements.

## Support for metrics APIs

By default, the `HorizontalPodAutoscaler` controller retrieves metrics from a series of APIs. In order for it to access these APIs, cluster administrators must ensure that:

- The [API aggregation layer](#) is enabled.
- The corresponding APIs are registered:
  - For resource metrics, this is the `metrics.k8s.io` API, generally provided by [metrics-server](#). It can be launched as a cluster addon.
  - For custom metrics, this is the `custom.metrics.k8s.io` API. It's provided by "adapter" API servers provided by metrics solution vendors. Check with your metrics pipeline, or the [list of known solutions](#). If you would like to write your own, check out the [boilerplate](#) to get started.
  - For external metrics, this is the `external.metrics.k8s.io` API. It may be provided by the custom metrics adapters provided above.
- The `--horizontal-pod-autoscaler-use-rest-clients` is true or unset. Setting this to false switches to Heapster-based autoscaling, which is deprecated.

For more information on these different metrics paths and how they differ please see the relevant design proposals for [the HPA V2](#), [custom.metrics.k8s.io](#) and [external.metrics.k8s.io](#).

For examples of how to use them see [the walkthrough for using custom metrics](#) and [the walkthrough for using external metrics](#).

## What's next

- Design documentation: [Horizontal Pod Autoscaling](#).
- `kubectl autoscale` command: [kubectl autoscale](#).
- Usage example of [Horizontal Pod Autoscaler](#).

# Feedback

Was this page helpful?

Yes No

Thanks for the feedback. If you have a specific, answerable question about how to use Kubernetes, ask it on [Stack Overflow](#). Open an issue in the GitHub repo if you want to [report a problem](#) or [suggest an improvement](#).

---

[Create an Issue](#) [Edit This Page](#)

Page last modified on February 03, 2020 at 5:21 PM PST by [Revert](#)  
["Configurable Scaling for the HPA \(#18157\)" \(#18963\)](#) ([Page History](#))

[Edit This Page](#)

# Horizontal Pod Autoscaler Walkthrough

Horizontal Pod Autoscaler automatically scales the number of pods in a replication controller, deployment, replica set or stateful set based on observed CPU utilization (or, with beta support, on some other, application-provided metrics).

This document walks you through an example of enabling Horizontal Pod Autoscaler for the php-apache server. For more information on how Horizontal Pod Autoscaler behaves, see the [Horizontal Pod Autoscaler user guide](#).

- [Before you begin](#)
- [Run & expose php-apache server](#)
- [Create Horizontal Pod Autoscaler](#)
- [Increase load](#)
- [Stop load](#)
- [Autoscaling on multiple metrics and custom metrics](#)
- [Appendix: Horizontal Pod Autoscaler Status Conditions](#)
- [Appendix: Quantities](#)
- [Appendix: Other possible scenarios](#)

## Before you begin

This example requires a running Kubernetes cluster and kubectl, version 1.2 or later. [metrics-server](#) monitoring needs to be deployed in the cluster to provide metrics via the resource metrics API, as Horizontal Pod Autoscaler uses this API to collect metrics. The instructions for deploying this are on the GitHub repository of [metrics-server](#), if you followed [getting started on GCE guide](#), metrics-server monitoring will be turned-on by default.

To specify multiple resource metrics for a Horizontal Pod Autoscaler, you must have a Kubernetes cluster and kubectl at version 1.6 or later. Furthermore, in order to make use of custom metrics, your cluster must be able to communicate with the API server providing the custom metrics API. Finally, to use metrics not related to any Kubernetes object you must have a Kubernetes cluster at version 1.10 or later, and you must be able to communicate with the API server that provides the external metrics API. See the [Horizontal Pod Autoscaler user guide](#) for more details.

## Run & expose php-apache server

To demonstrate Horizontal Pod Autoscaler we will use a custom docker image based on the php-apache image. The Dockerfile has the following content:

```
FROM php:5-apache
ADD index.php /var/www/html/index.php
RUN chmod a+rx index.php
```

It defines an index.php page which performs some CPU intensive computations:

```
<?php
$x = 0.0001;
for ($i = 0; $i <= 1000000; $i++) {
    $x += sqrt($x);
}
echo "OK!";
?>
```

First, we will start a deployment running the image and expose it as a service using the following configuration:

### [application/php-apache.yaml](#)

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: php-apache
spec:
  selector:
    matchLabels:
      run: php-apache
  replicas: 1
  template:
    metadata:
      labels:
        run: php-apache
    spec:
      containers:
        - name: php-apache
          image: k8s.gcr.io/hpa-example
          ports:
            - containerPort: 80
          resources:
            limits:
              cpu: 500m
            requests:
              cpu: 200m
```

---

```
apiVersion: v1
kind: Service
metadata:
  name: php-apache
  labels:
    run: php-apache
spec:
  ports:
    - port: 80
  selector:
    run: php-apache
```

Run the following command:

```
kubectl apply -f https://k8s.io/examples/application/php-
apache.yaml
```

```
deployment.apps/php-apache created
service/php-apache created
```

# Create Horizontal Pod Autoscaler

Now that the server is running, we will create the autoscaler using [kubectl autoscale](#). The following command will create a Horizontal Pod Autoscaler that maintains between 1 and 10 replicas of the Pods controlled by the php-apache deployment we created in the first step of these instructions.

Roughly speaking, HPA will increase and decrease the number of replicas (via the deployment) to maintain an average CPU utilization across all Pods of 50% (since each pod requests 200 milli-cores by `kubectl run`), this means average CPU usage of 100 milli-cores). See [here](#) for more details on the algorithm.

```
kubectl autoscale deployment php-apache --cpu-percent=50 --min=1  
--max=10
```

```
horizontalpodautoscaler.autoscaling/php-apache autoscaled
```

We may check the current status of autoscaler by running:

```
kubectl get hpa
```

NAME	REFERENCE	TARGET	MINPODS
MAXPODS	REPLICAS	AGE	
php-apache	Deployment/php-apache/scale	0% / 50%	1
10	1	18s	

Please note that the current CPU consumption is 0% as we are not sending any requests to the server (the TARGET column shows the average across all the pods controlled by the corresponding deployment).

## Increase load

Now, we will see how the autoscaler reacts to increased load. We will start a container, and send an infinite loop of queries to the php-apache service (please run it in a different terminal):

```
kubectl run --generator=run-pod/v1 -it --rm load-generator --  
image=busybox /bin/sh
```

Hit enter **for command** prompt

```
while true; do wget -q -O- http://php-  
apache.default.svc.cluster.local; done
```

Within a minute or so, we should see the higher CPU load by executing:

```
kubectl get hpa
```

NAME	REFERENCE	TARGET	MINPODS
MAXPODS	REPLICAS	AGE	
php-apache	Deployment/php-apache/scale	305% / 50%	1

10	1	3m
----	---	----

Here, CPU consumption has increased to 305% of the request. As a result, the deployment was resized to 7 replicas:

```
kubectl get deployment php-apache
```

NAME	READY	UP-TO-DATE	AVAILABLE	AGE
php-apache	7/7	7	7	19m

**Note:** It may take a few minutes to stabilize the number of replicas. Since the amount of load is not controlled in any way it may happen that the final number of replicas will differ from this example.

## Stop load

We will finish our example by stopping the user load.

In the terminal where we created the container with busybox image, terminate the load generation by typing <Ctrl> + C.

Then we will verify the result state (after a minute or so):

```
kubectl get hpa
```

NAME	REFERENCE	TARGET
MINPODS	MAXPODS	REPLICAS AGE
php-apache	Deployment/php-apache/scale	0% / 50%
1	10	1 11m

```
kubectl get deployment php-apache
```

NAME	READY	UP-TO-DATE	AVAILABLE	AGE
php-apache	1/1	1	1	27m

Here CPU utilization dropped to 0, and so HPA autoscaled the number of replicas back down to 1.

**Note:** Autoscaling the replicas may take a few minutes.

## Autoscaling on multiple metrics and custom metrics

You can introduce additional metrics to use when autoscaling the php-apache Deployment by making use of the `autoscaling/v2beta2` API version.

First, get the YAML of your HorizontalPodAutoscaler in the `autoscaling/v2beta2` form:

```
kubectl get hpa.v2beta2.autoscaling -o yaml > /tmp/hpa-v2.yaml
```

Open the `/tmp/hpa-v2.yaml` file in an editor, and you should see YAML which looks like this:

```
apiVersion: autoscaling/v2beta2
kind: HorizontalPodAutoscaler
metadata:
  name: php-apache
  namespace: default
spec:
  scaleTargetRef:
    apiVersion: apps/v1
    kind: Deployment
    name: php-apache
  minReplicas: 1
  maxReplicas: 10
  metrics:
    - type: Resource
      resource:
        name: cpu
        target:
          type: Utilization
          averageUtilization: 50
status:
  observedGeneration: 1
  lastScaleTime: <some-time>
  currentReplicas: 1
  desiredReplicas: 1
  currentMetrics:
    - type: Resource
      resource:
        name: cpu
        current:
          averageUtilization: 0
          averageValue: 0
```

Notice that the `targetCPUUtilizationPercentage` field has been replaced with an array called `metrics`. The CPU utilization metric is a *resource metric*, since it is represented as a percentage of a resource specified on pod containers. Notice that you can specify other resource metrics besides CPU. By default, the only other supported resource metric is memory. These resources do not change names from cluster to cluster, and should always be available, as long as the `metrics.k8s.io` API is available.

You can also specify resource metrics in terms of direct values, instead of as percentages of the requested value, by using a `target` type of `AverageValue` instead of `AverageUtilization`, and setting the corresponding `target.averageValue` field instead of the `target.averageUtilization`.

There are two other types of metrics, both of which are considered *custom metrics*: pod metrics and object metrics. These metrics may have names which are cluster specific, and require a more advanced cluster monitoring setup.

The first of these alternative metric types is *pod metrics*. These metrics describe pods, and are averaged together across pods and compared with a target value to determine the replica count. They work much like resource metrics, except that they *only* support a `target` type of `AverageValue`.

Pod metrics are specified using a metric block like this:

```
type: Pods
pods:
  metric:
    name: packets-per-second
  target:
    type: AverageValue
    averageValue: 1k
```

The second alternative metric type is *object metrics*. These metrics describe a different object in the same namespace, instead of describing pods. The metrics are not necessarily fetched from the object; they only describe it. Object metrics support `target` types of both `Value` and `AverageValue`. With `Value`, the target is compared directly to the returned metric from the API. With `AverageValue`, the value returned from the custom metrics API is divided by the number of pods before being compared to the target. The following example is the YAML representation of the `requests-per-second` metric.

```
type: Object
object:
  metric:
    name: requests-per-second
  describedObject:
    apiVersion: networking.k8s.io/v1beta1
    kind: Ingress
    name: main-route
  target:
    type: Value
    value: 2k
```

If you provide multiple such metric blocks, the `HorizontalPodAutoscaler` will consider each metric in turn. The `HorizontalPodAutoscaler` will calculate proposed replica counts for each metric, and then choose the one with the highest replica count.

For example, if you had your monitoring system collecting metrics about network traffic, you could update the definition above using `kubectl edit` to look like this:

```
apiVersion: autoscaling/v2beta2
kind: HorizontalPodAutoscaler
metadata:
  name: php-apache
  namespace: default
spec:
  scaleTargetRef:
```

```
apiVersion: apps/v1
kind: Deployment
name: php-apache
minReplicas: 1
maxReplicas: 10
metrics:
- type: Resource
  resource:
    name: cpu
    target:
      type: Utilization
      averageUtilization: 50
- type: Pods
  pods:
    metric:
      name: packets-per-second
      target:
        type: AverageValue
        averageValue: 1k
- type: Object
  object:
    metric:
      name: requests-per-second
    describedObject:
      apiVersion: networking.k8s.io/v1beta1
      kind: Ingress
      name: main-route
    target:
      type: Value
      value: 10k
status:
  observedGeneration: 1
  lastScaleTime: <some-time>
  currentReplicas: 1
  desiredReplicas: 1
  currentMetrics:
  - type: Resource
    resource:
      name: cpu
    current:
      averageUtilization: 0
      averageValue: 0
  - type: Object
    object:
      metric:
        name: requests-per-second
    describedObject:
      apiVersion: networking.k8s.io/v1beta1
      kind: Ingress
      name: main-route
    current:
      value: 10k
```

Then, your HorizontalPodAutoscaler would attempt to ensure that each pod was consuming roughly 50% of its requested CPU, serving 1000 packets per second, and that all pods behind the main-route Ingress were serving a total of 10000 requests per second.

## Autoscaling on more specific metrics

Many metrics pipelines allow you to describe metrics either by name or by a set of additional descriptors called *labels*. For all non-resource metric types (pod, object, and external, described below), you can specify an additional label selector which is passed to your metric pipeline. For instance, if you collect a metric `http_requests` with the `verb` label, you can specify the following metric block to scale only on GET requests:

```
type: Object
object:
  metric:
    name: `http_requests`
    selector: `verb=GET`
```

This selector uses the same syntax as the full Kubernetes label selectors. The monitoring pipeline determines how to collapse multiple series into a single value, if the name and selector match multiple series. The selector is additive, and cannot select metrics that describe objects that are **not** the target object (the target pods in the case of the `Pods` type, and the described object in the case of the `Object` type).

## Autoscaling on metrics not related to Kubernetes objects

Applications running on Kubernetes may need to autoscale based on metrics that don't have an obvious relationship to any object in the Kubernetes cluster, such as metrics describing a hosted service with no direct correlation to Kubernetes namespaces. In Kubernetes 1.10 and later, you can address this use case with *external metrics*.

Using external metrics requires knowledge of your monitoring system; the setup is similar to that required when using custom metrics. External metrics allow you to autoscale your cluster based on any metric available in your monitoring system. Just provide a `metric` block with a name and `select` or, as above, and use the `External` metric type instead of `Object`. If multiple time series are matched by the `metricSelector`, the sum of their values is used by the HorizontalPodAutoscaler. External metrics support both the `Value` and `AverageValue` target types, which function exactly the same as when you use the `Object` type.

For example if your application processes tasks from a hosted queue service, you could add the following section to your HorizontalPodAutoscaler manifest to specify that you need one worker per 30 outstanding tasks.

```
- type: External
  external:
    metric:
```

```

  name: queue_messages_ready
  selector: "queue=worker_tasks"
  target:
    type: AverageValue
    averageValue: 30

```

When possible, it's preferable to use the custom metric target types instead of external metrics, since it's easier for cluster administrators to secure the custom metrics API. The external metrics API potentially allows access to any metric, so cluster administrators should take care when exposing it.

## Appendix: Horizontal Pod Autoscaler Status Conditions

When using the `autoscaling/v2beta2` form of the `HorizontalPodAutoscaler`, you will be able to see *status conditions* set by Kubernetes on the `HorizontalPodAutoscaler`. These status conditions indicate whether or not the `HorizontalPodAutoscaler` is able to scale, and whether or not it is currently restricted in any way.

The conditions appear in the `status.conditions` field. To see the conditions affecting a `HorizontalPodAutoscaler`, we can use `kubectl describe hpa`:

```
kubectl describe hpa cm-test
```

```

Name:                      cm-test
Namespace:                 prom
Labels:                    <none>
Annotations:               <none>
CreationTimestamp:         Fri, 16 Jun 2017 18:09:22 +0000
Reference:                 ReplicationController/cm-test
Metrics:                   ( current / target )
  "http_requests" on pods: 66m / 500m
Min replicas:              1
Max replicas:              4
ReplicationController pods: 1 current / 1 desired
Conditions:
  Type        Status  Reason           Message
  ----        ----   ----             -----
  AbleToScale True    ReadyForNewScale the last
scale time was sufficiently old as to warrant a new scale
  ScalingActive True    ValidMetricFound the HPA
was able to successfully calculate a replica count from pods
metric http_requests
  ScalingLimited False   DesiredWithinRange the
desired replica count is within the acceptable range
Events:

```

For this `HorizontalPodAutoscaler`, we can see several conditions in a healthy state. The first, `AbleToScale`, indicates whether or not the HPA is able to fetch and update scales, as well as whether or not any backoff-related

conditions would prevent scaling. The second, `ScalingActive`, indicates whether or not the HPA is enabled (i.e. the replica count of the target is not zero) and is able to calculate desired scales. When it is `False`, it generally indicates problems with fetching metrics. Finally, the last condition, `ScalingLimited`, indicates that the desired scale was capped by the maximum or minimum of the HorizontalPodAutoscaler. This is an indication that you may wish to raise or lower the minimum or maximum replica count constraints on your HorizontalPodAutoscaler.

## Appendix: Quantities

All metrics in the HorizontalPodAutoscaler and metrics APIs are specified using a special whole-number notation known in Kubernetes as a *quantity*. For example, the quantity `10500m` would be written as `10.5` in decimal notation. The metrics APIs will return whole numbers without a suffix when possible, and will generally return quantities in milli-units otherwise. This means you might see your metric value fluctuate between `1` and `1500m`, or `1` and `1.5` when written in decimal notation. See the [glossary entry on quantities](#) for more information.

## Appendix: Other possible scenarios

### Creating the autoscaler declaratively

Instead of using `kubectl autoscale` command to create a HorizontalPodAutoscaler imperatively we can use the following file to create it declaratively:

```
application/hpa/php-apache.yaml

apiVersion: autoscaling/v1
kind: HorizontalPodAutoscaler
metadata:
  name: php-apache
  namespace: default
spec:
  scaleTargetRef:
    apiVersion: apps/v1
    kind: Deployment
    name: php-apache
  minReplicas: 1
  maxReplicas: 10
  targetCPUUtilizationPercentage: 50
```

We will create the autoscaler by executing the following command:

```
kubectl create -f https://k8s.io/examples/application/hpa/php-
apache.yaml
```

```
horizontalpodautoscaler.autoscaling/php-apache created
```

## Feedback

Was this page helpful?

Yes No

Thanks for the feedback. If you have a specific, answerable question about how to use Kubernetes, ask it on [Stack Overflow](#). Open an issue in the GitHub repo if you want to [report a problem](#) or [suggest an improvement](#).

---

[Create an Issue](#) [Edit This Page](#)

Page last modified on February 07, 2020 at 8:43 PM PST by [Fix of pull request #18960 \(#18974\)](#) ([Page History](#))

[Edit This Page](#)

# Specifying a Disruption Budget for your Application

This page shows how to limit the number of concurrent disruptions that your application experiences, allowing for higher availability while permitting the cluster administrator to manage the clusters nodes.

- [Before you begin](#)
- [Protecting an Application with a PodDisruptionBudget](#)
- [Identify an Application to Protect](#)
- [Think about how your application reacts to disruptions](#)
- [Specifying a PodDisruptionBudget](#)
- [Create the PDB object](#)
- [Check the status of the PDB](#)
- [Arbitrary Controllers and Selectors](#)

## Before you begin

- You are the owner of an application running on a Kubernetes cluster that requires high availability.
- You should know how to deploy [Replicated Stateless Applications](#) and/or [Replicated Stateful Applications](#).
- You should have read about [Pod Disruptions](#).
- You should confirm with your cluster owner or service provider that they respect Pod Disruption Budgets.

## Protecting an Application with a PodDisruptionBudget

1. Identify what application you want to protect with a PodDisruptionBudget (PDB).
2. Think about how your application reacts to disruptions.
3. Create a PDB definition as a YAML file.
4. Create the PDB object from the YAML file.

## Identify an Application to Protect

The most common use case when you want to protect an application specified by one of the built-in Kubernetes controllers:

- Deployment
- ReplicationController
- ReplicaSet
- StatefulSet

In this case, make a note of the controller's `.spec.selector`; the same selector goes into the PDBs `.spec.selector`.

From version 1.15 PDBs support custom controllers where the [scale subresource](#) is enabled.

You can also use PDBs with pods which are not controlled by one of the above controllers, or arbitrary groups of pods, but there are some restrictions, described in [Arbitrary Controllers and Selectors](#).

## Think about how your application reacts to disruptions

Decide how many instances can be down at the same time for a short period due to a voluntary disruption.

- Stateless frontends:
  - Concern: don't reduce serving capacity by more than 10%.
  - Solution: use PDB with `minAvailable 90%` for example.
- Single-instance Stateful Application:
  - Concern: do not terminate this application without talking to me.
  - Possible Solution 1: Do not use a PDB and tolerate occasional downtime.
  - Possible Solution 2: Set PDB with `maxUnavailable=0`. Have an understanding (outside of Kubernetes) that the cluster operator needs to consult you before termination. When the cluster operator contacts you, prepare for downtime, and then delete the PDB to indicate readiness for disruption. Recreate afterwards.
- Multiple-instance Stateful application such as Consul, ZooKeeper, or etcd:
  - Concern: Do not reduce number of instances below quorum, otherwise writes fail.
  - Possible Solution 1: set `maxUnavailable` to 1 (works with varying scale of application).
  - Possible Solution 2: set `minAvailable` to quorum-size (e.g. 3 when scale is 5). (Allows more disruptions at once).
- Restartable Batch Job:
  - Concern: Job needs to complete in case of voluntary disruption.
  - Possible solution: Do not create a PDB. The Job controller will create a replacement pod.

## Rounding logic when specifying percentages

Values for `minAvailable` or `maxUnavailable` can be expressed as integers or as a percentage.

- When you specify an integer, it represents a number of Pods. For instance, if you set `minAvailable` to 10, then 10 Pods must always be available, even during a disruption.
- When you specify a percentage by setting the value to a string representation of a percentage (eg. "50%"), it represents a percentage

of total Pods. For instance, if you set `maxUnavailable` to "50%", then only 50% of the Pods can be unavailable during a disruption.

When you specify the value as a percentage, it may not map to an exact number of Pods. For example, if you have 7 Pods and you set `minAvailable` to "50%", it's not immediately obvious whether that means 3 Pods or 4 Pods must be available. Kubernetes rounds up to the nearest integer, so in this case, 4 Pods must be available. You can examine the [code](#) that controls this behavior.

## Specifying a PodDisruptionBudget

A `PodDisruptionBudget` has three fields:

- A label selector `.spec.selector` to specify the set of pods to which it applies. This field is required.
- `.spec.minAvailable` which is a description of the number of pods from that set that must still be available after the eviction, even in the absence of the evicted pod. `minAvailable` can be either an absolute number or a percentage.
- `.spec.maxUnavailable` (available in Kubernetes 1.7 and higher) which is a description of the number of pods from that set that can be unavailable after the eviction. It can be either an absolute number or a percentage.

**Note:** For versions 1.8 and earlier: When creating a `PodDisruptionBudget` object using the `kubectl` command line tool, the `minAvailable` field has a default value of 1 if neither `minAvailable` nor `maxUnavailable` is specified.

You can specify only one of `maxUnavailable` and `minAvailable` in a single `PodDisruptionBudget`. `maxUnavailable` can only be used to control the eviction of pods that have an associated controller managing them. In the examples below, "desired replicas" is the scale of the controller managing the pods being selected by the `PodDisruptionBudget`.

Example 1: With a `minAvailable` of 5, evictions are allowed as long as they leave behind 5 or more healthy pods among those selected by the `PodDisruptionBudget`'s selector.

Example 2: With a `minAvailable` of 30%, evictions are allowed as long as at least 30% of the number of desired replicas are healthy.

Example 3: With a `maxUnavailable` of 5, evictions are allowed as long as there are at most 5 unhealthy replicas among the total number of desired replicas.

Example 4: With a `maxUnavailable` of 30%, evictions are allowed as long as no more than 30% of the desired replicas are unhealthy.

In typical usage, a single budget would be used for a collection of pods managed by a controller—for example, the pods in a single ReplicaSet or StatefulSet.

**Note:** A disruption budget does not truly guarantee that the specified number/percentage of pods will always be up. For example, a node that hosts a pod from the collection may fail when the collection is at the minimum size specified in the budget, thus bringing the number of available pods from the collection below the specified size. The budget can only protect against voluntary evictions, not all causes of unavailability.

If you set `maxUnavailable` to 0% or 0, or you set `minAvailable` to 100% or the number of replicas, you are requiring zero voluntary evictions. When you set zero voluntary evictions for a workload object such as ReplicaSet, then you cannot successfully drain a Node running one of those Pods. If you try to drain a Node where an unevictable Pod is running, the drain never completes. This is permitted as per the semantics of `PodDisruptionBudget`.

You can find examples of pod disruption budgets defined below. They match pods with the label `app: zookeeper`.

Example PDB Using `minAvailable`:

#### [`policy/zookeeper-pod-disruption-budget-minavailable.yaml`](#)

```
apiVersion: policy/v1beta1
kind: PodDisruptionBudget
metadata:
  name: zk-pdb
spec:
  minAvailable: 2
  selector:
    matchLabels:
      app: zookeeper
```

Example PDB Using `maxUnavailable` (Kubernetes 1.7 or higher):

#### [`policy/zookeeper-pod-disruption-budget-maxunavailable.yaml`](#)

```
apiVersion: policy/v1beta1
kind: PodDisruptionBudget
metadata:
  name: zk-pdb
spec:
  maxUnavailable: 1
  selector:
    matchLabels:
      app: zookeeper
```

For example, if the above zk-pdb object selects the pods of a StatefulSet of size 3, both specifications have the exact same meaning. The use of `maxUnavailable` is recommended as it automatically responds to changes in the number of replicas of the corresponding controller.

## Create the PDB object

You can create or update the PDB object with a command like `kubectl apply -f mypdb.yaml`.

## Check the status of the PDB

Use `kubectl` to check that your PDB is created.

Assuming you don't actually have pods matching `app: zookeeper` in your namespace, then you'll see something like this:

```
kubectl get poddisruptionbudgets
```

NAME	MIN-AVAILABLE	ALLOWED-DISRUPTIONS	AGE
zk-pdb	2	0	7s

If there are matching pods (say, 3), then you would see something like this:

```
kubectl get poddisruptionbudgets
```

NAME	MIN-AVAILABLE	ALLOWED-DISRUPTIONS	AGE
zk-pdb	2	1	7s

The non-zero value for `ALLOWED-DISRUPTIONS` means that the disruption controller has seen the pods, counted the matching pods, and updated the status of the PDB.

You can get more information about the status of a PDB with this command:

```
kubectl get poddisruptionbudgets zk-pdb -o yaml
```

```
apiVersion: policy/v1beta1
kind: PodDisruptionBudget
metadata:
  creationTimestamp: 2017-08-28T02:38:26Z
  generation: 1
  name: zk-pdb
status:
  currentHealthy: 3
  desiredHealthy: 3
  disruptedPods: null
  disruptionsAllowed: 1
  expectedPods: 3
  observedGeneration: 1
```

# Arbitrary Controllers and Selectors

You can skip this section if you only use PDBs with the built-in application controllers (Deployment, ReplicationController, ReplicaSet, and StatefulSet), with the PDB selector matching the controller's selector.

You can use a PDB with pods controlled by another type of controller, by an "operator", or bare pods, but with these restrictions:

- only `.spec.minAvailable` can be used, not `.spec.maxUnavailable`.
- only an integer value can be used with `.spec.minAvailable`, not a percentage.

You can use a selector which selects a subset or superset of the pods belonging to a built-in controller. However, when there are multiple PDBs in a namespace, you must be careful not to create PDBs whose selectors overlap.

## Feedback

Was this page helpful?

Yes No

Thanks for the feedback. If you have a specific, answerable question about how to use Kubernetes, ask it on [Stack Overflow](#). Open an issue in the GitHub repo if you want to [report a problem](#) or [suggest an improvement](#).

---

[Create an Issue](#) [Edit This Page](#)

Page last modified on December 06, 2019 at 4:48 PM PST by [Document correction for PDB \(#17573\)](#) ([Page History](#))

[Edit This Page](#)

# Running Automated Tasks with a CronJob

You can use a [CronJob](#)[Manages a Job that runs on a periodic schedule.](#) to run [Jobs](#)[A finite or batch task that runs to completion.](#) on a time-based schedule. These automated jobs run like [Cron](#) tasks on a Linux or UNIX system.

Cron jobs are useful for creating periodic and recurring tasks, like running backups or sending emails. Cron jobs can also schedule individual tasks for a specific time, such as if you want to schedule a job for a low activity period.

Cron jobs have limitations and idiosyncrasies. For example, in certain circumstances, a single cron job can create multiple jobs. Therefore, jobs should be idempotent.

For more limitations, see [CronJobs](#).

- [Before you begin](#)
- [Creating a Cron Job](#)
- [Deleting a Cron Job](#)
- [Writing a Cron Job Spec](#)

## Before you begin

- You need to have a Kubernetes cluster, and the kubectl command-line tool must be configured to communicate with your cluster. 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](#)

## Creating a Cron Job

Cron jobs require a config file. This example cron job config `.spec` file prints the current time and a hello message every minute:

## [application/job/cronjob.yaml](#)

```
apiVersion: batch/v1beta1
kind: CronJob
metadata:
  name: hello
spec:
  schedule: "*/1 * * * *"
  jobTemplate:
    spec:
      template:
        spec:
          containers:
            - name: hello
              image: busybox
              args:
                - /bin/sh
                - -c
                - date; echo Hello from the Kubernetes cluster
  restartPolicy: OnFailure
```

Run the example CronJob by using this command:

```
kubectl create -f https://k8s.io/examples/application/job/cronjob.yaml
```

The output is similar to this:

```
cronjob.batch/hello created
```

Alternatively, you can use `kubectl run` to create a cron job without writing a full config:

```
kubectl run hello --schedule="*/1 * * * *" --restart=OnFailure --image=busybox -- /bin/sh -c "date; echo Hello from the Kubernetes cluster"
```

After creating the cron job, get its status using this command:

```
kubectl get cronjob hello
```

The output is similar to this:

NAME	SCHEDULE	SUSPEND	ACTIVE	LAST SCHEDULE	AGE
hello	*/1 * * * *	False	0	<none>	10s

As you can see from the results of the command, the cron job has not scheduled or run any jobs yet. Watch for the job to be created in around one minute:

```
kubectl get jobs --watch
```

The output is similar to this:

NAME	COMPLETIONS	DURATION	AGE
hello-4111706356	0/1		0s
hello-4111706356	0/1	0s	0s
hello-4111706356	1/1	5s	5s

Now you've seen one running job scheduled by the "hello" cron job. You can stop watching the job and view the cron job again to see that it scheduled the job:

```
kubectl get cronjob hello
```

The output is similar to this:

NAME	SCHEDULE	SUSPEND	ACTIVE	LAST SCHEDULE	AGE
hello	*/1 * * * *	False	0	50s	75s

You should see that the cron job hello successfully scheduled a job at the time specified in LAST SCHEDULE. There are currently 0 active jobs, meaning that the job has completed or failed.

Now, find the pods that the last scheduled job created and view the standard output of one of the pods.

**Note:** The job name and pod name are different.

```
# Replace "hello-4111706356" with the job name in your system
pods=$(kubectl get pods --selector=job-name=hello-4111706356 --
output=jsonpath={.items[*].metadata.name})
```

Show pod log:

```
kubectl logs $pods
```

The output is similar to this:

```
Fri Feb 22 11:02:09 UTC 2019
Hello from the Kubernetes cluster
```

## Deleting a Cron Job

When you don't need a cron job any more, delete it with `kubectl delete cronjob <cronjob name>`:

```
kubectl delete cronjob hello
```

Deleting the cron job removes all the jobs and pods it created and stops it from creating additional jobs. You can read more about removing jobs in [garbage collection](#).

# Writing a Cron Job Spec

As with all other Kubernetes configs, a cron job needs `apiVersion`, `kind`, and `metadata` fields. For general information about working with config files, see [deploying applications](#), and [using kubectl to manage resources](#) documents.

A cron job config also needs a [.spec section](#).

**Note:** All modifications to a cron job, especially its `.spec`, are applied only to the following runs.

## Schedule

The `.spec.schedule` is a required field of the `.spec`. It takes a [Cron](#) format string, such as `0 * * * *` or `@hourly`, as schedule time of its jobs to be created and executed.

The format also includes extended vixie cron step values. As explained in the [FreeBSD manual](#):

Step values can be used in conjunction with ranges. Following a range with `/<number>` specifies skips of the number's value through the range. For example, `0-23/2` can be used in the hours field to specify command execution every other hour (the alternative in the V7 standard is `0,2,4,6,8,10,12,14,16,18,20,22`). Steps are also permitted after an asterisk, so if you want to say "every two hours", just use `*/2`.

**Note:** A question mark (?) in the schedule has the same meaning as an asterisk \*, that is, it stands for any of available value for a given field.

## Job Template

The `.spec.jobTemplate` is the template for the job, and it is required. It has exactly the same schema as a [Job](#), except that it is nested and does not have an `apiVersion` or `kind`. For information about writing a job `.spec`, see [Writing a Job Spec](#).

## Starting Deadline

The `.spec.startingDeadlineSeconds` field is optional. It stands for the deadline in seconds for starting the job if it misses its scheduled time for any reason. After the deadline, the cron job does not start the job. Jobs that do not meet their deadline in this way count as failed jobs. If this field is not specified, the jobs have no deadline.

The CronJob controller counts how many missed schedules happen for a cron job. If there are more than 100 missed schedules, the cron job is no longer scheduled. When `.spec.startingDeadlineSeconds` is not set, the

CronJob controller counts missed schedules from `status.lastScheduleTime` until now.

For example, one cron job is supposed to run every minute, the `status.lastScheduleTime` of the cronjob is 5:00am, but now it's 7:00am. That means 120 schedules were missed, so the cron job is no longer scheduled.

If the `.spec.startingDeadlineSeconds` field is set (not null), the CronJob controller counts how many missed jobs occurred from the value of `.spec.startingDeadlineSeconds` until now.

For example, if it is set to 200, it counts how many missed schedules occurred in the last 200 seconds. In that case, if there were more than 100 missed schedules in the last 200 seconds, the cron job is no longer scheduled.

## Concurrency Policy

The `.spec.concurrencyPolicy` field is also optional. It specifies how to treat concurrent executions of a job that is created by this cron job. The spec may specify only one of the following concurrency policies:

- **Allow** (default): The cron job allows concurrently running jobs
- **Forbid**: The cron job does not allow concurrent runs; if it is time for a new job run and the previous job run hasn't finished yet, the cron job skips the new job run
- **Replace**: If it is time for a new job run and the previous job run hasn't finished yet, the cron job replaces the currently running job run with a new job run

Note that concurrency policy only applies to the jobs created by the same cron job. If there are multiple cron jobs, their respective jobs are always allowed to run concurrently.

## Suspend

The `.spec.suspend` field is also optional. If it is set to `true`, all subsequent executions are suspended. This setting does not apply to already started executions. Defaults to `false`.

**Caution:** Executions that are suspended during their scheduled time count as missed jobs. When `.spec.suspend` changes from `true` to `false` on an existing cron job without a [starting deadline](#), the missed jobs are scheduled immediately.

## Jobs History Limits

The `.spec.successfulJobsHistoryLimit` and `.spec.failedJobsHistoryLimit` fields are optional. These fields specify how many completed and failed jobs should be kept. By default, they are set to 3 and 1 respectively. Setting a limit to 0 corresponds to keeping none of the corresponding kind of jobs after they finish.

# Feedback

Was this page helpful?

Yes No

Thanks for the feedback. If you have a specific, answerable question about how to use Kubernetes, ask it on [Stack Overflow](#). Open an issue in the GitHub repo if you want to [report a problem](#) or [suggest an improvement](#).

---

[Create an Issue](#) [Edit This Page](#)

Page last modified on February 10, 2020 at 9:39 PM PST by [Update automated-tasks-with-cron-jobs.md \(#19043\)](#) ([Page History](#))

[Edit This Page](#)

# Parallel Processing using Expansions

In this example, we will run multiple Kubernetes Jobs created from a common template. You may want to be familiar with the basic, non-parallel, use of [Jobs](#) first.

- [Basic Template Expansion](#)
- [Multiple Template Parameters](#)
- [Alternatives](#)

## Basic Template Expansion

First, download the following template of a job to a file called `job-tmpl.yaml`

### [application/job/job-tmpl.yaml](#)

```
apiVersion: batch/v1
kind: Job
metadata:
  name: process-item-$ITEM
  labels:
    jobgroup: jobexample
spec:
  template:
    metadata:
      name: jobexample
      labels:
        jobgroup: jobexample
    spec:
      containers:
        - name: c
          image: busybox
          command: ["sh", "-c", "echo Processing item $ITEM && sleep 5"]
      restartPolicy: Never
```

Unlike a *pod template*, our *job template* is not a Kubernetes API type. It is just a yaml representation of a Job object that has some placeholders that need to be filled in before it can be used. The `$ITEM` syntax is not meaningful to Kubernetes.

In this example, the only processing the container does is to echo a string and sleep for a bit. In a real use case, the processing would be some substantial computation, such as rendering a frame of a movie, or processing a range of rows in a database. The `$ITEM` parameter would specify for example, the frame number or the row range.

This Job and its Pod template have a label: `jobgroup=jobexample`. There is nothing special to the system about this label. This label makes it convenient to operate on all the jobs in this group at once. We also put the same label on the pod template so that we can check on all Pods of these Jobs with a single command. After the job is created, the system will add more labels that distinguish one Job's pods from another Job's pods. Note that the label key `jobgroup` is not special to Kubernetes. You can pick your own label scheme.

Next, expand the template into multiple files, one for each item to be processed.

```
# Download job-templ.yaml
curl -L -s -0 https://k8s.io/examples/application/job/job-
tmpl.yaml

# Expand files into a temporary directory
mkdir ./jobs
for i in apple banana cherry
do
    cat job-tmpl.yaml | sed "s/\$ITEM/$i/" > ./jobs/job-$i.yaml
done
```

Check if it worked:

```
ls jobs/
```

The output is similar to this:

```
job-apple.yaml
job-banana.yaml
job-cherry.yaml
```

Here, we used `sed` to replace the string `$ITEM` with the loop variable. You could use any type of template language (jinja2, erb) or write a program to generate the Job objects.

Next, create all the jobs with one `kubectl` command:

```
kubectl create -f ./jobs
```

The output is similar to this:

```
job.batch/process-item-apple created
job.batch/process-item-banana created
job.batch/process-item-cherry created
```

Now, check on the jobs:

```
kubectl get jobs -l jobgroup=jobexample
```

The output is similar to this:

NAME	COMPLETIONS	DURATION	AGE
process-item-apple	1/1	14s	20s
process-item-banana	1/1	12s	20s
process-item-cherry	1/1	12s	20s

Here we use the `-l` option to select all jobs that are part of this group of jobs. (There might be other unrelated jobs in the system that we do not care to see.)

We can check on the pods as well using the same label selector:

```
kubectl get pods -l jobgroup=jobexample
```

The output is similar to this:

NAME	READY	STATUS	RESTARTS	AGE
process-item-apple-kixwv	0/1	Completed	0	4m
process-item-banana-wrsf7	0/1	Completed	0	4m
process-item-cherry-dnfu9	0/1	Completed	0	4m

We can use this single command to check on the output of all jobs at once:

```
kubectl logs -f -l jobgroup=jobexample
```

The output is:

```
Processing item apple
Processing item banana
Processing item cherry
```

## Multiple Template Parameters

In the first example, each instance of the template had one parameter, and that parameter was also used as a label. However label keys are limited in [what characters they can contain](#).

This slightly more complex example uses the jinja2 template language to generate our objects. We will use a one-line python script to convert the template to a file.

First, copy and paste the following template of a Job object, into a file called `job.yaml.jinja2`:

```
{%- set params = [{ "name": "apple", "url": "https://www.orangepippin.com/varieties/apples", },
                  { "name": "banana", "url": "https://en.wikipedia.org/wiki/Banana", },
                  { "name": "raspberry", "url": "https://www.raspberrypi.org/" }]
%}
{%- for p in params %}
{%- set name = p["name"] %}
{%- set url = p["url"] %}
```

```

apiVersion: batch/v1
kind: Job
metadata:
  name: jobexample-{{ name }}
  labels:
    jobgroup: jobexample
spec:
  template:
    metadata:
      name: jobexample
      labels:
        jobgroup: jobexample
    spec:
      containers:
        - name: c
          image: busybox
          command: ["sh", "-c", "echo Processing URL {{ url }} &&
sleep 5"]
      restartPolicy: Never
    ---
{%- endfor %}

```

The above template defines parameters for each job object using a list of python dicts (lines 1-4). Then a for loop emits one job yaml object for each set of parameters (remaining lines). We take advantage of the fact that multiple yaml documents can be concatenated with the --- separator (second to last line). ) We can pipe the output directly to kubectl to create the objects.

You will need the jinja2 package if you do not already have it: pip install --user jinja2. Now, use this one-line python program to expand the template:

```
alias render_template='python -c "from jinja2 import Template;
import sys; print(Template(sys.stdin.read()).render());'"
```

The output can be saved to a file, like this:

```
cat job.yaml.jinja2 | render_template > jobs.yaml
```

Or sent directly to kubectl, like this:

```
cat job.yaml.jinja2 | render_template | kubectl apply -f -
```

## Alternatives

If you have a large number of job objects, you may find that:

- Even using labels, managing so many Job objects is cumbersome.
- You exceed resource quota when creating all the Jobs at once, and do not want to wait to create them incrementally.

- Very large numbers of jobs created at once overload the Kubernetes apiserver, controller, or scheduler.

In this case, you can consider one of the other [job patterns](#).

## Feedback

Was this page helpful?

Yes No

Thanks for the feedback. If you have a specific, answerable question about how to use Kubernetes, ask it on [Stack Overflow](#). Open an issue in the GitHub repo if you want to [report a problem](#) or [suggest an improvement](#).

---

[Edit This Page](#)

# Coarse Parallel Processing Using a Work Queue

In this example, we will run a Kubernetes Job with multiple parallel worker processes.

In this example, as each pod is created, it picks up one unit of work from a task queue, completes it, deletes it from the queue, and exits.

Here is an overview of the steps in this example:

1. **Start a message queue service.** In this example, we use RabbitMQ, but you could use another one. In practice you would set up a message queue service once and reuse it for many jobs.
  2. **Create a queue, and fill it with messages.** Each message represents one task to be done. In this example, a message is just an integer that we will do a lengthy computation on.
  3. **Start a Job that works on tasks from the queue.** The Job starts several pods. Each pod takes one task from the message queue, processes it, and repeats until the end of the queue is reached.
- [Before you begin](#)
  - [Starting a message queue service](#)
  - [Testing the message queue service](#)
  - [Filling the Queue with tasks](#)
  - [Create an Image](#)
  - [Defining a Job](#)
  - [Running the Job](#)
  - [Alternatives](#)
  - [Caveats](#)

## Before you begin

Be familiar with the basic, non-parallel, use of [Job](#).

You need to have a Kubernetes cluster, and the kubectl command-line tool must be configured to communicate with your cluster. 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](#)

# Starting a message queue service

This example uses RabbitMQ, but it should be easy to adapt to another AMQP-type message service.

In practice you could set up a message queue service once in a cluster and reuse it for many jobs, as well as for long-running services.

Start RabbitMQ as follows:

```
kubectl create -f https://raw.githubusercontent.com/kubernetes/
kubernetes/release-1.3/examples/celery-rabbitmq/rabbitmq-
service.yaml
```

```
service "rabbitmq-service" created
```

```
kubectl create -f https://raw.githubusercontent.com/kubernetes/
kubernetes/release-1.3/examples/celery-rabbitmq/rabbitmq-
controller.yaml
```

```
replicationcontroller "rabbitmq-controller" created
```

We will only use the rabbitmq part from the [celery-rabbitmq example](#).

## Testing the message queue service

Now, we can experiment with accessing the message queue. We will create a temporary interactive pod, install some tools on it, and experiment with queues.

First create a temporary interactive Pod.

```
# Create a temporary interactive container
kubectl run -i --tty temp --image ubuntu:18.04
```

```
Waiting for pod default/temp-loe07 to be running, status is
Pending, pod ready: false
... [ previous line repeats several times .. hit return when it
stops ] ...
```

Note that your pod name and command prompt will be different.

Next install the amqp-tools so we can work with message queues.

```
# Install some tools
root@temp-loe07:/# apt-get update
.... [ lots of output ] ....
root@temp-loe07:/# apt-get install -y curl ca-certificates amqp-
tools python dnsutils
.... [ lots of output ] ....
```

Later, we will make a docker image that includes these packages.

Next, we will check that we can discover the rabbitmq service:

```
# Note the rabbitmq-service has a DNS name, provided by
Kubernetes:
```

```
root@temp-loe07:/# nslookup rabbitmq-service
Server:      10.0.0.10
Address:    10.0.0.10#53

Name:      rabbitmq-service.default.svc.cluster.local
Address:  10.0.147.152

# Your address will vary.
```

If Kube-DNS is not setup correctly, the previous step may not work for you. You can also find the service IP in an env var:

```
# env | grep RABBIT | grep HOST
RABBITMQ_SERVICE_SERVICE_HOST=10.0.147.152
# Your address will vary.
```

Next we will verify we can create a queue, and publish and consume messages.

```
# In the next line, rabbitmq-service is the hostname where the
# rabbitmq-service
# can be reached. 5672 is the standard port for rabbitmq.
```

```
root@temp-loe07:/# export BROKER_URL=amqp://guest:guest@rabbitmq-
service:5672
# If you could not resolve "rabbitmq-service" in the previous
step,
# then use this command instead:
# root@temp-loe07:/# BROKER_URL=amqp://
guest:guest@$RABBITMQ_SERVICE_SERVICE_HOST:5672
```

```
# Now create a queue:
```

```
root@temp-loe07:/# /usr/bin/amqp-declare-queue --url=$BROKER_URL
-q foo -d
foo
```

```
# Publish one message to it:
```

```
root@temp-loe07:/# /usr/bin/amqp-publish --url=$BROKER_URL -r
foo -p -b Hello
```

```
# And get it back.
```

```
root@temp-loe07:/# /usr/bin/amqp-consume --url=$BROKER_URL -q
foo -c 1 cat && echo
```

```
Hello  
root@temp-loe07:/#
```

In the last command, the `amqp-consume` tool takes one message (`-c 1`) from the queue, and passes that message to the standard input of an arbitrary command. In this case, the program `cat` is just printing out what it gets on the standard input, and the `echo` is just to add a carriage return so the example is readable.

## Filling the Queue with tasks

Now let's fill the queue with some "tasks". In our example, our tasks are just strings to be printed.

In practice, the content of the messages might be:

- names of files to that need to be processed
- extra flags to the program
- ranges of keys in a database table
- configuration parameters to a simulation
- frame numbers of a scene to be rendered

In practice, if there is large data that is needed in a read-only mode by all pods of the Job, you will typically put that in a shared file system like NFS and mount that readonly on all the pods, or the program in the pod will natively read data from a cluster file system like HDFS.

For our example, we will create the queue and fill it using the `amqp` command line tools. In practice, you might write a program to fill the queue using an `amqp` client library.

```
/usr/bin/amqp-declare-queue --url=$BROKER_URL -q job1 -d  
job1  
  
for f in apple banana cherry date fig grape lemon melon  
do  
  /usr/bin/amqp-publish --url=$BROKER_URL -r job1 -p -b $f  
done
```

So, we filled the queue with 8 messages.

## Create an Image

Now we are ready to create an image that we will run as a job.

We will use the `amqp-consume` utility to read the message from the queue and run our actual program. Here is a very simple example program:

### [application/job/rabbitmq/worker.py](#)

```
#!/usr/bin/env python

# Just prints standard out and sleeps for 10 seconds.
import sys
import time
print("Processing " + sys.stdin.readlines()[0])
time.sleep(10)
```

Give the script execution permission:

```
chmod +x worker.py
```

Now, build an image. If you are working in the source tree, then change directory to examples/job/work-queue-1. Otherwise, make a temporary directory, change to it, download the [Dockerfile](#), and [worker.py](#). In either case, build the image with this command:

```
docker build -t job-wq-1 .
```

For the [Docker Hub](#), tag your app image with your username and push to the Hub with the below commands. Replace <username> with your Hub username.

```
docker tag job-wq-1 <username>/job-wq-1
docker push <username>/job-wq-1
```

If you are using [Google Container Registry](#), tag your app image with your project ID, and push to GCR. Replace <project> with your project ID.

```
docker tag job-wq-1 gcr.io/<project>/job-wq-1
gcloud docker -- push gcr.io/<project>/job-wq-1
```

## Defining a Job

Here is a job definition. You'll need to make a copy of the Job and edit the image to match the name you used, and call it `./job.yaml`.

### [application/job/rabbitmq/job.yaml](#)

```
apiVersion: batch/v1
kind: Job
metadata:
  name: job-wq-1
spec:
  completions: 8
  parallelism: 2
  template:
    metadata:
      name: job-wq-1
    spec:
      containers:
        - name: c
          image: gcr.io/<project>/job-wq-1
          env:
            - name: BROKER_URL
              value: amqp://guest:guest@rabbitmq-service:5672
            - name: QUEUE
              value: job1
  restartPolicy: OnFailure
```

In this example, each pod works on one item from the queue and then exits. So, the completion count of the Job corresponds to the number of work items done. So we set, `.spec.completions: 8` for the example, since we put 8 items in the queue.

## Running the Job

So, now run the Job:

```
kubectl apply -f ./job.yaml
```

Now wait a bit, then check on the job.

```
kubectl describe jobs/job-wq-1
```

Name:	job-wq-1
Namespace:	default
Selector:	controller-uid=41d75705-92df-11e7-b85e-fa163ee3c11f
Labels:	controller-uid=41d75705-92df-11e7-b85e-fa163ee3c11f
Annotations:	job-name=job-wq-1
Parallelism:	2
Completions:	8
Start Time:	Wed, 06 Sep 2017 16:42:02 +0800
Pods Statuses:	0 Running / 8 Succeeded / 0 Failed

```

Pod Template:
  Labels:      controller-uid=41d75705-92df-11e7-b85e-
  fa163ee3c11f          job-name=job-wq-1
  Containers:
    c:
      Image:      gcr.io/causal-jigsaw-637/job-wq-1
      Port:
      Environment:
        BROKER_URL:      amqp://guest:guest@rabbitmq-service:5672
        QUEUE:            job1
        Mounts:
        Volumes:
Events:
  FirstSeen  LastSeen   Count  From           SubobjectPath
Type      Reason      Message
  27s       27s        1      {job }
Normal   SuccessfulCreate  Created pod: job-wq-1-hcobb
  27s       27s        1      {job }
Normal   SuccessfulCreate  Created pod: job-wq-1-weytj
  27s       27s        1      {job }
Normal   SuccessfulCreate  Created pod: job-wq-1-qaam5
  27s       27s        1      {job }
Normal   SuccessfulCreate  Created pod: job-wq-1-b67sr
  26s       26s        1      {job }
Normal   SuccessfulCreate  Created pod: job-wq-1-xe5hj
  15s       15s        1      {job }
Normal   SuccessfulCreate  Created pod: job-wq-1-w2zqe
  14s       14s        1      {job }
Normal   SuccessfulCreate  Created pod: job-wq-1-d6ppa
  14s       14s        1      {job }
Normal   SuccessfulCreate  Created pod: job-wq-1-p17e0

```

All our pods succeeded. Yay.

## Alternatives

This approach has the advantage that you do not need to modify your "worker" program to be aware that there is a work queue.

It does require that you run a message queue service. If running a queue service is inconvenient, you may want to consider one of the other [job patterns](#).

This approach creates a pod for every work item. If your work items only take a few seconds, though, creating a Pod for every work item may add a

lot of overhead. Consider another [example](#), that executes multiple work items per Pod.

In this example, we use the `amqp-consume` utility to read the message from the queue and run our actual program. This has the advantage that you do not need to modify your program to be aware of the queue. A [different example](#), shows how to communicate with the work queue using a client library.

## Caveats

If the number of completions is set to less than the number of items in the queue, then not all items will be processed.

If the number of completions is set to more than the number of items in the queue, then the Job will not appear to be completed, even though all items in the queue have been processed. It will start additional pods which will block waiting for a message.

There is an unlikely race with this pattern. If the container is killed in between the time that the message is acknowledged by the `amqp-consume` command and the time that the container exits with success, or if the node crashes before the kubelet is able to post the success of the pod back to the api-server, then the Job will not appear to be complete, even though all items in the queue have been processed.

## Feedback

Was this page helpful?

Yes No

Thanks for the feedback. If you have a specific, answerable question about how to use Kubernetes, ask it on [Stack Overflow](#). Open an issue in the GitHub repo if you want to [report a problem](#) or [suggest an improvement](#).

---

[Create an Issue](#) [Edit This Page](#)

Page last modified on January 10, 2020 at 8:52 PM PST by [Remove version checks for Job and CronJob task pages \(#18337\)](#) ([Page History](#))

[Edit This Page](#)

# Fine Parallel Processing Using a Work Queue

In this example, we will run a Kubernetes Job with multiple parallel worker processes in a given pod.

In this example, as each pod is created, it picks up one unit of work from a task queue, processes it, and repeats until the end of the queue is reached.

Here is an overview of the steps in this example:

1. **Start a storage service to hold the work queue.** In this example, we use Redis to store our work items. In the previous example, we used RabbitMQ. In this example, we use Redis and a custom work-queue client library because AMQP does not provide a good way for clients to detect when a finite-length work queue is empty. In practice you would set up a store such as Redis once and reuse it for the work queues of many jobs, and other things.
  2. **Create a queue, and fill it with messages.** Each message represents one task to be done. In this example, a message is just an integer that we will do a lengthy computation on.
  3. **Start a Job that works on tasks from the queue.** The Job starts several pods. Each pod takes one task from the message queue, processes it, and repeats until the end of the queue is reached.
- [Before you begin](#)
  - [Starting Redis](#)
  - [Filling the Queue with tasks](#)
  - [Create an Image](#)
  - [Defining a Job](#)
  - [Running the Job](#)
  - [Alternatives](#)

## Before you begin

You need to have a Kubernetes cluster, and the kubectl command-line tool must be configured to communicate with your cluster. 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](#)

## Starting Redis

For this example, for simplicity, we will start a single instance of Redis. See the [Redis Example](#) for an example of deploying Redis scalably and redundantly.

You could also download the following files directly:

- [redis-pod.yaml](#)
- [redis-service.yaml](#)
- [Dockerfile](#)
- [job.yaml](#)
- [rediswq.py](#)
- [worker.py](#)

# Filling the Queue with tasks

Now let's fill the queue with some "tasks". In our example, our tasks are just strings to be printed.

Start a temporary interactive pod for running the Redis CLI.

```
kubectl run -i --tty temp --image redis --command "/bin/sh"
Waiting for pod default/redis2-c7h78 to be running, status is
Pending, pod ready: false
Hit enter for command prompt
```

Now hit enter, start the redis CLI, and create a list with some work items in it.

```
# redis-cli -h redis
redis:6379> rpush job2 "apple"
(integer) 1
redis:6379> rpush job2 "banana"
(integer) 2
redis:6379> rpush job2 "cherry"
(integer) 3
redis:6379> rpush job2 "date"
(integer) 4
redis:6379> rpush job2 "fig"
(integer) 5
redis:6379> rpush job2 "grape"
(integer) 6
redis:6379> rpush job2 "lemon"
(integer) 7
redis:6379> rpush job2 "melon"
(integer) 8
redis:6379> rpush job2 "orange"
(integer) 9
redis:6379> lrange job2 0 -1
1) "apple"
2) "banana"
3) "cherry"
4) "date"
5) "fig"
6) "grape"
7) "lemon"
8) "melon"
9) "orange"
```

So, the list with key job2 will be our work queue.

Note: if you do not have Kube DNS setup correctly, you may need to change the first step of the above block to `redis-cli -h $REDIS_SERVICE_HOST`.

# Create an Image

Now we are ready to create an image that we will run.

We will use a python worker program with a redis client to read the messages from the message queue.

A simple Redis work queue client library is provided, called `rediswq.py` ([Download](#)).

The "worker" program in each Pod of the Job uses the work queue client library to get work. Here it is:

## [application/job/redis/worker.py](#)

```
#!/usr/bin/env python

import time
import rediswq

host="redis"
# Uncomment next two lines if you do not have Kube-DNS working.
# import os
# host = os.getenv("REDIS_SERVICE_HOST")

q = rediswq.RedisWQ(name="job2", host="redis")
print("Worker with sessionID: " + q.sessionID())
print("Initial queue state: empty=" + str(q.empty()))
while not q.empty():
    item = q.lease(lease_secs=10, block=True, timeout=2)
    if item is not None:
        itemstr = item.decode("utf-8")
        print("Working on " + itemstr)
        time.sleep(10) # Put your actual work here instead of sleep.
        q.complete(item)
    else:
        print("Waiting for work")
print("Queue empty, exiting")
```

You could also download [worker.py](#), [rediswq.py](#), and [Dockerfile](#) files, then build the image:

```
docker build -t job-wq-2 .
```

## Push the image

For the [Docker Hub](#), tag your app image with your username and push to the Hub with the below commands. Replace <username> with your Hub username.

```
docker tag job-wq-2 <username>/job-wq-2
docker push <username>/job-wq-2
```

You need to push to a public repository or [configure your cluster to be able to access your private repository](#).

If you are using [Google Container Registry](#), tag your app image with your project ID, and push to GCR. Replace <project> with your project ID.

```
docker tag job-wq-2 gcr.io/<project>/job-wq-2
gcloud docker -- push gcr.io/<project>/job-wq-2
```

## Defining a Job

Here is the job definition:

### [application/job/redis/job.yaml](#)

```
apiVersion: batch/v1
kind: Job
metadata:
  name: job-wq-2
spec:
  parallelism: 2
  template:
    metadata:
      name: job-wq-2
    spec:
      containers:
        - name: c
          image: gcr.io/myproject/job-wq-2
        restartPolicy: OnFailure
```

Be sure to edit the job template to change `gcr.io/myproject` to your own path.

In this example, each pod works on several items from the queue and then exits when there are no more items. Since the workers themselves detect when the workqueue is empty, and the Job controller does not know about the workqueue, it relies on the workers to signal when they are done working. The workers signal that the queue is empty by exiting with success. So, as soon as any worker exits with success, the controller knows the work is done, and the Pods will exit soon. So, we set the completion count of the Job to 1. The job controller will wait for the other pods to complete too.

## Running the Job

So, now run the Job:

```
kubectl apply -f ./job.yaml
```

Now wait a bit, then check on the job.

```
kubectl describe jobs/job-wq-2
Name:           job-wq-2
Namespace:      default
Selector:       controller-uid=b1c7e4e3-92e1-11e7-b85e-
                fa163ee3c11f
Labels:         controller-uid=b1c7e4e3-92e1-11e7-b85e-
                fa163ee3c11f
                job-name=job-wq-2
Annotations:   <none>
Parallelism:   2
Completions:   <unset>
Start Time:    Mon, 11 Jan 2016 17:07:59 -0800
Pods Statuses: 1 Running / 0 Succeeded / 0 Failed
Pod Template:
  Labels:      controller-uid=b1c7e4e3-92e1-11e7-b85e-
                fa163ee3c11f
                job-name=job-wq-2
  Containers:
    c:
      Image:      gcr.io/exampleproject/job-wq-2
      Port:       8080
      Environment: <none>
      Mounts:     <none>
      Volumes:    <none>
Events:
FirstSeen  LastSeen  Count  From                    Message
SubobjectPath  Type      Reason
-----  -----  -----  -----
33s        33s       1      {job-controller } 
Normal      SuccessfulCreate  Created pod: job-wq-2-lglf8
```

```
kubectl logs pods/job-wq-2-7r7b2
Worker with sessionID: bbd72d0a-9e5c-4dd6-abf6-416cc267991f
Initial queue state: empty=False
Working on banana
Working on date
Working on lemon
```

As you can see, one of our pods worked on several work units.

## Alternatives

If running a queue service or modifying your containers to use a work queue is inconvenient, you may want to consider one of the other [job patterns](#).

If you have a continuous stream of background processing work to run, then consider running your background workers with a ReplicaSet instead, and consider running a background processing library such as <https://github.com/resque/resque>.

## Feedback

Was this page helpful?

Yes No

Thanks for the feedback. If you have a specific, answerable question about how to use Kubernetes, ask it on [Stack Overflow](#). Open an issue in the GitHub repo if you want to [report a problem](#) or [suggest an improvement](#).

---

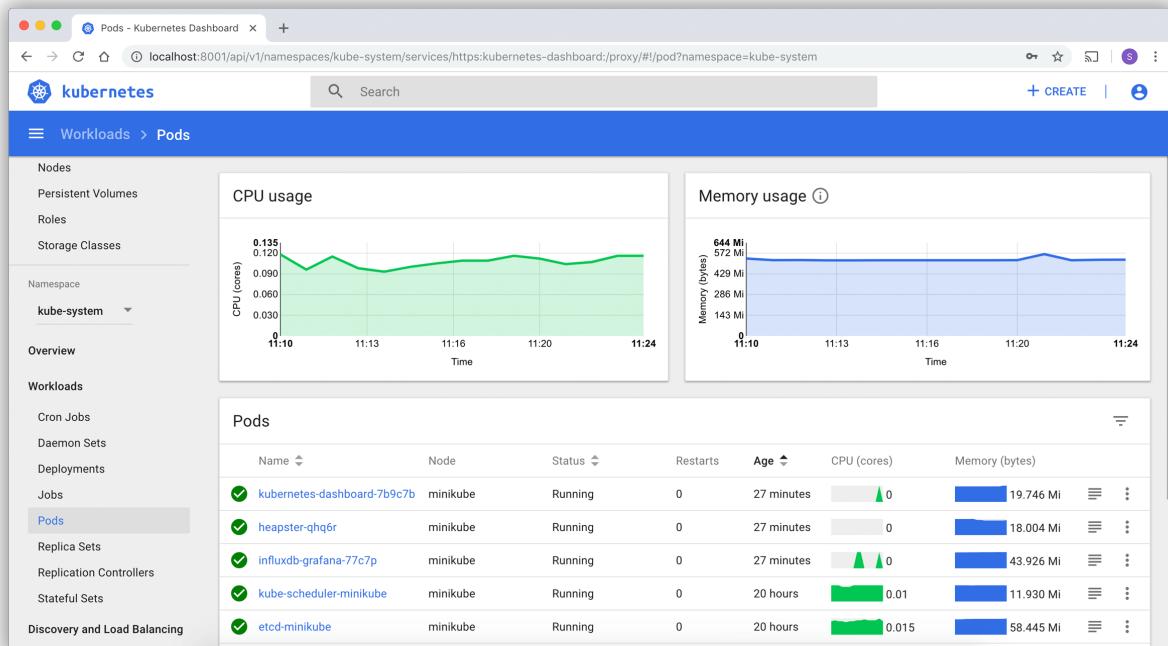
Page last modified on January 10, 2020 at 8:52 PM PST by [Remove version checks for Job and CronJob task pages \(#18337\)](#) ([Page History](#))

[Edit This Page](#)

# Web UI (Dashboard)

Dashboard is a web-based Kubernetes user interface. You can use Dashboard to deploy containerized applications to a Kubernetes cluster, troubleshoot your containerized application, and manage the cluster resources. You can use Dashboard to get an overview of applications running on your cluster, as well as for creating or modifying individual Kubernetes resources (such as Deployments, Jobs, DaemonSets, etc). For example, you can scale a Deployment, initiate a rolling update, restart a pod or deploy new applications using a deploy wizard.

Dashboard also provides information on the state of Kubernetes resources in your cluster and on any errors that may have occurred.



- [Deploying the Dashboard UI](#)
- [Accessing the Dashboard UI](#)
- [Welcome view](#)
- [Deploying containerized applications](#)
- [Using Dashboard](#)
- [What's next](#)

# Deploying the Dashboard UI

The Dashboard UI is not deployed by default. To deploy it, run the following command:

```
kubectl apply -f https://raw.githubusercontent.com/kubernetes/dashboard/v2.0.0-beta8/aio/deploy/recommended.yaml
```

## Accessing the Dashboard UI

To protect your cluster data, Dashboard deploys with a minimal RBAC configuration by default. Currently, Dashboard only supports logging in with a Bearer Token. To create a token for this demo, you can follow our guide on [creating a sample user](#).

**Warning:** The sample user created in the tutorial will have administrative privileges and is for educational purposes only.

### Command line proxy

You can access Dashboard using the kubectl command-line tool by running the following command:

```
kubectl proxy
```

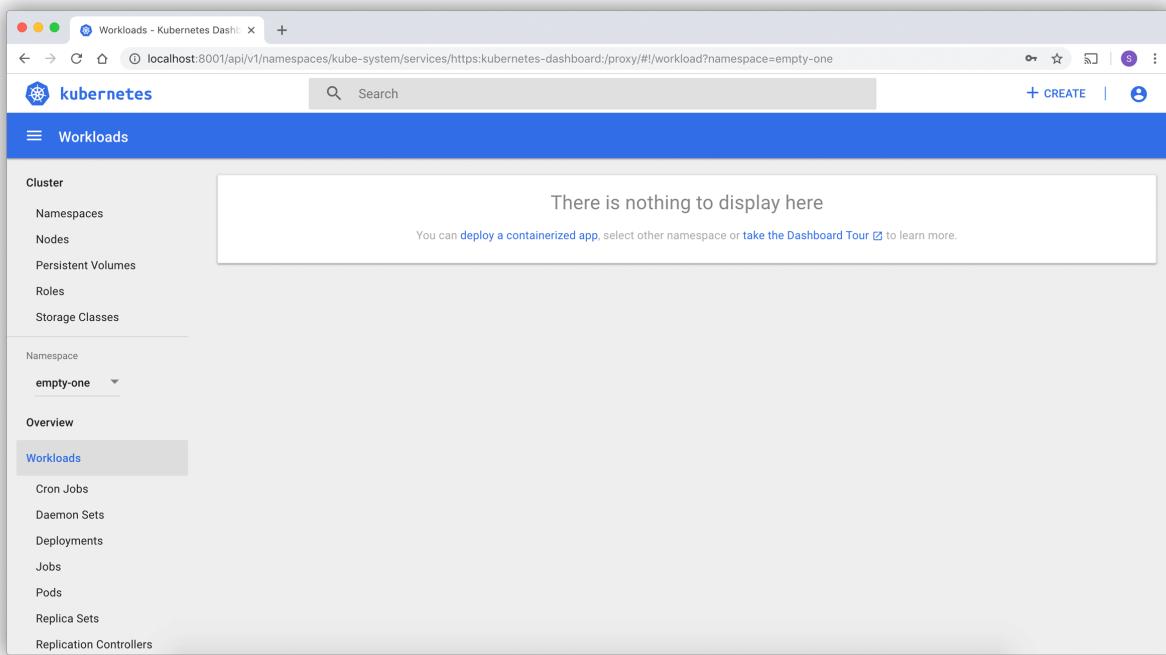
Kubectl will make Dashboard available at <http://localhost:8001/api/v1/namespaces/kubernetes-dashboard/services/https:kubernetes-dashboard:/proxy/>.

The UI can *only* be accessed from the machine where the command is executed. See `kubectl proxy --help` for more options.

**Note:** Kubeconfig Authentication method does NOT support external identity providers or x509 certificate-based authentication.

## Welcome view

When you access Dashboard on an empty cluster, you'll see the welcome page. This page contains a link to this document as well as a button to deploy your first application. In addition, you can view which system applications are running by default in the kube-system [namespace](#) of your cluster, for example the Dashboard itself.



## Deploying containerized applications

Dashboard lets you create and deploy a containerized application as a Deployment and optional Service with a simple wizard. You can either manually specify application details, or upload a YAML or JSON file containing application configuration.

Click the **CREATE** button in the upper right corner of any page to begin.

### Specifying application details

The deploy wizard expects that you provide the following information:

- **App name** (mandatory): Name for your application. A [label](#) with the name will be added to the Deployment and Service, if any, that will be deployed.

The application name must be unique within the selected Kubernetes [namespace](#). It must start with a lowercase character, and end with a lowercase character or a number, and contain only lowercase letters, numbers and dashes (-). It is limited to 24 characters. Leading and trailing spaces are ignored.

- **Container image** (mandatory): The URL of a public Docker [container image](#) on any registry, or a private image (commonly hosted on the Google Container Registry or Docker Hub). The container image specification must end with a colon.
- **Number of pods** (mandatory): The target number of Pods you want your application to be deployed in. The value must be a positive integer.

A [Deployment](#) will be created to maintain the desired number of Pods across your cluster.

- **Service** (optional): For some parts of your application (e.g. frontends) you may want to expose a [Service](#) onto an external, maybe public IP address outside of your cluster (external Service). For external Services, you may need to open up one or more ports to do so. Find more details [here](#).

Other Services that are only visible from inside the cluster are called internal Services.

Irrespective of the Service type, if you choose to create a Service and your container listens on a port (incoming), you need to specify two ports. The Service will be created mapping the port (incoming) to the target port seen by the container. This Service will route to your deployed Pods. Supported protocols are TCP and UDP. The internal DNS name for this Service will be the value you specified as application name above.

If needed, you can expand the **Advanced options** section where you can specify more settings:

- **Description:** The text you enter here will be added as an [annotation](#) to the Deployment and displayed in the application's details.
- **Labels:** Default [labels](#) to be used for your application are application name and version. You can specify additional labels to be applied to the Deployment, Service (if any), and Pods, such as release, environment, tier, partition, and release track.

Example:

```
release=1.0
tier=frontend
environment=prod
track=stable
```

- **Namespace:** Kubernetes supports multiple virtual clusters backed by the same physical cluster. These virtual clusters are called [namespaces](#). They let you partition resources into logically named groups.

Dashboard offers all available namespaces in a dropdown list, and allows you to create a new namespace. The namespace name may contain a maximum of 63 alphanumeric characters and dashes (-) but can not contain capital letters. Namespace names should not consist of only numbers. If the name is set as a number, such as 10, the pod will be put in the default namespace.

In case the creation of the namespace is successful, it is selected by default. If the creation fails, the first namespace is selected.

- **Image Pull Secret:** In case the specified Docker container image is private, it may require [pull secret](#) credentials.

Dashboard offers all available secrets in a dropdown list, and allows you to create a new secret. The secret name must follow the DNS domain name syntax, e.g. `new.image-pull.secret`. The content of a secret must be base64-encoded and specified in a [.dockercfg](#) file. The secret name may consist of a maximum of 253 characters.

In case the creation of the image pull secret is successful, it is selected by default. If the creation fails, no secret is applied.

- **CPU requirement (cores) and Memory requirement (MiB):** You can specify the minimum [resource limits](#) for the container. By default, Pods run with unbounded CPU and memory limits.
- **Run command and Run command arguments:** By default, your containers run the specified Docker image's default [entrypoint command](#). You can use the command options and arguments to override the default.
- **Run as privileged:** This setting determines whether processes in [privileged containers](#) are equivalent to processes running as root on the host. Privileged containers can make use of capabilities like manipulating the network stack and accessing devices.
- **Environment variables:** Kubernetes exposes Services through [environment variables](#). You can compose environment variable or pass arguments to your commands using the values of environment variables. They can be used in applications to find a Service. Values can reference other variables using the `$(VAR_NAME)` syntax.

## Uploading a YAML or JSON file

Kubernetes supports declarative configuration. In this style, all configuration is stored in YAML or JSON configuration files using the Kubernetes [API](#) resource schemas.

As an alternative to specifying application details in the deploy wizard, you can define your application in YAML or JSON files, and upload the files using Dashboard.

# Using Dashboard

Following sections describe views of the Kubernetes Dashboard UI; what they provide and how can they be used.

## Navigation

When there are Kubernetes objects defined in the cluster, Dashboard shows them in the initial view. By default only objects from the *default* namespace are shown and this can be changed using the namespace selector located in the navigation menu.

Dashboard shows most Kubernetes object kinds and groups them in a few menu categories.

## **Admin Overview**

For cluster and namespace administrators, Dashboard lists Nodes, Namespaces and Persistent Volumes and has detail views for them. Node list view contains CPU and memory usage metrics aggregated across all Nodes. The details view shows the metrics for a Node, its specification, status, allocated resources, events and pods running on the node.

## **Workloads**

Shows all applications running in the selected namespace. The view lists applications by workload kind (e.g., Deployments, Replica Sets, Stateful Sets, etc.) and each workload kind can be viewed separately. The lists summarize actionable information about the workloads, such as the number of ready pods for a Replica Set or current memory usage for a Pod.

Detail views for workloads show status and specification information and surface relationships between objects. For example, Pods that Replica Set is controlling or New Replica Sets and Horizontal Pod Autoscalers for Deployments.

## **Services**

Shows Kubernetes resources that allow for exposing services to external world and discovering them within a cluster. For that reason, Service and Ingress views show Pods targeted by them, internal endpoints for cluster connections and external endpoints for external users.

## **Storage**

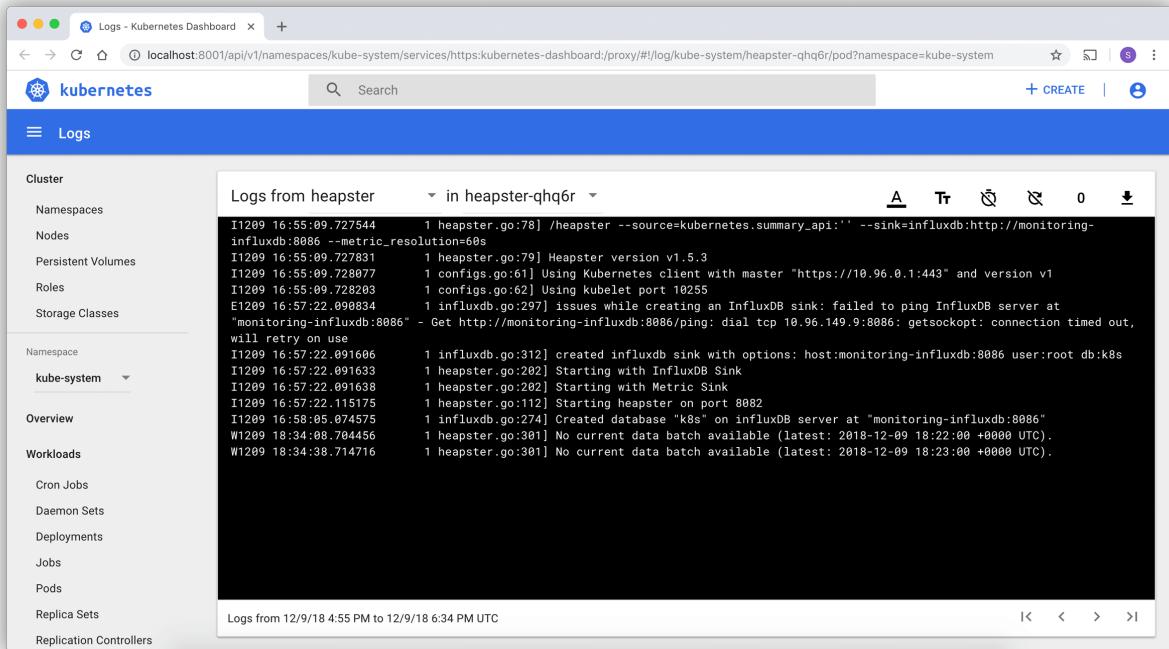
Storage view shows Persistent Volume Claim resources which are used by applications for storing data.

## **Config Maps and Secrets**

Shows all Kubernetes resources that are used for live configuration of applications running in clusters. The view allows for editing and managing config objects and displays secrets hidden by default.

## **Logs viewer**

Pod lists and detail pages link to a logs viewer that is built into Dashboard. The viewer allows for drilling down logs from containers belonging to a single Pod.



## What's next

For more information, see the [Kubernetes Dashboard project page](#).

## Feedback

Was this page helpful?

Yes No

Thanks for the feedback. If you have a specific, answerable question about how to use Kubernetes, ask it on [Stack Overflow](#). Open an issue in the GitHub repo if you want to [report a problem](#) or [suggest an improvement](#).

---

[Create an Issue](#) [Edit This Page](#)

Page last modified on December 17, 2019 at 8:17 PM PST by [Update k/dashboard to v2.0.0-beta8 \(#17994\)](#) ([Page History](#))

[Edit This Page](#)

# Accessing Clusters

This topic discusses multiple ways to interact with clusters.

- [Accessing for the first time with kubectl](#)
- [Directly accessing the REST API](#)
- [Programmatic access to the API](#)
- [Accessing the API from a Pod](#)
- [Accessing services running on the cluster](#)
- [Requesting redirects](#)

- [So Many Proxies](#)

## Accessing for the first time with kubectl

When accessing the Kubernetes API for the first time, we suggest using the Kubernetes CLI, `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` and complete documentation is found in the [kubectl manual](#).

## Directly accessing the REST API

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

- Run `kubectl` in proxy mode.
  - Recommended approach.
  - Uses stored `apiserver` location.
  - Verifies identity of `apiserver` using self-signed cert. No MITM possible.
  - Authenticates to `apiserver`.
  - In future, may do intelligent client-side load-balancing and failover.
- Provide the location and credentials directly to the http client.
  - Alternate approach.
  - Works with some types of client code that are confused by using a proxy.
  - Need to import a root cert into your browser to protect against MITM.

## Using `kubectl proxy`

The following command runs `kubectl` in a mode where it acts as a reverse proxy. It handles locating the `apiserver` 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, replacing `localhost` with `[::1]` for IPv6, like so:

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

The output is similar to this:

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

## Without kubectl proxy

Use `kubectl describe secret...` to get the token for the default service account with grep/cut:

```
APISERVER=$(kubectl config view --minify | grep server | cut -f  
2- -d ":" | tr -d " ")  
SECRET_NAME=$(kubectl get secrets | grep ^default | cut -f1 -d '  
' )  
TOKEN=$(kubectl describe secret $SECRET_NAME | grep -E '^token'  
| cut -f2 -d ':' | tr -d " ")  
  
curl $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:

```
APISERVER=$(kubectl config view --minify -o jsonpath='{.clusters[  
0].cluster.server}')  
SECRET_NAME=$(kubectl get serviceaccount default -o jsonpath='{.s
```

```

secrets[0].name')
TOKEN=$(kubectl get secret $SECRET_NAME -o jsonpath='{.data.token}'
} | base64 --decode)

curl $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"
    }
  ]
}
```

The above examples use 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 apiserver does not require authentication; it may serve on localhost, or be protected by a firewall. There is not a standard for this. [Configuring Access to the API](#) describes how a cluster admin can configure this. Such approaches may conflict with future high-availability support.

## Programmatic access to the API

Kubernetes officially supports [Go](#) and [Python](#) client libraries.

### Go client

- To get the library, run the following command: `go get k8s.io/client-go@kubernetes-<kubernetes-version-number>`, see [INSTALL.md](#) for detailed installation instructions. See <https://github.com/kubernetes/client-go> to see which versions are supported.
- Write an application atop of the client-go clients. Note that client-go defines its own API objects, so if needed, please import API definitions from client-go rather than from the main repository, e.g., `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 apiserver. See this [example](#).

If the application is deployed as a Pod in the cluster, please refer to the [next section](#).

## 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 apiserver. See this [example](#).

## Other languages

There are [client libraries](#) for accessing the API from other languages. See documentation for other libraries for how they authenticate.

## Accessing the API from a Pod

When accessing the API from a pod, locating and authenticating to the apiserver are somewhat different.

The recommended way to locate the apiserver within the pod is with the `kubernetes.default.svc` DNS name, which resolves to a Service IP which in turn will be routed to an apiserver.

The recommended way to authenticate to the apiserver is with a [service account](#) credential. By kube-system, a pod is associated with a service account, and a credential (token) for that service account is placed into the filesystem tree of each container in that pod, at `/var/run/secrets/kubernetes.io/serviceaccount/token`.

If available, a certificate bundle is placed into the filesystem tree of each container at `/var/run/secrets/kubernetes.io/serviceaccount/ca.crt`, and should be used to verify the serving certificate of the apiserver.

Finally, the default namespace to be used for namespaced API operations is placed in a file at `/var/run/secrets/kubernetes.io/serviceaccount/namespace` in each container.

From within a pod the recommended ways to connect to API are:

- Run `kubectl proxy` in a sidecar container in the pod, or as a background process within the container. This proxies the Kubernetes API to the localhost interface of the pod, so that other processes in any container of the pod can access it.
- Use the Go client library, and create a client using the `rest.InClusterConfig()` and `kubernetes.NewForConfig()` functions. They handle locating and authenticating to the apiserver. [example](#)

In each case, the credentials of the pod are used to communicate securely with the apiserver.

# Accessing services running on the cluster

The previous section was about connecting the Kubernetes API server. This section is about connecting to other services running on Kubernetes cluster. In Kubernetes, the [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 just 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/elasticsearch-logging/proxy
kibana-logging is running at https://104.197.5.247/api/v1/
namespaces/kube-system/services/kibana-logging/proxy
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-grafana/proxy
heapster is running at https://104.197.5.247/api/v1/namespaces/
kube-system/services/monitoring-heapster/proxy
```

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. Logging can also be reached through a kubectl proxy, for example at: `http://localhost:8080/api/v1/namespaces/kube-system/services/elasticsearch-logging/proxy/`. (See [above](#) 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 simply append to the service's proxy URL: `http://kubernetes_master_address/api/v1/namespaces/namespace_name/services/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.

By default, the API server proxies to your service using http. To use https, prefix the service name with https:: `http://kubernetes_master_address/api/v1/namespaces/namespace_name/services/https:service_name:[port_name]/proxy`

The supported formats for the name segment of the URL are:

- `<service_name>` - proxies to the default or unnamed port using http
- `<service_name>:<port_name>` - proxies to the specified port using http
- `https:<service_name>:` - proxies to the default or unnamed port using https (note the trailing colon)
- `https:<service_name>:<port_name>` - proxies to the specified port using https

## 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/_search?  
q=user:kimchy
```

- 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/_cluster/health?pretty=true`

```
{  
  "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  
}
```

## 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.

## Requesting redirects

The redirect capabilities have been deprecated and removed. Please use a proxy (see below) instead.

## So Many Proxies

There are several different proxies you may encounter when using Kubernetes:

1. The [kubectl proxy](#):

- runs on a user's desktop or in a pod
- proxies from a localhost address to the Kubernetes apiserver
- client to proxy uses HTTP
- proxy to apiserver uses HTTPS
- locates apiserver

- adds authentication headers

## 2. The [apiserver proxy](#):

- is a bastion built into the apiserver
- connects a user outside of the cluster to cluster IPs which otherwise might not be reachable
- runs in the apiserver processes
- client to proxy uses HTTPS (or http if apiserver so configured)
- proxy to target may use HTTP or HTTPS as chosen by proxy using available information
- can be used to reach a Node, Pod, or Service
- does load balancing when used to reach a Service

## 3. The [kube proxy](#):

- runs on each node
- proxies UDP and TCP
- does not understand HTTP
- provides load balancing
- is just used to reach services

## 4. A Proxy/Load-balancer in front of apiserver(s):

- existence and implementation varies from cluster to cluster (e.g. nginx)
- sits between all clients and one or more apiservers
- acts as load balancer if there are several apiservers.

## 5. Cloud Load Balancers on external services:

- are provided by some cloud providers (e.g. AWS ELB, Google Cloud Load Balancer)
- are created automatically when the Kubernetes service has type `LoadBalancer`
- use UDP/TCP only
- implementation varies by cloud provider.

Kubernetes users will typically not need to worry about anything other than the first two types. The cluster admin will typically ensure that the latter types are setup correctly.

# Feedback

Was this page helpful?

Yes No

Thanks for the feedback. If you have a specific, answerable question about how to use Kubernetes, ask it on [Stack Overflow](#). Open an issue in the GitHub repo if you want to [report a problem](#) or [suggest an improvement](#).

---

[Create an Issue](#) [Edit This Page](#)

Page last modified on January 01, 2020 at 2:03 AM PST by [fix kubectl proxy output for en \(#18242\)](#) ([Page History](#))

[Edit This Page](#)

# Configure Access to Multiple Clusters

This page shows how to configure access to multiple clusters by using configuration files. After your clusters, users, and contexts are defined in one or more configuration files, you can quickly switch between clusters by using the `kubectl config use-context` command.

**Note:** A file that is used to configure access to a cluster is sometimes called a *kubeconfig file*. This is a generic way of referring to configuration files. It does not mean that there is a file named kubeconfig.

- [Before you begin](#)
- [Define clusters, users, and contexts](#)
- [Create a second configuration file](#)
- [Set the KUBECONFIG environment variable](#)
- [Explore the \\$HOME/.kube directory](#)
- [Append \\$HOME/.kube/config to your KUBECONFIG environment variable](#)
- [Clean up](#)
- [What's next](#)

## Before you begin

You need to have a Kubernetes cluster, and the kubectl command-line tool must be configured to communicate with your cluster. 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`.

## Define clusters, users, and contexts

Suppose you have two clusters, one for development work and one for scratch work. In the development cluster, your frontend developers work in a namespace called `frontend`, and your storage developers work in a namespace called `storage`. In your scratch cluster, developers work in the default namespace, or they create auxiliary namespaces as they see fit. Access to the development cluster requires authentication by certificate. Access to the scratch cluster requires authentication by username and password.

Create a directory named `config-exercise`. In your `config-exercise` directory, create a file named `config-demo` with this content:

```
apiVersion: v1
kind: Config
preferences: {}

clusters:
- cluster:
  name: development
- cluster:
  name: scratch
```

```
users:
- name: developer
- name: experimenter

contexts:
- context:
  name: dev-frontend
- context:
  name: dev-storage
- context:
  name: exp-scratch
```

A configuration file describes clusters, users, and contexts. Your config-demo file has the framework to describe two clusters, two users, and three contexts.

Go to your config-exercise directory. Enter these commands to add cluster details to your configuration file:

```
kubectl config --kubeconfig=config-demo set-cluster development
--server=https://1.2.3.4 --certificate-authority=fake-ca-file
kubectl config --kubeconfig=config-demo set-cluster scratch --
server=https://5.6.7.8 --insecure-skip-tls-verify
```

Add user details to your configuration file:

```
kubectl config --kubeconfig=config-demo set-credentials
developer --client-certificate=fake-cert-file --client-key=fake-
key-seefile
kubectl config --kubeconfig=config-demo set-credentials
experimenter --username=exp --password=some-password
```

**Note:**

- To delete a user you can run `kubectl --kubeconfig=config-demo config unset users.<name>`
- To remove a cluster, you can run `kubectl --kubeconfig=config-demo config unset clusters.<name>`
- To remove a context, you can run `kubectl --kubeconfig=config-demo config unset contexts.<name>`

Add context details to your configuration file:

```
kubectl config --kubeconfig=config-demo set-context dev-frontend
--cluster=development --namespace=frontend --user=developer
kubectl config --kubeconfig=config-demo set-context dev-storage
--cluster=development --namespace=storage --user=developer
kubectl config --kubeconfig=config-demo set-context exp-scratch
--cluster=scratch --namespace=default --user=experimenter
```

Open your config-demo file to see the added details. As an alternative to opening the config-demo file, you can use the config view command.

```
kubectl config --kubeconfig=config-demo view
```

The output shows the two clusters, two users, and three contexts:

```
apiVersion: v1
clusters:
- cluster:
    certificate-authority: fake-ca-file
    server: https://1.2.3.4
    name: development
- cluster:
    insecure-skip-tls-verify: true
    server: https://5.6.7.8
    name: scratch
contexts:
- context:
    cluster: development
    namespace: frontend
    user: developer
    name: dev-frontend
- context:
    cluster: development
    namespace: storage
    user: developer
    name: dev-storage
- context:
    cluster: scratch
    namespace: default
    user: experimenter
    name: exp-scratch
current-context: ""
kind: Config
preferences: {}
users:
- name: developer
  user:
    client-certificate: fake-cert-file
    client-key: fake-key-file
- name: experimenter
  user:
    password: some-password
    username: exp
```

The `fake-ca-file`, `fake-cert-file` and `fake-key-file` above is the placeholders for the real path of the certification files. You need change these to the real path of certification files in your environment.

Some times you may want to use base64 encoded data here instead of the path of the certification files, then you need add the suffix `-data` to the keys. For example, `certificate-authority-data`, `client-certificate-data`, `client-key-data`.

Each context is a triple (cluster, user, namespace). For example, the `dev-frontend` context says, Use the credentials of the `developer` user to access the `frontend` namespace of the `development` cluster.

Set the current context:

```
kubectl config --kubeconfig=config-demo use-context dev-frontend
```

Now whenever you enter a `kubectl` command, the action will apply to the cluster, and namespace listed in the `dev-frontend` context. And the command will use the credentials of the user listed in the `dev-frontend` context.

To see only the configuration information associated with the current context, use the `--minify` flag.

```
kubectl config --kubeconfig=config-demo view --minify
```

The output shows configuration information associated with the `dev-frontend` context:

```
apiVersion: v1
clusters:
- cluster:
    certificate-authority: fake-ca-file
    server: https://1.2.3.4
    name: development
contexts:
- context:
    cluster: development
    namespace: frontend
    user: developer
    name: dev-frontend
current-context: dev-frontend
kind: Config
preferences: {}
users:
- name: developer
  user:
    client-certificate: fake-cert-file
    client-key: fake-key-file
```

Now suppose you want to work for a while in the `scratch` cluster.

Change the current context to `exp-scratch`:

```
kubectl config --kubeconfig=config-demo use-context exp-scratch
```

Now any `kubectl` command you give will apply to the default namespace of the `scratch` cluster. And the command will use the credentials of the user listed in the `exp-scratch` context.

View configuration associated with the new current context, `exp-scratch`.

```
kubectl config --kubeconfig=config-demo view --minify
```

Finally, suppose you want to work for a while in the storage namespace of the development cluster.

Change the current context to dev-storage:

```
kubectl config --kubeconfig=config-demo use-context dev-storage
```

View configuration associated with the new current context, dev-storage.

```
kubectl config --kubeconfig=config-demo view --minify
```

## Create a second configuration file

In your config-exercise directory, create a file named config-demo-2 with this content:

```
apiVersion: v1
kind: Config
preferences: {}

contexts:
- context:
    cluster: development
    namespace: ramp
    user: developer
    name: dev-ramp-up
```

The preceding configuration file defines a new context named dev-ramp-up.

## Set the KUBECONFIG environment variable

See whether you have an environment variable named KUBECONFIG. If so, save the current value of your KUBECONFIG environment variable, so you can restore it later. For example:

### Linux

```
export KUBECONFIG_SAVED=$KUBECONFIG
```

### Windows PowerShell

```
$Env:KUBECONFIG_SAVED=$Env:KUBECONFIG
```

The KUBECONFIG environment variable is a list of paths to configuration files. The list is colon-delimited for Linux and Mac, and semicolon-delimited for Windows. If you have a KUBECONFIG environment variable, familiarize yourself with the configuration files in the list.

Temporarily append two paths to your KUBECONFIG environment variable.  
For example:

## Linux

```
export KUBECONFIG=$KUBECONFIG:config-demo:config-demo-2
```

## Windows PowerShell

```
$Env:KUBECONFIG=( "config-demo;config-demo-2" )
```

In your config-exercise directory, enter this command:

```
kubectl config view
```

The output shows merged information from all the files listed in your KUBECONFIG environment variable. In particular, notice that the merged information has the dev-ramp-up context from the config-demo-2 file and the three contexts from the config-demo file:

```
contexts:
- context:
    cluster: development
    namespace: frontend
    user: developer
    name: dev-frontend
- context:
    cluster: development
    namespace: ramp
    user: developer
    name: dev-ramp-up
- context:
    cluster: development
    namespace: storage
    user: developer
    name: dev-storage
- context:
    cluster: scratch
    namespace: default
    user: experimenter
    name: exp-scratch
```

For more information about how kubeconfig files are merged, see [Organizing Cluster Access Using kubeconfig Files](#)

## Explore the \$HOME/.kube directory

If you already have a cluster, and you can use kubectl to interact with the cluster, then you probably have a file named config in the \$HOME/.kube directory.

Go to `$HOME/.kube`, and see what files are there. Typically, there is a file named `config`. There might also be other configuration files in this directory. Briefly familiarize yourself with the contents of these files.

## Append `$HOME/.kube/config` to your KUBECONFIG environment variable

If you have a `$HOME/.kube/config` file, and it's not already listed in your KUBECONFIG environment variable, append it to your KUBECONFIG environment variable now. For example:

### Linux

```
export KUBECONFIG=$KUBECONFIG:$HOME/.kube/config
```

### Windows Powershell

```
$Env:KUBECONFIG=($Env:KUBECONFIG;$HOME/.kube/config)
```

View configuration information merged from all the files that are now listed in your KUBECONFIG environment variable. In your config-exercise directory, enter:

```
kubectl config view
```

## Clean up

Return your KUBECONFIG environment variable to its original value. For example:

Linux:

```
export KUBECONFIG=$KUBECONFIG_SAVED
```

Windows PowerShell

```
$Env:KUBECONFIG=$Env:KUBECONFIG_SAVED
```

## What's next

- [Organizing Cluster Access Using kubeconfig Files](#)
- [kubectl config](#)

## Feedback

Was this page helpful?

Yes No

Thanks for the feedback. If you have a specific, answerable question about how to use Kubernetes, ask it on [Stack Overflow](#). Open an issue in the GitHub repo if you want to [report a problem](#) or [suggest an improvement](#).

---

[Create an Issue](#) [Edit This Page](#)

Page last modified on February 15, 2020 at 8:33 PM PST by [Issue with k8s.io/docs/tasks/access-application-cluster/configure-access-multiple-clusters/ \(#18560\)](#) ([Page History](#))

[Edit This Page](#)

# Use Port Forwarding to Access Applications in a Cluster

This page shows how to use `kubectl port-forward` to connect to a Redis server running in a Kubernetes cluster. This type of connection can be useful for database debugging.

- [Before you begin](#)
- [Creating Redis deployment and service](#)
- [Forward a local port to a port on the pod](#)
- [Discussion](#)
- [What's next](#)

## Before you begin

- You need to have a Kubernetes cluster, and the `kubectl` command-line tool must be configured to communicate with your cluster. 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`.

- Install [redis-cli](#).

## Creating Redis deployment and service

1. Create a Redis deployment:

```
kubectl apply -f https://k8s.io/examples/application/  
guestbook/redis-master-deployment.yaml
```

The output of a successful command verifies that the deployment was created:

```
deployment.apps/redis-master created
```

View the pod status to check that it is ready:

```
kubectl get pods
```

The output displays the pod created:

NAME	READY	STATUS
RESTARTS	AGE	
redis-master-765d459796-258hz	1/1	Running
0	50s	

View the deployment status:

```
kubectl get deployment
```

The output displays that the deployment was created:

NAME	READY	UP-TO-DATE	AVAILABLE	AGE
redis-master	1/1	1	1	55s

View the replicaset status using:

```
kubectl get rs
```

The output displays that the replicaset was created:

NAME	DESIRED	CURRENT	READY	AGE
redis-master-765d459796	1	1	1	1m

## 2. Create a Redis service:

```
kubectl apply -f https://k8s.io/examples/application/guestbook/redis-master-service.yaml
```

The output of a successful command verifies that the service was created:

```
service/redis-master created
```

Check the service created:

```
kubectl get svc | grep redis
```

The output displays the service created:

NAME	TYPE	CLUSTER-IP	EXTERNAL-IP
redis-master	ClusterIP	10.0.0.213	<none>
TCP 27s			6379/

## 3. Verify that the Redis server is running in the pod and listening on port 6379:

```
kubectl get pods redis-master-765d459796-258hz --template='{{(index .spec.containers 0).ports 0).containerPort}}\n'
```

The output displays the port:

```
6379
```

## Forward a local port to a port on the pod

1. `kubectl port-forward` allows using resource name, such as a pod name, to select a matching pod to port forward to since Kubernetes v1.10.

```
kubectl port-forward redis-master-765d459796-258hz 7000:6379
```

which is the same as

```
kubectl port-forward pods/redis-master-765d459796-258hz  
7000:6379
```

or

```
kubectl port-forward deployment/redis-master 7000:6379
```

or

```
kubectl port-forward rs/redis-master 7000:6379
```

or

```
kubectl port-forward svc/redis-master 7000:6379
```

Any of the above commands works. The output is similar to this:

```
I0710 14:43:38.274550    3655 portforward.go:225] Forwarding  
from 127.0.0.1:7000 -> 6379  
I0710 14:43:38.274797    3655 portforward.go:225] Forwarding  
from [::1]:7000 -> 6379
```

2. Start the Redis command line interface:

```
redis-cli -p 7000
```

3. At the Redis command line prompt, enter the `ping` command:

```
127.0.0.1:7000>ping
```

A successful ping request returns PONG.

## Discussion

Connections made to local port 7000 are forwarded to port 6379 of the pod that is running the Redis server. With this connection in place you can use your local workstation to debug the database that is running in the pod.

**Warning:** Due to known limitations, port forward today only works for TCP protocol. The support to UDP protocol is being tracked in [issue 47862](#).

# What's next

Learn more about [kubectl port-forward](#).

## Feedback

Was this page helpful?

Yes No

Thanks for the feedback. If you have a specific, answerable question about how to use Kubernetes, ask it on [Stack Overflow](#). Open an issue in the GitHub repo if you want to [report a problem](#) or [suggest an improvement](#).

---

Page last modified on November 02, 2019 at 12:31 PM PST by [change](#)  
[kubectl get deployment output \(#17351\)](#) ([Page History](#))

[Edit This Page](#)

# Use a Service to Access an Application in a Cluster

This page shows how to create a Kubernetes Service object that external clients can use to access an application running in a cluster. The Service provides load balancing for an application that has two running instances.

- [Objectives](#)
- [Before you begin](#)
- [Creating a service for an application running in two pods](#)
- [Using a service configuration file](#)
- [Cleaning up](#)
- [What's next](#)

## Objectives

- Run two instances of a Hello World application.
- Create a Service object that exposes a node port.
- Use the Service object to access the running application.

## Before you begin

You need to have a Kubernetes cluster, and the `kubectl` command-line tool must be configured to communicate with your cluster. 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`.

## Creating a service for an application running in two pods

Here is the configuration file for the application Deployment:

## [service/access/hello-application.yaml](#)

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: hello-world
spec:
  selector:
    matchLabels:
      run: load-balancer-example
  replicas: 2
  template:
    metadata:
      labels:
        run: load-balancer-example
    spec:
      containers:
        - name: hello-world
          image: gcr.io/google-samples/node-hello:1.0
          ports:
            - containerPort: 8080
              protocol: TCP
```

1. Run a Hello World application in your cluster: Create the application Deployment using the file above:

```
kubectl apply -f https://k8s.io/examples/service/access/
hello-application.yaml
```

The preceding command creates a [Deployment](#) object and an associated [ReplicaSet](#) object. The ReplicaSet has two [Pods](#), each of which runs the Hello World application.

2. Display information about the Deployment:

```
kubectl get deployments hello-world
kubectl describe deployments hello-world
```

3. Display information about your ReplicaSet objects:

```
kubectl get replicsets
kubectl describe replicsets
```

4. Create a Service object that exposes the deployment:

```
kubectl expose deployment hello-world --type=NodePort --
name=example-service
```

5. Display information about the Service:

```
kubectl describe services example-service
```

The output is similar to this:

Name:	example-service
Namespace:	default
Labels:	run=load-balancer-example
Annotations:	<none>
Selector:	run=load-balancer-example
Type:	NodePort
IP:	10.32.0.16
Port:	<unset> 8080/TCP
TargetPort:	8080/TCP
NodePort:	<unset> 31496/TCP
Endpoints:	10.200.1.4:8080,10.200.2.5:8080
Session Affinity:	None
Events:	<none>

Make a note of the NodePort value for the service. For example, in the preceding output, the NodePort value is 31496.

6. List the pods that are running the Hello World application:

```
kubectl get pods --selector="run=load-balancer-example" --output=wide
```

The output is similar to this:

NAME	READY	STATUS	...
IP	NODE		
hello-world-2895499144-bsbk5	1/1	Running	... 10.
200.1.4 worker1			
hello-world-2895499144-m1pwt	1/1	Running	... 10.
200.2.5 worker2			

7. Get the public IP address of one of your nodes that is running a Hello World pod. How you get this address depends on how you set up your cluster. For example, if you are using Minikube, you can see the node address by running `kubectl cluster-info`. If you are using Google Compute Engine instances, you can use the `gcloud compute instances list` command to see the public addresses of your nodes.
8. On your chosen node, create a firewall rule that allows TCP traffic on your node port. For example, if your Service has a NodePort value of 31568, create a firewall rule that allows TCP traffic on port 31568. Different cloud providers offer different ways of configuring firewall rules.
9. Use the node address and node port to access the Hello World application:

```
curl http://<public-node-ip>:<node-port>
```

where <public-node-ip> is the public IP address of your node, and <node-port> is the NodePort value for your service. The response to a successful request is a hello message:

```
Hello Kubernetes!
```

## Using a service configuration file

As an alternative to using `kubectl expose`, you can use a [service configuration file](#) to create a Service.

## Cleaning up

To delete the Service, enter this command:

```
kubectl delete services example-service
```

To delete the Deployment, the ReplicaSet, and the Pods that are running the Hello World application, enter this command:

```
kubectl delete deployment hello-world
```

## What's next

Learn more about [connecting applications with services](#).

## Feedback

Was this page helpful?

Yes No

Thanks for the feedback. If you have a specific, answerable question about how to use Kubernetes, ask it on [Stack Overflow](#). Open an issue in the GitHub repo if you want to [report a problem](#) or [suggest an improvement](#).

---

[Create an Issue](#) [Edit This Page](#)

Page last modified on June 27, 2019 at 2:17 AM PST by [update tutorial not to use 'kubectl run' for creating deployment \(#14980\)](#) ([Page History](#))

[Edit This Page](#)

# Connect a Front End to a Back End Using a Service

This task shows how to create a frontend and a backend microservice. The backend microservice is a hello greeter. The frontend and backend are connected using a Kubernetes [ServiceA way to expose an application running on a set of Pods as a network service.](#) object.

- [Objectives](#)

- [Before you begin](#)
- [What's next](#)

## Objectives

- Create and run a microservice using a [Deployment](#)  
[An API object that manages a replicated application.](#) object.
- Route traffic to the backend using a frontend.
- Use a Service object to connect the frontend application to the backend application.

## Before you begin

- You need to have a Kubernetes cluster, and the kubectl command-line tool must be configured to communicate with your cluster. 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 task uses [Services with external load balancers](#), which require a supported environment. If your environment does not support this, you can use a Service of type [NodePort](#) instead.

## Creating the backend using a Deployment

The backend is a simple hello greeter microservice. Here is the configuration file for the backend Deployment:

## [service/access/hello.yaml](#)

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: hello
spec:
  selector:
    matchLabels:
      app: hello
      tier: backend
      track: stable
  replicas: 7
  template:
    metadata:
      labels:
        app: hello
        tier: backend
        track: stable
    spec:
      containers:
        - name: hello
          image: "gcr.io/google-samples/hello-go-gke:1.0"
          ports:
            - name: http
              containerPort: 80
```

Create the backend Deployment:

```
kubectl apply -f https://k8s.io/examples/service/access/
hello.yaml
```

View information about the backend Deployment:

```
kubectl describe deployment hello
```

The output is similar to this:

```
Name:                      hello
Namespace:                  default
CreationTimestamp:           Mon, 24 Oct 2016 14:21:02 -0700
Labels:                     app=hello
                            tier=backend
                            track=stable
Annotations:                deployment.kubernetes.io/
revision=1
Selector:                   app=hello,tier=backend,track=stable
Replicas:                   7 desired | 7 updated | 7 total
                            | 7 available | 0 unavailable
StrategyType:               RollingUpdate
```

```

MinReadySeconds:          0
RollingUpdateStrategy:   1 max unavailable, 1 max surge
Pod Template:
  Labels:      app=hello
                tier=backend
                track=stable
  Containers:
    hello:
      Image:      "gcr.io/google-samples/hello-go-gke:1.0"
      Port:       80/TCP
      Environment: <none>
      Mounts:     <none>
      Volumes:    <none>
  Conditions:
    Type  Status  Reason
    ----  ----- 
    Available  True  MinimumReplicasAvailable
    Progressing  True  NewReplicaSetAvailable
OldReplicaSets:           <none>
NewReplicaSet:            hello-3621623197 (7/7 replicas
created)
Events:
...

```

## Creating the backend Service object

The key to connecting a frontend to a backend is the backend Service. A Service creates a persistent IP address and DNS name entry so that the backend microservice can always be reached. A Service uses [selectors](#) to filter a list of resources based on labels. to find the Pods that it routes traffic to.

First, explore the Service configuration file:

### [service/access/hello-service.yaml](#)

```

apiVersion: v1
kind: Service
metadata:
  name: hello
spec:
  selector:
    app: hello
    tier: backend
  ports:
  - protocol: TCP
    port: 80
    targetPort: http

```

In the configuration file, you can see that the Service routes traffic to Pods that have the labels `app: hello` and `tier: backend`.

Create the `hello` Service:

```
kubectl apply -f https://k8s.io/examples/service/access/hello-service.yaml
```

At this point, you have a backend Deployment running, and you have a Service that can route traffic to it.

## Creating the frontend

Now that you have your backend, you can create a frontend that connects to the backend. The frontend connects to the backend worker Pods by using the DNS name given to the backend Service. The DNS name is "hello", which is the value of the `name` field in the preceding Service configuration file.

The Pods in the frontend Deployment run an nginx image that is configured to find the hello backend Service. Here is the nginx configuration file:

### [service/access/frontend.conf](#)

```
upstream hello {
    server hello;
}

server {
    listen 80;

    location / {
        proxy_pass http://hello;
    }
}
```

Similar to the backend, the frontend has a Deployment and a Service. The configuration for the Service has `type: LoadBalancer`, which means that the Service uses the default load balancer of your cloud provider.

## [service/access/frontend.yaml](#)

```
apiVersion: v1
kind: Service
metadata:
  name: frontend
spec:
  selector:
    app: hello
    tier: frontend
  ports:
  - protocol: "TCP"
    port: 80
    targetPort: 80
  type: LoadBalancer
---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: frontend
spec:
  selector:
    matchLabels:
      app: hello
      tier: frontend
      track: stable
  replicas: 1
  template:
    metadata:
      labels:
        app: hello
        tier: frontend
        track: stable
    spec:
      containers:
      - name: nginx
        image: "gcr.io/google-samples/hello-frontend:1.0"
        lifecycle:
          preStop:
            exec:
              command: ["/usr/sbin/nginx", "-s", "quit"]
```

Create the frontend Deployment and Service:

```
kubectl apply -f https://k8s.io/examples/service/access/
frontend.yaml
```

The output verifies that both resources were created:

```
deployment.apps/frontend created
service/frontend created
```

**Note:** The nginx configuration is baked into the [container image](#). A better way to do this would be to use a [ConfigMap](#), so that you can change the configuration more easily.

## Interact with the frontend Service

Once you've created a Service of type LoadBalancer, you can use this command to find the external IP:

```
kubectl get service frontend --watch
```

This displays the configuration for the frontend Service and watches for changes. Initially, the external IP is listed as <pending>:

NAME	TYPE	CLUSTER-IP	EXTERNAL-IP	PORT(S)
AGE				
frontend	LoadBalancer	10.51.252.116	<pending>	80/TCP

As soon as an external IP is provisioned, however, the configuration updates to include the new IP under the EXTERNAL-IP heading:

NAME	TYPE	CLUSTER-IP	EXTERNAL-IP	PORT(S)
PORT(S)	AGE			
frontend	LoadBalancer	10.51.252.116	XXX.XXX.XXX.XXX	80/TCP

That IP can now be used to interact with the frontend service from outside the cluster.

## Send traffic through the frontend

The frontend and backends are now connected. You can hit the endpoint by using the curl command on the external IP of your frontend Service.

```
curl http://${EXTERNAL_IP} # replace this with the EXTERNAL-IP  
you saw earlier
```

The output shows the message generated by the backend:

```
{"message": "Hello"}
```

## What's next

- Learn more about [Services](#)
- Learn more about [ConfigMaps](#)

## Feedback

Was this page helpful?

Yes No

Thanks for the feedback. If you have a specific, answerable question about how to use Kubernetes, ask it on [Stack Overflow](#). Open an issue in the GitHub repo if you want to [report a problem](#) or [suggest an improvement](#).

---

[Create an Issue](#) [Edit This Page](#)

Page last modified on March 25, 2019 at 5:06 PM PST by [Official 1.14](#)  
[Release Docs \(#13174\)](#) ([Page History](#))

[Edit This Page](#)

## Create an External Load Balancer

This page shows how to create an External Load Balancer.

**Note:** This feature is only available for cloud providers or environments which support external load balancers.

When creating a service, you have the option of automatically creating a cloud network load balancer. This provides an externally-accessible IP address that sends traffic to the correct port on your cluster nodes *provided your cluster runs in a supported environment and is configured with the correct cloud load balancer provider package*.

For information on provisioning and using an Ingress resource that can give services externally-reachable URLs, load balance the traffic, terminate SSL etc., please check the [Ingress](#) documentation.

- [Before you begin](#)
- [Configuration file](#)
- [Using kubectl](#)
- [Finding your IP address](#)
- [Preserving the client source IP](#)
- [Garbage Collecting Load Balancers](#)
- [External Load Balancer Providers](#)
- [Caveats and Limitations when preserving source IPs](#)

## Before you begin

- You need to have a Kubernetes cluster, and the kubectl command-line tool must be configured to communicate with your cluster. 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`.

## Configuration file

To create an external load balancer, add the following line to your [service configuration file](#):

```
type: LoadBalancer
```

Your configuration file might look like:

```
apiVersion: v1
kind: Service
metadata:
  name: example-service
spec:
  selector:
    app: example
  ports:
```

```
- port: 8765
  targetPort: 9376
type: LoadBalancer
```

## Using kubectl

You can alternatively create the service with the `kubectl expose` command and its `--type=LoadBalancer` flag:

```
kubectl expose rc example --port=8765 --target-port=9376 \
--name=example-service --type=LoadBalancer
```

This command creates a new service using the same selectors as the referenced resource (in the case of the example above, a replication controller named `example`).

For more information, including optional flags, refer to the [kubectl expose reference](#).

## Finding your IP address

You can find the IP address created for your service by getting the service information through `kubectl`:

```
kubectl describe services example-service
```

which should produce output like this:

Name:	example-service
Namespace:	default
Labels:	<none>
Annotations:	<none>
Selector:	app=example
Type:	LoadBalancer
IP:	10.67.252.103
LoadBalancer Ingress:	192.0.2.89
Port:	<unnamed> 80/TCP
NodePort:	<unnamed> 32445/TCP
Endpoints:	10. 64.0.4:80,10.64.1.5:80,10.64.2.4:80
Session Affinity:	None
Events:	<none>

The IP address is listed next to `LoadBalancer Ingress`.

**Note:** If you are running your service on Minikube, you can find the assigned IP address and port with:

```
minikube service example-service --url
```

# Preserving the client source IP

Due to the implementation of this feature, the source IP seen in the target container is *not the original source IP* of the client. To enable preservation of the client IP, the following fields can be configured in the service spec (supported in GCE/Google Kubernetes Engine environments):

- `service.spec.externalTrafficPolicy` - denotes if this Service desires to route external traffic to node-local or cluster-wide endpoints. There are two available options: Cluster (default) and Local. Cluster obscures the client source IP and may cause a second hop to another node, but should have good overall load-spreading. Local preserves the client source IP and avoids a second hop for LoadBalancer and NodePort type services, but risks potentially imbalanced traffic spreading.
- `service.spec.healthCheckNodePort` - specifies the health check node port (numeric port number) for the service. If `healthCheckNodePort` isn't specified, the service controller allocates a port from your cluster's NodePort range. You can configure that range by setting an API server command line option, `--service-node-port-range`. It will use the user-specified `healthCheckNodePort` value if specified by the client. It only has an effect when `type` is set to LoadBalancer and `externalTrafficPolicy` is set to Local.

Setting `externalTrafficPolicy` to Local in the Service configuration file activates this feature.

```
apiVersion: v1
kind: Service
metadata:
  name: example-service
spec:
  selector:
    app: example
  ports:
    - port: 8765
      targetPort: 9376
  externalTrafficPolicy: Local
  type: LoadBalancer
```

# Garbage Collecting Load Balancers

**FEATURE STATE:** Kubernetes v1.17 [stable](#)

This feature is *stable*, meaning:

[Edit This Page](#)

# Configure Your Cloud Provider's Firewalls

Many cloud providers (e.g. Google Compute Engine) define firewalls that help prevent inadvertent exposure to the internet. When exposing a service to the external world, you may need to open up one or more ports in these firewalls to serve traffic. This document describes this process, as well as any provider specific details that may be necessary.

- [Before you begin](#)
- [Restrict Access For LoadBalancer Service](#)
- [Google Compute Engine](#)

## Before you begin

You need to have a Kubernetes cluster, and the `kubectl` command-line tool must be configured to communicate with your cluster. 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`.

## Restrict Access For LoadBalancer Service

When using a Service with `spec.type: LoadBalancer`, you can specify the IP ranges that are allowed to access the load balancer by using `spec.loadBalancerSourceRanges`. This field takes a list of IP CIDR ranges, which Kubernetes will use to configure firewall exceptions. This feature is currently supported on Google Compute Engine, Google Kubernetes Engine, AWS Elastic Kubernetes Service, Azure Kubernetes Service, and IBM Cloud Kubernetes Service. This field will be ignored if the cloud provider does not support the feature.

Assuming 10.0.0.0/8 is the internal subnet. In the following example, a load balancer will be created that is only accessible to cluster internal IPs. This will not allow clients from outside of your Kubernetes cluster to access the load balancer.

```
apiVersion: v1
kind: Service
metadata:
  name: myapp
spec:
  ports:
  - port: 8765
    targetPort: 9376
```

```
selector:
  app: example
type: LoadBalancer
loadBalancerSourceRanges:
- 10.0.0.0/8
```

In the following example, a load balancer will be created that is only accessible to clients with IP addresses from 130.211.204.1 and 130.211.204.2.

```
apiVersion: v1
kind: Service
metadata:
  name: myapp
spec:
  ports:
    - port: 8765
      targetPort: 9376
  selector:
    app: example
  type: LoadBalancer
  loadBalancerSourceRanges:
    - 130.211.204.1/32
    - 130.211.204.2/32
```

## Google Compute Engine

When using a Service with `spec.type: LoadBalancer`, the firewall will be opened automatically. When using `spec.type: NodePort`, however, the firewall is *not* opened by default.

Google Compute Engine firewalls are documented [elsewhere](#).

You can add a firewall with the `gcloud` command line tool:

```
gcloud compute firewall-rules create my-rule --allow=tcp:<port>
```

### Note:

GCE firewalls are defined per-vm, rather than per-ip address. This means that when you open a firewall for a service's ports, anything that serves on that port on that VM's host IP address may potentially serve traffic. Note that this is not a problem for other Kubernetes services, as they listen on IP addresses that are different than the host node's external IP address.

Consider:

- You create a Service with an external load balancer (IP Address 1.2.3.4) and port 80

- You open the firewall for port 80 for all nodes in your cluster, so that the external Service actually can deliver packets to your Service
- You start an nginx server, running on port 80 on the host virtual machine (IP Address 2.3.4.5). This nginx is also exposed to the internet on the VM's external IP address.

Consequently, please be careful when opening firewalls in Google Compute Engine or Google Kubernetes Engine. You may accidentally be exposing other services to the wilds of the internet.

## Feedback

Was this page helpful?

Yes No

Thanks for the feedback. If you have a specific, answerable question about how to use Kubernetes, ask it on [Stack Overflow](#). Open an issue in the GitHub repo if you want to [report a problem](#) or [suggest an improvement](#).

---

[Create an Issue](#) [Edit This Page](#)

Page last modified on January 23, 2019 at 10:55 AM PST by [Update](#)  
[configure-cloud-provider-firewall.md \(#12304\)](#) ([Page History](#))

[Edit This Page](#)

# List All Container Images Running in a Cluster

This page shows how to use kubectl to list all of the Container images for Pods running in a cluster.

- [Before you begin](#)
- [List all Containers in all namespaces](#)
- [List Containers by Pod](#)

- [List Containers filtering by Pod label](#)
- [List Containers filtering by Pod namespace](#)
- [List Containers using a go-template instead of jsonpath](#)
- [What's next](#)

## Before you begin

You need to have a Kubernetes cluster, and the `kubectl` command-line tool must be configured to communicate with your cluster. 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`.

In this exercise you will use `kubectl` to fetch all of the Pods running in a cluster, and format the output to pull out the list of Containers for each.

## List all Containers in all namespaces

- Fetch all Pods in all namespaces using `kubectl get pods --all-namespaces`
- Format the output to include only the list of Container image names using `-o jsonpath={..image}`. This will recursively parse out the `image` field from the returned json.
  - See the [jsonpath reference](#) for further information on how to use jsonpath.
- Format the output using standard tools: `tr`, `sort`, `uniq`
  - Use `tr` to replace spaces with newlines
  - Use `sort` to sort the results
  - Use `uniq` to aggregate image counts

```
kubectl get pods --all-namespaces -o jsonpath="{..image}"
" |\
tr -s '[[:space:]]' '\n' |\
sort |\
uniq -c
```

The above command will recursively return all fields named `image` for all items returned.

As an alternative, it is possible to use the absolute path to the `image` field within the Pod. This ensures the correct field is retrieved even when the field name is repeated, e.g. many fields are called `name` within a given item:

```
kubectl get pods --all-namespaces -o jsonpath=".items[*].spec.co
ntainers[*].image"
```

The jsonpath is interpreted as follows:

- `.items[*]`: for each returned value
- `.spec`: get the spec
- `.containers[*]`: for each container
- `.image`: get the image

**Note:** When fetching a single Pod by name, e.g. `kubectl get pod nginx`, the `.items[*]` portion of the path should be omitted because a single Pod is returned instead of a list of items.

## List Containers by Pod

The formatting can be controlled further by using the `range` operation to iterate over elements individually.

```
kubectl get pods --all-namespaces -o=jsonpath='{range .items[*]}  
{"\n"}{.metadata.name}{":\t"}{range .spec.containers[*]}{.image}  
{", "}{end}{end}' |  
sort
```

## List Containers filtering by Pod label

To target only Pods matching a specific label, use the `-l` flag. The following matches only Pods with labels matching `app=nginx`.

```
kubectl get pods --all-namespaces -o=jsonpath=".image" -l  
app=nginx
```

## List Containers filtering by Pod namespace

To target only pods in a specific namespace, use the `namespace` flag. The following matches only Pods in the `kube-system` namespace.

```
kubectl get pods --namespace kube-system -o jsonpath=".image"
```

## List Containers using a go-template instead of jsonpath

As an alternative to jsonpath, Kubectl supports using [go-templates](#) for formatting the output:

```
kubectl get pods --all-namespaces -o go-template --template="{{range  
.items}}{{range .spec.containers}}{{.image}} {{end}}{{end}}"
```

# What's next

## Reference

- [Jsonpath](#) reference guide
- [Go template](#) reference guide

## Feedback

Was this page helpful?

Yes No

Thanks for the feedback. If you have a specific, answerable question about how to use Kubernetes, ask it on [Stack Overflow](#). Open an issue in the GitHub repo if you want to [report a problem](#) or [suggest an improvement](#).

---

[Create an Issue](#) [Edit This Page](#)

Page last modified on November 06, 2018 at 11:33 AM PST by [Add admonition type to shortcode \(#9482\)](#) ([Page History](#))

[Edit This Page](#)

# Set up Ingress on Minikube with the NGINX Ingress Controller

An [Ingress](#) is an API object that defines rules which allow external access to services in a cluster. An [Ingress controller](#) fulfills the rules set in the Ingress.

**Caution:** For the Ingress resource to work, the cluster **must** also have an Ingress controller running.

This page shows you how to set up a simple Ingress which routes requests to Service web or web2 depending on the HTTP URI.

- [Before you begin](#)
- [Create a Minikube cluster](#)
- [Enable the Ingress controller](#)
- [Deploy a hello, world app](#)
- [Create an Ingress resource](#)
- [Create Second Deployment](#)
- [Edit Ingress](#)
- [Test Your Ingress](#)
- [What's next](#)

## Before you begin

You need to have a Kubernetes cluster, and the kubectl command-line tool must be configured to communicate with your cluster. 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 Minikube cluster

1. Click **Launch Terminal**
-

[Create an Issue](#) [Edit This Page](#)

Page last modified on October 23, 2019 at 8:24 AM PST by [Fixes path in first ingress resource yaml \(#17017\)](#) ([Page History](#))

[Edit This Page](#)

# Communicate Between Containers in the Same Pod Using a Shared Volume

This page shows how to use a Volume to communicate between two Containers running in the same Pod. See also how to allow processes to communicate by [sharing process namespace](#) between containers.

- [Before you begin](#)
- [Creating a Pod that runs two Containers](#)

- [Discussion](#)
- [What's next](#)

## Before you begin

You need to have a Kubernetes cluster, and the kubectl command-line tool must be configured to communicate with your cluster. 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`.

## Creating a Pod that runs two Containers

In this exercise, you create a Pod that runs two Containers. The two containers share a Volume that they can use to communicate. Here is the configuration file for the Pod:

### [pods/two-container-pod.yaml](#)

```
apiVersion: v1
kind: Pod
metadata:
  name: two-containers
spec:
  restartPolicy: Never
  volumes:
  - name: shared-data
    emptyDir: {}
  containers:
  - name: nginx-container
    image: nginx
    volumeMounts:
    - name: shared-data
      mountPath: /usr/share/nginx/html
  - name: debian-container
    image: debian
    volumeMounts:
    - name: shared-data
      mountPath: /pod-data
    command: ["/bin/sh"]
    args: ["-c", "echo Hello from the debian container > /pod-data/index.html"]
```

In the configuration file, you can see that the Pod has a Volume named `shared-data`.

The first container listed in the configuration file runs an nginx server. The mount path for the shared Volume is `/usr/share/nginx/html`. The second container is based on the debian image, and has a mount path of `/pod-data`. The second container runs the following command and then terminates.

```
echo Hello from the debian container > /pod-data/index.html
```

Notice that the second container writes the `index.html` file in the root directory of the nginx server.

Create the Pod and the two Containers:

```
kubectl apply -f https://k8s.io/examples/pods/two-container-pod.yaml
```

View information about the Pod and the Containers:

```
kubectl get pod two-containers --output=yaml
```

Here is a portion of the output:

```
apiVersion: v1
kind: Pod
metadata:
  ...
  name: two-containers
  namespace: default
  ...
spec:
  ...
  containerStatuses:
    - containerID: docker://c1d8abd1 ...
      image: debian
      ...
      lastState:
        terminated:
        ...
        name: debian-container
        ...
    - containerID: docker://96c1ff2c5bb ...
      image: nginx
      ...
      name: nginx-container
      ...
      state:
        running:
        ...

```

You can see that the debian Container has terminated, and the nginx Container is still running.

Get a shell to nginx Container:

```
kubectl exec -it two-containers -c nginx-container -- /bin/bash
```

In your shell, verify that nginx is running:

```
root@two-containers:/# apt-get update
root@two-containers:/# apt-get install curl procps
root@two-containers:/# ps aux
```

The output is similar to this:

USER	PID	...	STAT	START	TIME	COMMAND
root	1	...	Ss	21:12	0:00	nginx: master process
nginx	-g	daemon	off;			

Recall that the debian Container created the `index.html` file in the nginx root directory. Use `curl` to send a GET request to the nginx server:

```
root@two-containers:/# curl localhost
```

The output shows that nginx serves a web page written by the debian container:

```
Hello from the debian container
```

## Discussion

The primary reason that Pods can have multiple containers is to support helper applications that assist a primary application. Typical examples of helper applications are data pullers, data pushers, and proxies. Helper and primary applications often need to communicate with each other. Typically this is done through a shared filesystem, as shown in this exercise, or through the loopback network interface, `localhost`. An example of this pattern is a web server along with a helper program that polls a Git repository for new updates.

The Volume in this exercise provides a way for Containers to communicate during the life of the Pod. If the Pod is deleted and recreated, any data stored in the shared Volume is lost.

## What's next

- Learn more about [patterns for composite containers](#).
- Learn about [composite containers for modular architecture](#).
- See [Configuring a Pod to Use a Volume for Storage](#).
- See [Configure a Pod to share process namespace between containers in a Pod](#)
- See [Volume](#).
- See [Pod](#).

## Feedback

Was this page helpful?

Yes No

Thanks for the feedback. If you have a specific, answerable question about how to use Kubernetes, ask it on [Stack Overflow](#). Open an issue in the GitHub repo if you want to [report a problem](#) or [suggest an improvement](#).

---

[Create an Issue](#) [Edit This Page](#)

Page last modified on June 11, 2019 at 9:52 AM PST by [Add Ref to shared process namespaces \(#14194\)](#) ([Page History](#))

[Edit This Page](#)

# Configure DNS for a Cluster

Kubernetes offers a DNS cluster addon, which most of the supported environments enable by default. In Kubernetes version 1.11 and later, CoreDNS is recommended and is installed by default with kubeadm.

For more information on how to configure CoreDNS for a Kubernetes cluster, see the [Customizing DNS Service](#). An example demonstrating how to use Kubernetes DNS with kube-dns, see the [Kubernetes DNS sample plugin](#).

# Feedback

Was this page helpful?

Yes No

Thanks for the feedback. If you have a specific, answerable question about how to use Kubernetes, ask it on [Stack Overflow](#). Open an issue in the GitHub repo if you want to [report a problem](#) or [suggest an improvement](#).

---

[Create an Issue](#) [Edit This Page](#)

Page last modified on November 30, 2019 at 8:59 AM PST by  
[Tasks/Configure DNS for a Cluster: add some descriptions about CoreDNS \(#17879\)](#) ([Page History](#))

[Edit This Page](#)

# Application Introspection and Debugging

Once your application is running, you'll inevitably need to debug problems with it. Earlier we described how you can use `kubectl get pods` to retrieve simple status information about your pods. But there are a number of ways to get even more information about your application.

- [Using `kubectl describe pod` to fetch details about pods](#)
- [Example: debugging Pending Pods](#)
- [Example: debugging a down/unreachable node](#)
- [What's next](#)

## Using `kubectl describe pod` to fetch details about pods

For this example we'll use a Deployment to create two pods, similar to the earlier example.

### [application/nginx-with-request.yaml](#)

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deployment
spec:
  selector:
    matchLabels:
      app: nginx
  replicas: 2
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
        - name: nginx
          image: nginx
          resources:
            limits:
              memory: "128Mi"
              cpu: "500m"
          ports:
            - containerPort: 80
```

Create deployment by running following command:

```
kubectl apply -f https://k8s.io/examples/application/nginx-with-request.yaml
```

```
deployment.apps/nginx-deployment created
```

Check pod status by following command:

```
kubectl get pods
```

NAME	READY	STATUS
RESTARTS	AGE	
nginx-deployment-1006230814-6winp0	1/1	Running 11s
nginx-deployment-1006230814-fmgu30	1/1	Running 11s

We can retrieve a lot more information about each of these pods using `kubectl describe pod`. For example:

```
kubectl describe pod nginx-deployment-1006230814-6winp
```

```
Name:      nginx-deployment-1006230814-6winp
Namespace: default
Node:      kubernetes-node-wul5/10.240.0.9
Start Time: Thu, 24 Mar 2016 01:39:49 +0000
Labels:    app=nginx,pod-template-hash=1006230814
Annotations:  kubernetes.io/created-
by={"kind":"SerializedReference","apiVersion":"v1","reference":>{"kind":"ReplicaSet","namespace":"default","name":"nginx-deployment-1956810328","uid":"14e607e7-8ba1-11e7-b5cb-fa16"...
Status:    Running
IP:      10.244.0.6
Controllers:ReplicaSet/nginx-deployment-1006230814
Containers:
  nginx:
    Container ID: docker://90315cc9f513c724e9957a4788d3e625a078de84750f244a40f97ae355eb1149
    Image:      nginx
    Image ID:   docker://6f62f48c4e55d700cf3eb1b5e33fa051802986b77b874cc351cce539e5163707
    Port:      80/TCP
    QoS Tier:
      cpu:  Guaranteed
      memory:  Guaranteed
    Limits:
      cpu:  500m
      memory:  128Mi
    Requests:
      memory:  128Mi
      cpu:  500m
```

```

State:      Running
Started:    Thu, 24 Mar 2016 01:39:51 +0000
Ready:      True
Restart Count: 0
Environment: <none>
Mounts:
  /var/run/secrets/kubernetes.io/serviceaccount from
  default-token-5kdvl (ro)
Conditions:
  Type        Status
  Initialized  True
  Ready       True
  PodScheduled  True
Volumes:
  default-token-4bcibi:
    Type:     Secret (a volume populated by a Secret)
    SecretName: default-token-4bcibi
    Optional:  false
  QoS Class:  Guaranteed
  Node-Selectors: <none>
  Tolerations: <none>
Events:
  FirstSeen  LastSeenCount  From
  SubobjectPath   Type  Reason      Message
  -----
  -          -          -
  54s        54s        1  {default-
scheduler }           Normal      Scheduled
Successfully assigned nginx-deployment-1006230814-6winp to
kubernetes-node-wul5
  54s        54s        1  {kubelet kubernetes-node-wul5}
spec.containers{nginx}  Normal      Pulling      pulling
image "nginx"
  53s        53s        1  {kubelet kubernetes-node-wul5}
spec.containers{nginx}  Normal      Pulled
Successfully pulled image "nginx"
  53s        53s        1  {kubelet kubernetes-node-wul5}
spec.containers{nginx}  Normal      Created      Created
container with docker id 90315cc9f513
  53s        53s        1  {kubelet kubernetes-node-wul5}
spec.containers{nginx}  Normal      Started      Started
container with docker id 90315cc9f513

```

Here you can see configuration information about the container(s) and Pod (labels, resource requirements, etc.), as well as status information about the container(s) and Pod (state, readiness, restart count, events, etc.).

The container state is one of Waiting, Running, or Terminated. Depending on the state, additional information will be provided - here

you can see that for a container in Running state, the system tells you when the container started.

Ready tells you whether the container passed its last readiness probe. (In this case, the container does not have a readiness probe configured; the container is assumed to be ready if no readiness probe is configured.)

Restart Count tells you how many times the container has been restarted; this information can be useful for detecting crash loops in containers that are configured with a restart policy of ‘always’.

Currently the only Condition associated with a Pod is the binary Ready condition, which indicates that the pod is able to service requests and should be added to the load balancing pools of all matching services.

Lastly, you see a log of recent events related to your Pod. The system compresses multiple identical events by indicating the first and last time it was seen and the number of times it was seen. “From” indicates the component that is logging the event, “SubobjectPath” tells you which object (e.g. container within the pod) is being referred to, and “Reason” and “Message” tell you what happened.

## Example: debugging Pending Pods

A common scenario that you can detect using events is when you've created a Pod that won't fit on any node. For example, the Pod might request more resources than are free on any node, or it might specify a label selector that doesn't match any nodes. Let's say we created the previous Deployment with 5 replicas (instead of 2) and requesting 600 millicores instead of 500, on a four-node cluster where each (virtual) machine has 1 CPU. In that case one of the Pods will not be able to schedule. (Note that because of the cluster addon pods such as fluentd, skydns, etc., that run on each node, if we requested 1000 millicores then none of the Pods would be able to schedule.)

```
kubectl get pods
```

NAME	READY	STATUS
nginx-deployment-1006230814-6winp	1/1	Running
nginx-deployment-1006230814-fmgu3	1/1	Running
nginx-deployment-1370807587-6ekbw	1/1	Running
nginx-deployment-1370807587-fg172	0/1	Pending
nginx-deployment-1370807587-fz9sd	0/1	Pending

To find out why the nginx-deployment-1370807587-fz9sd pod is not running, we can use `kubectl describe pod` on the pending Pod and look at its events:

```
kubectl describe pod nginx-deployment-1370807587-fz9sd
```

```
Name:      nginx-deployment-1370807587-fz9sd
Namespace: default
Node:      /
Labels:    app=nginx, pod-template-hash=1370807587
Status:    Pending
IP:
Controllers: ReplicaSet/nginx-deployment-1370807587
Containers:
  nginx:
    Image: nginx
    Port: 80/TCP
    QoS Tier:
      memory: Guaranteed
      cpu: Guaranteed
    Limits:
      cpu: 1
      memory: 128Mi
    Requests:
      cpu: 1
      memory: 128Mi
  Environment Variables:
  Volumes:
    default-token-4bcbi:
      Type: Secret (a volume populated by a Secret)
      SecretName: default-token-4bcbi
Events:
FirstSeen   LastSeenCount   From
SubobjectPath   Type   Reason           Message
-----  -----  -----  -----
-----  -----  -----  -----
1m        48s       7     {default-
scheduler }          Warning   FailedScheduling
(nginx-deployment-1370807587-fz9sd) failed to fit in any node
  fit failure on node (kubernetes-node-6ta5): Node didn't
have enough resource: CPU, requested: 1000, used: 1420,
capacity: 2000
  fit failure on node (kubernetes-node-wul5): Node didn't
have enough resource: CPU, requested: 1000, used: 1100,
capacity: 2000
```

Here you can see the event generated by the scheduler saying that the Pod failed to schedule for reason `FailedScheduling` (and possibly others). The message tells us that there were not enough resources for the Pod on any of the nodes.

To correct this situation, you can use `kubectl scale` to update your Deployment to specify four or fewer replicas. (Or you could just leave the one Pod pending, which is harmless.)

Events such as the ones you saw at the end of `kubectl describe pod` are persisted in etcd and provide high-level information on what is happening in the cluster. To list all events you can use

```
kubectl get events
```

but you have to remember that events are namespaced. This means that if you're interested in events for some namespaced object (e.g. what happened with Pods in namespace `my-namespace`) you need to explicitly provide a namespace to the command:

```
kubectl get events --namespace=my-namespace
```

To see events from all namespaces, you can use the `--all-namespaces` argument.

In addition to `kubectl describe pod`, another way to get extra information about a pod (beyond what is provided by `kubectl get pod`) is to pass the `-o yaml` output format flag to `kubectl get pod`. This will give you, in YAML format, even more information than `kubectl describe pod`-essentially all of the information the system has about the Pod. Here you will see things like annotations (which are key-value metadata without the label restrictions, that is used internally by Kubernetes system components), restart policy, ports, and volumes.

```
kubectl get pod nginx-deployment-1006230814-6winp -o yaml
```

```
apiVersion: v1
kind: Pod
metadata:
  annotations:
    kubernetes.io/created-by: /
  {"kind": "SerializedReference", "apiVersion": "v1", "reference": {
    "kind": "ReplicaSet", "namespace": "default", "name": "nginx-deployment-1006230814", "uid": "4c84c175-f161-11e5-9a78-42010af00005", "apiVersion": "extensions", "resourceVersion": "133434"}}
  creationTimestamp: 2016-03-24T01:39:50Z
  generateName: nginx-deployment-1006230814-
  labels:
    app: nginx
    pod-template-hash: "1006230814"
  name: nginx-deployment-1006230814-6winp
  namespace: default
  resourceVersion: "133447"
  uid: 4c879808-f161-11e5-9a78-42010af00005
spec:
  containers:
```

```
- image: nginx
  imagePullPolicy: Always
  name: nginx
  ports:
    - containerPort: 80
      protocol: TCP
  resources:
    limits:
      cpu: 500m
      memory: 128Mi
    requests:
      cpu: 500m
      memory: 128Mi
  terminationMessagePath: /dev/termination-log
  volumeMounts:
    - mountPath: /var/run/secrets/kubernetes.io/
serviceaccount
  name: default-token-4bcbi
  readOnly: true
dnsPolicy: ClusterFirst
nodeName: kubernetes-node-wul5
restartPolicy: Always
securityContext: {}
serviceAccount: default
serviceAccountName: default
terminationGracePeriodSeconds: 30
volumes:
- name: default-token-4bcbi
  secret:
    secretName: default-token-4bcbi
status:
  conditions:
    - lastProbeTime: null
      lastTransitionTime: 2016-03-24T01:39:51Z
      status: "True"
      type: Ready
  containerStatuses:
    - containerID: docker://
90315cc9f513c724e9957a4788d3e625a078de84750f244a40f97ae355eb1
149
      image: nginx
      imageID: docker://
6f62f48c4e55d700cf3eb1b5e33fa051802986b77b874cc351cce539e5163
707
      lastState: {}
      name: nginx
      ready: true
      restartCount: 0
      state:
        running:
          startedAt: 2016-03-24T01:39:51Z
hostIP: 10.240.0.9
```

```
phase: Running
podIP: 10.244.0.6
startTime: 2016-03-24T01:39:49Z
```

## Example: debugging a down/unreachable node

Sometimes when debugging it can be useful to look at the status of a node - for example, because you've noticed strange behavior of a Pod that's running on the node, or to find out why a Pod won't schedule onto the node. As with Pods, you can use `kubectl describe node` and `kubectl get node -o yaml` to retrieve detailed information about nodes. For example, here's what you'll see if a node is down (disconnected from the network, or kubelet dies and won't restart, etc.). Notice the events that show the node is NotReady, and also notice that the pods are no longer running (they are evicted after five minutes of NotReady status).

```
kubectl get nodes
```

NAME	STATUS	ROLES	AGE
VERSION			
kubernetes-node-861h v1.13.0	NotReady	<none>	1h
kubernetes-node-bols v1.13.0	Ready	<none>	1h
kubernetes-node-st6x v1.13.0	Ready	<none>	1h
kubernetes-node-unaj v1.13.0	Ready	<none>	1h

```
kubectl describe node kubernetes-node-861h
```

Name:	kubernetes-node-861h		
Role			
Labels:	kubernetes.io/arch=amd64 kubernetes.io/os=linux kubernetes.io/hostname=kubernetes-node-861h		
Annotations:	node.alpha.kubernetes.io/ttl=0 volumes.kubernetes.io/controller-managed-attach-detach=true		
Taints:	<none>		
CreationTimestamp:	Mon, 04 Sep 2017 17:13:23 +0800		
Phase:			
Conditions:			
Type	Status	LastHeartbeatTime	Message
LastTransitionTime		Reason	
-----	-----	-----	-----
-----	-----	-----	-----
OutOfDisk 16:04:28 +0800 +0800	Unknown Fri, 08 Sep 2017 16:20:58 NodeStatusUnknown	Fri, 08 Sep 2017 16:20:58 Kubelet stopped	

```

posting node status.
  MemoryPressure          Unknown      Fri, 08 Sep 2017
16:04:28 +0800      Fri, 08 Sep 2017 16:20:58
+0800      NodeStatusUnknown      Kubelet stopped
posting node status.
  DiskPressure          Unknown      Fri, 08 Sep 2017
16:04:28 +0800      Fri, 08 Sep 2017 16:20:58
+0800      NodeStatusUnknown      Kubelet stopped
posting node status.
  Ready                  Unknown      Fri, 08 Sep 2017
16:04:28 +0800      Fri, 08 Sep 2017 16:20:58
+0800      NodeStatusUnknown      Kubelet stopped
posting node status.
Addresses: 10.240.115.55,104.197.0.26
Capacity:
  cpu:                2
  hugePages:          0
  memory:             4046788Ki
  pods:               110
Allocatable:
  cpu:                1500m
  hugePages:          0
  memory:             1479263Ki
  pods:               110
System Info:
  Machine ID:         8e025a21a4254e11b028584d9d8b12c4
  System UUID:        349075D1-D169-4F25-9F2A-
E886850C47E3
  Boot ID:            5cd18b37-c5bd-4658-94e0-
e436d3f110e0
  Kernel Version:     4.4.0-31-generic
  OS Image:           Debian GNU/Linux 8 (jessie)
  Operating System:   linux
  Architecture:       amd64
  Container Runtime Version: docker://1.12.5
  Kubelet Version:    v1.6.9+a3d1dfa6f4335
  Kube-Proxy Version: v1.6.9+a3d1dfa6f4335
  ExternalID:         15233045891481496305
  Non-terminated Pods: (9 in total)
    Namespace
Name                                     CPU
Requests      CPU Limits      Memory Requests      Memory Limits
-----      -----
-----      -----
-----      -----
.....
Allocated resources:
  (Total limits may be over 100 percent, i.e.,
overcommitted.)
  CPU Requests  CPU Limits      Memory Requests
  Memory Limits

```

```
-----  
-----  
 900m (60%)  2200m (146%)  1009286400 (66%)  
5681286400 (375%)  
Events: <none>  
  
kubectl get node kubernetes-node-861h -o yaml  
  
apiVersion: v1  
kind: Node  
metadata:  
  creationTimestamp: 2015-07-10T21:32:29Z  
  labels:  
    kubernetes.io/hostname: kubernetes-node-861h  
    name: kubernetes-node-861h  
    resourceVersion: "757"  
    uid: 2a69374e-274b-11e5-a234-42010af0d969  
spec:  
  externalID: "15233045891481496305"  
  podCIDR: 10.244.0.0/24  
  providerID: gce://striped-torus-760/us-central1-b/  
kubernetes-node-861h  
status:  
  addresses:  
  - address: 10.240.115.55  
    type: InternalIP  
  - address: 104.197.0.26  
    type: ExternalIP  
  capacity:  
    cpu: "1"  
    memory: 3800808Ki  
    pods: "100"  
  conditions:  
  - lastHeartbeatTime: 2015-07-10T21:34:32Z  
    lastTransitionTime: 2015-07-10T21:35:15Z  
    reason: Kubelet stopped posting node status.  
    status: Unknown  
    type: Ready  
nodeInfo:  
  bootID: 4e316776-b40d-4f78-a4ea-ab0d73390897  
  containerRuntimeVersion: docker://Unknown  
  kernelVersion: 3.16.0-0.bpo.4-amd64  
  kubeProxyVersion: v0.21.1-185-gffc5a86098dc01  
  kubeletVersion: v0.21.1-185-gffc5a86098dc01  
  machineID: ""  
  osImage: Debian GNU/Linux 7 (wheezy)  
  systemUUID: ABE5F6B4-D44B-108B-C46A-24CCE16C8B6E
```

## What's next

Learn about additional debugging tools, including:

- [Logging](#)
- [Monitoring](#)
- [Getting into containers via exec](#)
- [Connecting to containers via proxies](#)
- [Connecting to containers via port forwarding](#)
- [Inspect Kubernetes node with crictl](#)

## Feedback

Was this page helpful?

Yes No

Thanks for the feedback. If you have a specific, answerable question about how to use Kubernetes, ask it on [Stack Overflow](#). Open an issue in the GitHub repo if you want to [report a problem](#) or [suggest an improvement](#).

---

[Create an Issue](#) [Edit This Page](#)

Page last modified on August 09, 2019 at 4:59 AM PST by [Remove references to selflinks \(#15751\)](#) ([Page History](#))

[Edit This Page](#)

# Auditing

Kubernetes auditing provides a security-relevant chronological set of records documenting the sequence of activities that have affected system by individual users, administrators or other components of the system. It allows cluster administrator to answer the following questions:

- what happened?
- when did it happen?
- who initiated it?
- on what did it happen?

- where was it observed?
- from where was it initiated?
- to where was it going?
- [Audit Policy](#)
- [Audit backends](#)
- [Setup for multiple API servers](#)
- [Log Collector Examples](#)
- [What's next](#)

[Kube-apiserver](#) performs auditing. Each request on each stage of its execution generates an event, which is then pre-processed according to a certain policy and written to a backend. The policy determines what's recorded and the backends persist the records. The current backend implementations include logs files and webhooks.

Each request can be recorded with an associated "stage". The known stages are:

- RequestReceived - The stage for events generated as soon as the audit handler receives the request, and before it is delegated down the handler chain.
- ResponseStarted - Once the response headers are sent, but before the response body is sent. This stage is only generated for long-running requests (e.g. watch).
- ResponseComplete - The response body has been completed and no more bytes will be sent.
- Panic - Events generated when a panic occurred.

**Note:** The audit logging feature increases the memory consumption of the API server because some context required for auditing is stored for each request. Additionally, memory consumption depends on the audit logging configuration.

## Audit Policy

Audit policy defines rules about what events should be recorded and what data they should include. The audit policy object structure is defined in the [audit.k8s.io API group](#). When an event is processed, it's compared against the list of rules in order. The first matching rule sets the "audit level" of the event. The known audit levels are:

- None - don't log events that match this rule.
- Metadata - log request metadata (requesting user, timestamp, resource, verb, etc.) but not request or response body.
- Request - log event metadata and request body but not response body. This does not apply for non-resource requests.
- RequestResponse - log event metadata, request and response bodies. This does not apply for non-resource requests.

You can pass a file with the policy to [kube-apiserver](#) using the `--audit-policy-file` flag. If the flag is omitted, no events are logged. Note that

the `rules` field **must** be provided in the audit policy file. A policy with no (0) rules is treated as illegal.

Below is an example audit policy file:

```

apiVersion: audit.k8s.io/v1 # This is required.
kind: Policy
# Don't generate audit events for all requests in
RequestReceived stage.
omitStages:
- "RequestReceived"
rules:
# Log pod changes at RequestResponse level
- level: RequestResponse
resources:
- group: ""
  # Resource "pods" doesn't match requests to any
  subresource of pods,
  # which is consistent with the RBAC policy.
  resources: ["pods"]
# Log "pods/log", "pods/status" at Metadata level
- level: Metadata
resources:
- group: ""
  resources: ["pods/log", "pods/status"]

# Don't log requests to a configmap called "controller-leader"
- level: None
resources:
- group: ""
  resources: ["configmaps"]
  resourceNames: ["controller-leader"]

# Don't log watch requests by the "system:kube-proxy" on
endpoints or services
- level: None
users: ["system:kube-proxy"]
verbs: ["watch"]
resources:
- group: "" # core API group
  resources: ["endpoints", "services"]

# Don't log authenticated requests to certain non-
resource URL paths.
- level: None
userGroups: ["system:authenticated"]
nonResourceURLs:
- "/api*" # Wildcard matching.
- "/version"

# Log the request body of configmap changes in kube-
system.
- level: Request
resources:
- group: "" # core API group
  resources: ["configmaps"]
# This rule only applies to resources in the "kube-
system" namespace.
  # The empty string "" can be used to select non-
  namespaced resources

```

You can use a minimal audit policy file to log all requests at the Metadata level:

```
# Log all requests at the Metadata level.
apiVersion: audit.k8s.io/v1
kind: Policy
rules:
- level: Metadata
```

The audit profile used by GCE should be used as reference by admins constructing their own audit profiles. You can check the [configure-helper.sh](#) script, which generates the audit policy file. You can see most of the audit policy file by looking directly at the script.

## Audit backends

Audit backends persist audit events to an external storage. [Kube-apiserver](#) out of the box provides three backends:

- Log backend, which writes events to a disk
- Webhook backend, which sends events to an external API
- Dynamic backend, which configures webhook backends through an AuditSink API object.

In all cases, audit events structure is defined by the API in the `audit.k8s.io` API group. The current version of the API is [v1](#).

### Note:

In case of patches, request body is a JSON array with patch operations, not a JSON object with an appropriate Kubernetes API object. For example, the following request body is a valid patch request to `/apis/batch/v1/namespaces/some-namespace/jobs/some-job-name`.

```
[  
  {  
    "op": "replace",  
    "path": "/spec/parallelism",  
    "value": 0  
  },  
  {  
    "op": "remove",  
    "path": "/spec/template/spec/containers/0/  
terminationMessagePolicy"  
  }  
]
```

## Log backend

Log backend writes audit events to a file in JSON format. You can configure log audit backend using the following [kube-apiserver](#) flags:

- `--audit-log-path` specifies the log file path that log backend uses to write audit events. Not specifying this flag disables log backend.  
- means standard out
- `--audit-log-maxage` defined the maximum number of days to retain old audit log files
- `--audit-log-maxbackup` defines the maximum number of audit log files to retain
- `--audit-log-maxsize` defines the maximum size in megabytes of the audit log file before it gets rotated

## Webhook backend

Webhook backend sends audit events to a remote API, which is assumed to be the same API as [kube-apiserver](#) exposes. You can configure webhook audit backend using the following kube-apiserver flags:

- `--audit-webhook-config-file` specifies the path to a file with a webhook configuration. Webhook configuration is effectively a [kubeconfig](#).
- `--audit-webhook-initial-backoff` specifies the amount of time to wait after the first failed request before retrying. Subsequent requests are retried with exponential backoff.

The webhook config file uses the kubeconfig format to specify the remote address of the service and credentials used to connect to it.

In v1.13 webhook backends can be configured [dynamically](#).

## Batching

Both log and webhook backends support batching. Using webhook as an example, here's the list of available flags. To get the same flag for log backend, replace webhook with log in the flag name. By default, batching is enabled in webhook and disabled in log. Similarly, by default throttling is enabled in webhook and disabled in log.

- `--audit-webhook-mode` defines the buffering strategy. One of the following:
  - `batch` - buffer events and asynchronously process them in batches. This is the default.
  - `blocking` - block API server responses on processing each individual event.
  - `blocking-strict` - Same as blocking, but when there is a failure during audit logging at RequestReceived stage, the whole request to apiserver will fail.

The following flags are used only in the batch mode.

- `--audit-webhook-batch-buffer-size` defines the number of events to buffer before batching. If the rate of incoming events overflows the buffer, events are dropped.
- `--audit-webhook-batch-max-size` defines the maximum number of events in one batch.
- `--audit-webhook-batch-max-wait` defines the maximum amount of time to wait before unconditionally batching events in the queue.
- `--audit-webhook-batch-throttle-qps` defines the maximum average number of batches generated per second.
- `--audit-webhook-batch-throttle-burst` defines the maximum number of batches generated at the same moment if the allowed QPS was underutilized previously.

## Parameter tuning

Parameters should be set to accommodate the load on the apiserver.

For example, if kube-apiserver receives 100 requests each second, and each request is audited only on `ResponseStarted` and `ResponseComplete` stages, you should account for ~200 audit events being generated each second. Assuming that there are up to 100 events in a batch, you should set throttling level at least 2 QPS. Assuming that the backend can take up to 5 seconds to write events, you should set the buffer size to hold up to 5 seconds of events, i.e. 10 batches, i.e. 1000 events.

In most cases however, the default parameters should be sufficient and you don't have to worry about setting them manually. You can look at the following Prometheus metrics exposed by kube-apiserver and in the logs to monitor the state of the auditing subsystem.

- `apiserver_audit_event_total` metric contains the total number of audit events exported.
- `apiserver_audit_error_total` metric contains the total number of events dropped due to an error during exporting.

## Truncate

Both log and webhook backends support truncating. As an example, the following is the list of flags available for the log backend:

- `audit-log-truncate-enabled` whether event and batch truncating is enabled.
- `audit-log-truncate-max-batch-size` maximum size in bytes of the batch sent to the underlying backend.
- `audit-log-truncate-max-event-size` maximum size in bytes of the audit event sent to the underlying backend.

By default truncate is disabled in both webhook and log, a cluster administrator should set audit-log-truncate-enabled or audit-webhook-truncate-enabled to enable the feature.

## Dynamic backend

**FEATURE STATE:** Kubernetes v1.13 [alpha](#)

This feature is currently in a *alpha* state, meaning:

[Edit This Page](#)

# Auditing with Falco

## Use Falco to collect audit events

[Falco](#) is an open source project for intrusion and abnormality detection for Cloud Native platforms. This section describes how to set up Falco, how to send audit events to the Kubernetes Audit endpoint exposed by Falco, and how Falco applies a set of rules to automatically detect suspicious behavior.

### Install Falco

Install Falco by using one of the following methods:

- [Standalone Falco](#)
- [Kubernetes DaemonSet](#)
- [Falco Helm Chart](#)

Once Falco is installed make sure it is configured to expose the Audit webhook. To do so, use the following configuration:

```
webserver:  
  enabled: true  
  listen_port: 8765  
  k8s_audit_endpoint: /k8s_audit  
  ssl_enabled: false  
  ssl_certificate: /etc/falco/falco.pem
```

This configuration is typically found in the `/etc/falco/falco.yaml` file. If Falco is installed as a Kubernetes DaemonSet, edit the `falco-config` ConfigMap and add this configuration.

### Configure Kubernetes Audit

1. Create a [kubeconfig file](#) for the [kube-apiserver](#) webhook audit backend.

```
cat <<EOF > /etc/kubernetes/audit-webhook-kubeconfig  
apiVersion: v1
```

```
kind: Config
clusters:
- cluster:
    server: http://<ip_of_falco>:8765/k8s_audit
    name: falco
contexts:
- context:
    cluster: falco
    user: ""
    name: default-context
current-context: default-context
preferences: {}
users: []
EOF
```

2. Start [kube-apiserver](#) with the following options:

```
--audit-policy-file=/etc/kubernetes/audit-
policy.yaml --audit-webhook-config-file=/etc/
kubernetes/audit-webhook-kubeconfig
```

## Audit Rules

Rules devoted to Kubernetes Audit Events can be found in [k8s\\_audit\\_rules.yaml](#). If Audit Rules is installed as a native package or using the official Docker images, Falco copies the rules file to /etc/falco/, so they are available for use.

There are three classes of rules.

The first class of rules looks for suspicious or exceptional activities, such as:

- Any activity by an unauthorized or anonymous user.
- Creating a pod with an unknown or disallowed image.
- Creating a privileged pod, a pod mounting a sensitive filesystem from the host, or a pod using host networking.
- Creating a NodePort service.
- Creating a ConfigMap containing private credentials, such as passwords and cloud provider secrets.
- Attaching to or executing a command on a running pod.
- Creating a namespace external to a set of allowed namespaces.
- Creating a pod or service account in the kube-system or kube-public namespaces.
- Trying to modify or delete a system ClusterRole.
- Creating a ClusterRoleBinding to the cluster-admin role.
- Creating a ClusterRole with wildcarded verbs or resources. For example, overly permissive.
- Creating a ClusterRole with write permissions or a ClusterRole that can execute commands on pods.

A second class of rules tracks resources being created or destroyed, including:

- Deployments
- Services
- ConfigMaps
- Namespaces
- Service accounts
- Role/ClusterRoles
- Role/ClusterRoleBindings

The final class of rules simply displays any Audit Event received by Falco. This rule is disabled by default, as it can be quite noisy.

For further details, see [Kubernetes Audit Events](#) in the Falco documentation.

## Feedback

Was this page helpful?

Yes No

Thanks for the feedback. If you have a specific, answerable question about how to use Kubernetes, ask it on [Stack Overflow](#). Open an issue in the GitHub repo if you want to [report a problem](#) or [suggest an improvement](#).

---

[Create an Issue](#) [Edit This Page](#)

Page last modified on August 23, 2019 at 2:13 PM PST by [split falco section to a new page \(#16011\)](#) ([Page History](#))

[Edit This Page](#)

# Debug a StatefulSet

This task shows you how to debug a StatefulSet.

- [Before you begin](#)
- [Debugging a StatefulSet](#)
- [What's next](#)

## Before you begin

- You need to have a Kubernetes cluster, and the kubectl command-line tool must be configured to communicate with your cluster.
- You should have a StatefulSet running that you want to investigate.

# Debugging a StatefulSet

In order to list all the pods which belong to a StatefulSet, which have a label `app=myapp` set on them, you can use the following:

```
kubectl get pods -l app=myapp
```

If you find that any Pods listed are in Unknown or Terminating state for an extended period of time, refer to the [Deleting StatefulSet Pods](#) task for instructions on how to deal with them. You can debug individual Pods in a StatefulSet using the [Debugging Pods](#) guide.

## What's next

Learn more about [debugging an init-container](#).

## Feedback

Was this page helpful?

Yes No

Thanks for the feedback. If you have a specific, answerable question about how to use Kubernetes, ask it on [Stack Overflow](#). Open an issue in the GitHub repo if you want to [report a problem](#) or [suggest an improvement](#).

---

[Create an Issue](#) [Edit This Page](#)

Page last modified on May 05, 2018 at 6:00 PM PST by [Convert site to Hugo \(#8316\)](#) ([Page History](#))

[Edit This Page](#)

# Debug Init Containers

This page shows how to investigate problems related to the execution of Init Containers. The example command lines below refer to the Pod as <pod-name> and the Init Containers as <init-container-1> and <init-container-2>.

- [Before you begin](#)
- [Checking the status of Init Containers](#)
- [Getting details about Init Containers](#)
- [Accessing logs from Init Containers](#)
- [Understanding Pod status](#)

# Before you begin

You need to have a Kubernetes cluster, and the kubectl command-line tool must be configured to communicate with your cluster. 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`.

- You should be familiar with the basics of [Init Containers](#).
- You should have [Configured an Init Container](#).

## Checking the status of Init Containers

Display the status of your pod:

```
kubectl get pod <pod-name>
```

For example, a status of `Init:1/2` indicates that one of two Init Containers has completed successfully:

NAME	READY	STATUS	RESTARTS	AGE
<pod-name>	0/1	Init:1/2	0	7s

See [Understanding Pod status](#) for more examples of status values and their meanings.

## Getting details about Init Containers

View more detailed information about Init Container execution:

```
kubectl describe pod <pod-name>
```

For example, a Pod with two Init Containers might show the following:

```
Init Containers:  
  <init-container-1>:  
    Container ID: ...  
    ...  
    State: Terminated  
    Reason: Completed  
    Exit Code: 0  
    Started: ...  
    Finished: ...  
    Ready: True  
    Restart Count: 0  
    ...  
  <init-container-2>:
```

```
Container ID:    ...
...
State:          Waiting
Reason:         CrashLoopBackOff
Last State:    Terminated
Reason:         Error
Exit Code:      1
Started:        ...
Finished:       ...
Ready:          False
Restart Count:  3
...
```

You can also access the Init Container statuses programmatically by reading the `status.initContainerStatuses` field on the Pod Spec:

```
kubectl get pod nginx --template '{{.status.initContainerStatuses}}'
```

This command will return the same information as above in raw JSON.

## Accessing logs from Init Containers

Pass the Init Container name along with the Pod name to access its logs.

```
kubectl logs <pod-name> -c <init-container-2>
```

Init Containers that run a shell script print commands as they're executed. For example, you can do this in Bash by running `set -x` at the beginning of the script.

## Understanding Pod status

A Pod status beginning with `Init:` summarizes the status of Init Container execution. The table below describes some example status values that you might see while debugging Init Containers.

Status	Meaning
<code>Init:N/M</code>	The Pod has M Init Containers, and N have completed so far.
<code>Init:Error</code>	An Init Container has failed to execute.
<code>Init:CrashLoopBackOff</code>	An Init Container has failed repeatedly.
<code>Pending</code>	The Pod has not yet begun executing Init Containers.
<code>PodInitializing or Running</code>	The Pod has already finished executing Init Containers.

# Feedback

Was this page helpful?

Yes No

Thanks for the feedback. If you have a specific, answerable question about how to use Kubernetes, ask it on [Stack Overflow](#). Open an issue in the GitHub repo if you want to [report a problem](#) or [suggest an improvement](#).

---

[Create an Issue](#) [Edit This Page](#)

Page last modified on May 05, 2018 at 6:00 PM PST by [Convert site to Hugo \(#8316\)](#) ([Page History](#))

[Edit This Page](#)

# Debug Pods and ReplicationControllers

This page shows how to debug Pods and ReplicationControllers.

- [Before you begin](#)
- [Debugging Pods](#)
- [Debugging ReplicationControllers](#)

## Before you begin

You need to have a Kubernetes cluster, and the kubectl command-line tool must be configured to communicate with your cluster. 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`.

- You should be familiar with the basics of [Pods](#) and [Pod Lifecycle](#).

## Debugging Pods

The first step in debugging a pod is taking a look at it. Check the current state of the pod and recent events with the following command:

```
kubectl describe pods ${POD_NAME}
```

Look at the state of the containers in the pod. Are they all Running? Have there been recent restarts?

Continue debugging depending on the state of the pods.

## My pod stays pending

If a pod is stuck in Pending it means that it can not be scheduled onto a node. Generally this is because there are insufficient resources of one type or another that prevent scheduling. Look at the output of the `kube` `ctl` `describe`

... command above. There should be messages from the scheduler about why it can not schedule your pod. Reasons include:

## Insufficient resources

You may have exhausted the supply of CPU or Memory in your cluster. In this case you can try several things:

- [Add more nodes](#) to the cluster.
- [Terminate unneeded pods](#) to make room for pending pods.
- Check that the pod is not larger than your nodes. For example, if all nodes have a capacity of `cpu:1`, then a pod with a request of `cpu: 1.1` will never be scheduled.

You can check node capacities with the `kubectl get nodes -o <format>` command. Here are some example command lines that extract just the necessary information:

```
kubectl get nodes -o yaml | egrep '\sname:|cpu:|memory:'  
kubectl get nodes -o json | jq '.items[] |  
{name: .metadata.name, cap: .status.capacity}'
```

The [resource quota](#) feature can be configured to limit the total amount of resources that can be consumed. If used in conjunction with namespaces, it can prevent one team from hogging all the resources.

## Using hostPort

When you bind a pod to a `hostPort` there are a limited number of places that the pod can be scheduled. In most cases, `hostPort` is unnecessary; try using a service object to expose your pod. If you do require `hostPort` then you can only schedule as many pods as there are nodes in your container cluster.

## My pod stays waiting

If a pod is stuck in the `Waiting` state, then it has been scheduled to a worker node, but it can't run on that machine. Again, the information from `kubectl describe ...` should be informative. The most common cause of `Waiting` pods is a failure to pull the image. There are three things to check:

- Make sure that you have the name of the image correct.
- Have you pushed the image to the repository?
- Run a manual `docker pull <image>` on your machine to see if the image can be pulled.

## My pod is crashing or otherwise unhealthy

First, take a look at the logs of the current container:

```
kubectl logs ${POD_NAME} ${CONTAINER_NAME}
```

If your container has previously crashed, you can access the previous container's crash log with:

```
kubectl logs --previous ${POD_NAME} ${CONTAINER_NAME}
```

Alternately, you can run commands inside that container with `exec`:

```
kubectl exec ${POD_NAME} -c ${CONTAINER_NAME} -- ${CMD} ${ARG1} ${ARG2} ... ${ARGN}
```

**Note:** `-c ${CONTAINER_NAME}` is optional. You can omit it for pods that only contain a single container.

As an example, to look at the logs from a running Cassandra pod, you might run:

```
kubectl exec cassandra -- cat /var/log/cassandra/system.log
```

If your cluster enabled it, you can also try adding an [ephemeral container](#) into the existing pod. You can use the new temporary container to run arbitrary commands, for example, to diagnose problems inside the Pod. See the page about [ephemeral container](#) for more details, including feature availability.

If none of these approaches work, you can find the host machine that the pod is running on and SSH into that host.

## Debugging ReplicationControllers

ReplicationControllers are fairly straightforward. They can either create pods or they can't. If they can't create pods, then please refer to the [instructions above](#) to debug your pods.

You can also use `kubectl describe rc ${CONTROLLER_NAME}` to inspect events related to the replication controller.

## Feedback

Was this page helpful?

Yes No

Thanks for the feedback. If you have a specific, answerable question about how to use Kubernetes, ask it on [Stack Overflow](#). Open an issue in the GitHub repo if you want to [report a problem](#) or [suggest an improvement](#).

---

[Create an Issue](#) [Edit This Page](#)

Page last modified on January 30, 2020 at 6:28 AM PST by [feat: add ephemeral container approach inside pod debug page. \(#18754\)](#) ([Page History](#))

[Edit This Page](#)

# Debug Services

An issue that comes up rather frequently for new installations of Kubernetes is that a Service is not working properly. You've run your Deployment and created a Service, but you get no response when you try to access it. This document will hopefully help you to figure out what's going wrong.

- [Conventions](#)
- [Running commands in a Pod](#)
- [Setup](#)

- [Does the Service exist?](#)
- [Does the Service work by DNS?](#)
- [Does the Service work by IP?](#)
- [Is the Service correct?](#)
- [Does the Service have any Endpoints?](#)
- [Are the Pods working?](#)
- [Is the kube-proxy working?](#)
- [Seek help](#)
- [What's next](#)

## Conventions

Throughout this doc you will see various commands that you can run. Some commands need to be run within a Pod, others on a Kubernetes Node, and others can run anywhere you have kubectl and credentials for the cluster. To make it clear what is expected, this document will use the following conventions.

If the command "COMMAND" is expected to run in a Pod and produce "OUTPUT":

```
u@pod$ COMMAND
OUTPUT
```

If the command "COMMAND" is expected to run on a Node and produce "OUTPUT":

```
u@node$ COMMAND
OUTPUT
```

If the command is "kubectl ARGS":

```
kubectl ARGS
OUTPUT
```

## Running commands in a Pod

For many steps here you will want to see what a Pod running in the cluster sees. The simplest way to do this is to run an interactive alpine Pod:

```
kubectl run -it --rm --restart=Never alpine --image=alpine sh
/ #
```

**Note:** If you don't see a command prompt, try pressing enter.

If you already have a running Pod that you prefer to use, you can run a command in it using:

```
kubectl exec <POD-NAME> -c <CONTAINER-NAME> -- <COMMAND>
```

# Setup

For the purposes of this walk-through, let's run some Pods. Since you're probably debugging your own Service you can substitute your own details, or you can follow along and get a second data point.

```
kubectl run hostnames --image=k8s.gcr.io/serve_hostname \
    --labels=app=hostnames \
    --port=9376 \
    --replicas=3
deployment.apps/hostnames created
```

`kubectl` commands will print the type and name of the resource created or mutated, which can then be used in subsequent commands.

## Note:

This is the same as if you started the Deployment with the following YAML:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: hostnames
spec:
  selector:
    matchLabels:
      app: hostnames
  replicas: 3
  template:
    metadata:
      labels:
        app: hostnames
    spec:
      containers:
        - name: hostnames
          image: k8s.gcr.io/serve_hostname
          ports:
            - containerPort: 9376
              protocol: TCP
```

Confirm your Pods are running:

```
kubectl get pods -l app=hostnames
NAME                      READY   STATUS    RESTARTS   AGE
hostnames-632524106-bbpiw  1/1     Running   0          2m
hostnames-632524106-ly40y  1/1     Running   0          2m
hostnames-632524106-tlaok  1/1     Running   0          2m
```

## Does the Service exist?

The astute reader will have noticed that we did not actually create a Service yet - that is intentional. This is a step that sometimes gets forgotten, and is the first thing to check.

So what would happen if I tried to access a non-existent Service? Assuming you have another Pod that consumes this Service by name you would get something like:

```
u@pod$ wget -O- hostnames
Resolving hostnames (hostnames)... failed: Name or service
not known.
wget: unable to resolve host address 'hostnames'
```

So the first thing to check is whether that Service actually exists:

```
kubectl get svc hostnames
No resources found.
Error from server (NotFound): services "hostnames" not found
```

So we have a culprit, let's create the Service. As before, this is for the walk-through - you can use your own Service's details here.

```
kubectl expose deployment hostnames --port=80 --target-port=9
376
service/hostnames exposed
```

And read it back, just to be sure:

```
kubectl get svc hostnames
NAME      TYPE      CLUSTER-IP      EXTERNAL-IP      PORT(S)
AGE
hostnames   ClusterIP   10.0.1.175    <none>          80/TCP
5s
```

As before, this is the same as if you had started the Service with YAML:

```
apiVersion: v1
kind: Service
metadata:
  name: hostnames
spec:
  selector:
    app: hostnames
  ports:
  - name: default
    protocol: TCP
    port: 80
    targetPort: 9376
```

Now you can confirm that the Service exists.

# Does the Service work by DNS?

From a Pod in the same Namespace:

```
u@pod$ nslookup hostnames
Address 1: 10.0.0.10 kube-dns.kube-system.svc.cluster.local

Name:      hostnames
Address 1: 10.0.1.175 hostnames.default.svc.cluster.local
```

If this fails, perhaps your Pod and Service are in different Namespaces, try a namespace-qualified name:

```
u@pod$ nslookup hostnames.default
Address 1: 10.0.0.10 kube-dns.kube-system.svc.cluster.local

Name:      hostnames.default
Address 1: 10.0.1.175 hostnames.default.svc.cluster.local
```

If this works, you'll need to adjust your app to use a cross-namespace name, or run your app and Service in the same Namespace. If this still fails, try a fully-qualified name:

```
u@pod$ nslookup hostnames.default.svc.cluster.local
Address 1: 10.0.0.10 kube-dns.kube-system.svc.cluster.local

Name:      hostnames.default.svc.cluster.local
Address 1: 10.0.1.175 hostnames.default.svc.cluster.local
```

Note the suffix here: "default.svc.cluster.local". The "default" is the Namespace we're operating in. The "svc" denotes that this is a Service. The "cluster.local" is your cluster domain, which COULD be different in your own cluster.

You can also try this from a Node in the cluster:

**Note:** 10.0.0.10 is my DNS Service, yours might be different.

```
u@node$ nslookup hostnames.default.svc.cluster.local 10.0.0.10
Server:      10.0.0.10
Address:     10.0.0.10#53

Name:      hostnames.default.svc.cluster.local
Address:   10.0.1.175
```

If you are able to do a fully-qualified name lookup but not a relative one, you need to check that your /etc/resolv.conf file is correct.

```
u@pod$ cat /etc/resolv.conf
nameserver 10.0.0.10
search default.svc.cluster.local svc.cluster.local
```

```
cluster.local example.com
options ndots:5
```

The nameserver line must indicate your cluster's DNS Service. This is passed into kubelet with the `--cluster-dns` flag.

The search line must include an appropriate suffix for you to find the Service name. In this case it is looking for Services in the local Namespace (`default.svc.cluster.local`), Services in all Namespaces (`svc.cluster.local`), and the cluster (`cluster.local`). Depending on your own install you might have additional records after that (up to 6 total). The cluster suffix is passed into kubelet with the `--cluster-domain` flag. We assume that is "cluster.local" in this document, but yours might be different, in which case you should change that in all of the commands above.

The options line must set `ndots` high enough that your DNS client library considers search paths at all. Kubernetes sets this to 5 by default, which is high enough to cover all of the DNS names it generates.

## Does any Service exist in DNS?

If the above still fails - DNS lookups are not working for your Service - we can take a step back and see what else is not working. The Kubernetes master Service should always work:

```
u@pod$ nslookup kubernetes.default
Server: 10.0.0.10
Address 1: 10.0.0.10 kube-dns.kube-system.svc.cluster.local

Name: kubernetes.default
Address 1: 10.0.0.1 kubernetes.default.svc.cluster.local
```

If this fails, you might need to go to the kube-proxy section of this doc, or even go back to the top of this document and start over, but instead of debugging your own Service, debug DNS.

## Does the Service work by IP?

Assuming we can confirm that DNS works, the next thing to test is whether your Service works at all. From a node in your cluster, access the Service's IP (from `kubectl get` above).

```
u@node$ curl 10.0.1.175:80
hostnames-0uton

u@node$ curl 10.0.1.175:80
hostnames-yp2kp

u@node$ curl 10.0.1.175:80
hostnames-bvc05
```

If your Service is working, you should get correct responses. If not, there are a number of things that could be going wrong. Read on.

## Is the Service correct?

It might sound silly, but you should really double and triple check that your Service is correct and matches your Pod's port. Read back your Service and verify it:

```
kubectl get service hostnames -o json
```

```
{
  "kind": "Service",
  "apiVersion": "v1",
  "metadata": {
    "name": "hostnames",
    "namespace": "default",
    "uid": "428c8b6c-24bc-11e5-936d-42010af0a9bc",
    "resourceVersion": "347189",
    "creationTimestamp": "2015-07-07T15:24:29Z",
    "labels": {
      "app": "hostnames"
    }
  },
  "spec": {
    "ports": [
      {
        "name": "default",
        "protocol": "TCP",
        "port": 80,
        "targetPort": 9376,
        "nodePort": 0
      }
    ],
    "selector": {
      "app": "hostnames"
    },
    "clusterIP": "10.0.1.175",
    "type": "ClusterIP",
    "sessionAffinity": "None"
  },
  "status": {
    "loadBalancer": {}
  }
}
```

- Is the port you are trying to access in spec.ports[]?
- Is the targetPort correct for your Pods (many Pods choose to use a different port than the Service)?
- If you meant it to be a numeric port, is it a number (9376) or a string "9376"?

- If you meant it to be a named port, do your Pods expose a port with the same name?
- Is the port's protocol the same as the Pod's?

## Does the Service have any Endpoints?

If you got this far, we assume that you have confirmed that your Service exists and is resolved by DNS. Now let's check that the Pods you ran are actually being selected by the Service.

Earlier we saw that the Pods were running. We can re-check that:

```
kubectl get pods -l app=hostnames
NAME           READY   STATUS    RESTARTS   AGE
hostnames-0uton 1/1     Running   0          1h
hostnames-bvc05 1/1     Running   0          1h
hostnames-yp2kp 1/1     Running   0          1h
```

The "AGE" column says that these Pods are about an hour old, which implies that they are running fine and not crashing.

The `-l app=hostnames` argument is a label selector - just like our Service has. Inside the Kubernetes system is a control loop which evaluates the selector of every Service and saves the results into an Endpoints object.

```
kubectl get endpoints hostnames
NAME      ENDPOINTS
hostnames  10.244.0.5:9376,10.244.0.6:9376,10.244.0.7:9376
```

This confirms that the endpoints controller has found the correct Pods for your Service. If the hostnames row is blank, you should check that the `spec.selector` field of your Service actually selects for `metadata.labels` values on your Pods. A common mistake is to have a typo or other error, such as the Service selecting for `run=hostnames`, but the Deployment specifying `app=hostnames`.

## Are the Pods working?

At this point, we know that your Service exists and has selected your Pods. Let's check that the Pods are actually working - we can bypass the Service mechanism and go straight to the Pods.

**Note:** These commands use the Pod port (9376), rather than the Service port (80).

```
u@pod$ wget -qO- 10.244.0.5:9376
hostnames-0uton

pod $ wget -qO- 10.244.0.6:9376
hostnames-bvc05
```

```
u@pod$ wget -qO- 10.244.0.7:9376  
hostnames-yp2kp
```

We expect each Pod in the Endpoints list to return its own hostname. If this is not what happens (or whatever the correct behavior is for your own Pods), you should investigate what's happening there. You might find `kubectl logs` to be useful or `kubectl exec` directly to your Pods and check service from there.

Another thing to check is that your Pods are not crashing or being restarted. Frequent restarts could lead to intermittent connectivity issues.

kubectl get pods -l app=hostnames					
NAME	READY	STATUS	RESTARTS	AGE	
hostnames-632524106-bbpiw	1/1	Running	0	2m	
hostnames-632524106-ly40y	1/1	Running	0	2m	
hostnames-632524106-tlaok	1/1	Running	0	2m	

If the restart count is high, read more about how to [debug pods](#).

## Is the kube-proxy working?

If you get here, your Service is running, has Endpoints, and your Pods are actually serving. At this point, the whole Service proxy mechanism is suspect. Let's confirm it, piece by piece.

### Is kube-proxy running?

Confirm that kube-proxy is running on your Nodes. You should get something like the below:

```
u@node$ ps auxw | grep kube-proxy  
root 4194 0.4 0.1 101864 17696 ? Sl Jul04 25:43 /usr/  
local/bin/kube-proxy --master=https://kubernetes-master --  
kubeconfig=/var/lib/kube-proxy/kubeconfig --v=2
```

Next, confirm that it is not failing something obvious, like contacting the master. To do this, you'll have to look at the logs. Accessing the logs depends on your Node OS. On some OSes it is a file, such as `/var/log/kube-proxy.log`, while other OSes use `journalctl` to access logs. You should see something like:

```
I1027 22:14:53.995134 5063 server.go:200] Running in  
resource-only container "/kube-proxy"  
I1027 22:14:53.998163 5063 server.go:247] Using iptables  
Proxier.  
I1027 22:14:53.999055 5063 server.go:255] Tearing down  
userspace rules. Errors here are acceptable.  
I1027 22:14:54.038140 5063 proxier.go:352] Setting
```

```
endpoints for "kube-system/kube-dns:dns-tcp" to [10.244.1.3:53]
I1027 22:14:54.038164    5063 proxier.go:352] Setting
endpoints for "kube-system/kube-dns:dns" to [10.244.1.3:53]
I1027 22:14:54.038209    5063 proxier.go:352] Setting
endpoints for "default/kubernetes:https" to [10.240.0.2:443]
I1027 22:14:54.038238    5063 proxier.go:429] Not syncing
iptables until Services and Endpoints have been received
from master
I1027 22:14:54.040048    5063 proxier.go:294] Adding new
service "default/kubernetes:https" at 10.0.0.1:443/TCP
I1027 22:14:54.040154    5063 proxier.go:294] Adding new
service "kube-system/kube-dns:dns" at 10.0.0.10:53/UDP
I1027 22:14:54.040223    5063 proxier.go:294] Adding new
service "kube-system/kube-dns:dns-tcp" at 10.0.0.10:53/TCP
```

If you see error messages about not being able to contact the master, you should double-check your Node configuration and installation steps.

One of the possible reasons that kube-proxy cannot run correctly is that the required conntrack binary cannot be found. This may happen on some Linux systems, depending on how you are installing the cluster, for example, you are installing Kubernetes from scratch. If this is the case, you need to manually install the conntrack package (e.g. sudo apt install conntrack on Ubuntu) and then retry.

## Is kube-proxy writing iptables rules?

One of the main responsibilities of kube-proxy is to write the iptables rules which implement Services. Let's check that those rules are getting written.

The kube-proxy can run in "userspace" mode, "iptables" mode or "ipvs" mode. Hopefully you are using the "iptables" mode or "ipvs" mode. You should see one of the following cases.

### Userspace

```
u@node$ iptables-save | grep hostnames
-A KUBE-PORTALS-CONTAINER -d 10.0.1.175/32 -p tcp -m comment
--comment "default/hostnames:default" -m tcp --dport 80 -j
REDIRECT --to-ports 48577
-A KUBE-PORTALS-HOST -d 10.0.1.175/32 -p tcp -m comment --
comment "default/hostnames:default" -m tcp --dport 80 -j
DNAT --to-destination 10.240.115.247:48577
```

There should be 2 rules for each port on your Service (just one in this example) - a "KUBE-PORTALS-CONTAINER" and a "KUBE-PORTALS-HOST". If you do not see these, try restarting kube-proxy with the -v flag set to 4, and then look at the logs again.

Almost nobody should be using the "userspace" mode any more, so we won't spend more time on it here.

## Iptables

```
u@node$ iptables-save | grep hostnames
-A KUBE-SEP-57KPRZ3JQVENLNBR -s 10.244.3.6/32 -m comment --
comment "default/hostnames:" -j MARK --set-xmark
0x00004000/0x00004000
-A KUBE-SEP-57KPRZ3JQVENLNBR -p tcp -m comment --comment "def
ault/hostnames:" -m tcp -j DNAT --to-destination 10.
244.3.6:9376
-A KUBE-SEP-WNBA2IHDGP2B0BGZ -s 10.244.1.7/32 -m comment --
comment "default/hostnames:" -j MARK --set-xmark
0x00004000/0x00004000
-A KUBE-SEP-WNBA2IHDGP2B0BGZ -p tcp -m comment --comment "def
ault/hostnames:" -m tcp -j DNAT --to-destination 10.
244.1.7:9376
-A KUBE-SEP-X3P2623AGDH6CDF3 -s 10.244.2.3/32 -m comment --
comment "default/hostnames:" -j MARK --set-xmark
0x00004000/0x00004000
-A KUBE-SEP-X3P2623AGDH6CDF3 -p tcp -m comment --comment "def
ault/hostnames:" -m tcp -j DNAT --to-destination 10.
244.2.3:9376
-A KUBE-SERVICES -d 10.0.1.175/32 -p tcp -m comment --
comment "default/hostnames: cluster IP" -m tcp --dport 80 -j
KUBE-SVC-NWV5X2332I40T4T3
-A KUBE-SVC-NWV5X2332I40T4T3 -m comment --comment "default/
hostnames:" -m statistic --mode random --probability 0.
33332999982 -j KUBE-SEP-WNBA2IHDGP2B0BGZ
-A KUBE-SVC-NWV5X2332I40T4T3 -m comment --comment "default/
hostnames:" -m statistic --mode random --probability 0.
50000000000 -j KUBE-SEP-X3P2623AGDH6CDF3
-A KUBE-SVC-NWV5X2332I40T4T3 -m comment --comment "default/
hostnames:" -j KUBE-SEP-57KPRZ3JQVENLNBR
```

There should be 1 rule in KUBE-SERVICES, 1 or 2 rules per endpoint in KUBE-SVC- (hash) (depending on SessionAffinity), one KUBE-SEP- (hash) chain per endpoint, and a few rules in each KUBE-SEP- (hash) chain. The exact rules will vary based on your exact config (including node-ports and load-balancers).

## IPVS

```
u@node$ ipvsadm -ln
Prot LocalAddress:Port Scheduler Flags
    -> RemoteAddress:Port          Forward Weight ActiveConn
InActConn
...
TCP  10.0.1.175:80  rr
    -> 10.244.0.5:9376           Masq      1      0
```

```
0      -> 10.244.0.6:9376          Masq    1      0
0      -> 10.244.0.7:9376          Masq    1      0
0
...

```

IPVS proxy will create a virtual server for each service address(e.g. Cluster IP, External IP, NodePort IP, Load Balancer IP etc.) and some corresponding real servers for endpoints of the service, if any. In this example, service hostnames(10.0.1.175:80) has 3 endpoints(10.244.0.5:9376, 10.244.0.6:9376, 10.244.0.7:9376) and you'll get results similar to above.

## Is kube-proxy proxying?

Assuming you do see the above rules, try again to access your Service by IP:

```
u@node$ curl 10.0.1.175:80
hostnames-0utan
```

If this fails and you are using the userspace proxy, you can try accessing the proxy directly. If you are using the iptables proxy, skip this section.

Look back at the `iptables-save` output above, and extract the port number that `kube-proxy` is using for your Service. In the above examples it is "48577". Now connect to that:

```
u@node$ curl localhost:48577
hostnames-yp2kp
```

If this still fails, look at the `kube-proxy` logs for specific lines like:

```
Setting endpoints for default/hostnames:default to [10.
244.0.5:9376 10.244.0.6:9376 10.244.0.7:9376]
```

If you don't see those, try restarting `kube-proxy` with the `-v` flag set to 4, and then look at the logs again.

## A Pod cannot reach itself via Service IP

This can happen when the network is not properly configured for "hairpin" traffic, usually when `kube-proxy` is running in `iptables` mode and Pods are connected with bridge network. The Kubelet exposes a `hairpin-mode` [flag](#) that allows endpoints of a Service to loadbalance back to themselves if they try to access their own Service VIP. The `hairpin-mode` flag must either be set to `hairpin-veth` or `promiscuous-bridge`.

The common steps to trouble shoot this are as follows:

- Confirm hairpin-mode is set to hairpin-veth or promiscuous-bridge. You should see something like the below. hairpin-mode is set to promiscuous-bridge in the following example.

```
u@node$ ps auxw|grep kubelet
root      3392  1.1  0.8 186804 65208 ?          Sl   00:
51 11:11 /usr/local/bin/kubelet --enable-debugging-
handlers=true --config=/etc/kubernetes/manifests --
allow-privileged=True --v=4 --cluster-dns=10.0.0.10 --
cluster-domain=cluster.local --configure-cbr0=true --
cgroup-root=/ --system-cgroups=/system --hairpin-mode=pr
omiscuous-bridge --runtime-cgroups=/docker-daemon --
kubelet-cgroups=/kubelet --babysit-daemons=true --max-
pods=110 --serialize-image-pulls=false --outofdisk-
transition-frequency=0
```

- Confirm the effective hairpin-mode. To do this, you'll have to look at kubelet log. Accessing the logs depends on your Node OS. On some OSes it is a file, such as /var/log/kubelet.log, while other OSes use journalctl to access logs. Please be noted that the effective hairpin mode may not match --hairpin-mode flag due to compatibility. Check if there is any log lines with key word hairpin in kubelet.log. There should be log lines indicating the effective hairpin mode, like something below.

```
I0629 00:51:43.648698    3252 kubelet.go:380] Hairpin
mode set to "promiscuous-bridge"
```

- If the effective hairpin mode is hairpin-veth, ensure the Kubelet has the permission to operate in /sys on node. If everything works properly, you should see something like:

```
for intf in /sys/devices/virtual/net/cbr0/brif/*; do
cat $intf/hairpin_mode; done
1
1
1
1
```

- If the effective hairpin mode is promiscuous-bridge, ensure Kubelet has the permission to manipulate linux bridge on node. If cbr0 bridge is used and configured properly, you should see:

```
u@node$ ifconfig cbr0 |grep PROMISC
UP BROADCAST RUNNING PROMISC MULTICAST  MTU:1460
Metric:1
```

- Seek help if none of above works out.

## Seek help

If you get this far, something very strange is happening. Your Service is running, has Endpoints, and your Pods are actually serving. You have DNS working, iptables rules installed, and kube-proxy does not seem to be misbehaving. And yet your Service is not working. You should probably let us know, so we can help investigate!

Contact us on [Slack](#) or [Forum](#) or [GitHub](#).

## What's next

Visit [troubleshooting document](#) for more information.

## Feedback

Was this page helpful?

Yes No

Thanks for the feedback. If you have a specific, answerable question about how to use Kubernetes, ask it on [Stack Overflow](#). Open an issue in the GitHub repo if you want to [report a problem](#) or [suggest an improvement](#).

---

[Create an Issue](#) [Edit This Page](#)

Page last modified on January 03, 2020 at 12:13 PM PST by [add missing backquote \(#18413\)](#) ([Page History](#))

[Edit This Page](#)

# Debugging Kubernetes nodes with crictl

**FEATURE STATE:** Kubernetes v1.11 [stable](#)  
This feature is *stable*, meaning:

[Edit This Page](#)

# Determine the Reason for Pod Failure

This page shows how to write and read a Container termination message.

Termination messages provide a way for containers to write information about fatal events to a location where it can be easily retrieved and surfaced by tools like dashboards and monitoring software. In most cases, information that you put in a termination message should also be written to the general [Kubernetes logs](#).

- [Before you begin](#)
- [Writing and reading a termination message](#)
- [Customizing the termination message](#)
- [What's next](#)

## Before you begin

You need to have a Kubernetes cluster, and the kubectl command-line tool must be configured to communicate with your cluster. 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`.

## Writing and reading a termination message

In this exercise, you create a Pod that runs one container. The configuration file specifies a command that runs when the container starts.

## [debug/termination.yaml](#)

```
apiVersion: v1
kind: Pod
metadata:
  name: termination-demo
spec:
  containers:
    - name: termination-demo-container
      image: debian
      command: ["/bin/sh"]
      args: ["-c", "sleep 10 && echo Sleep expired > /dev/termination-log"]
```

1. Create a Pod based on the YAML configuration file:

```
kubectl apply -f https://k8s.io/examples/debug/termination.yaml
```

In the YAML file, in the `cmd` and `args` fields, you can see that the container sleeps for 10 seconds and then writes "Sleep expired" to the `/dev/termination-log` file. After the container writes the "Sleep expired" message, it terminates.

2. Display information about the Pod:

```
kubectl get pod termination-demo
```

Repeat the preceding command until the Pod is no longer running.

3. Display detailed information about the Pod:

```
kubectl get pod termination-demo --output=yaml
```

The output includes the "Sleep expired" message:

```
apiVersion: v1
kind: Pod
...
  lastState:
    terminated:
      containerID: ...
      exitCode: 0
      finishedAt: ...
      message: |
        Sleep expired
...
```

4. Use a Go template to filter the output so that it includes only the termination message:

```
kubectl get pod termination-demo -o go-template="{{range .status.containerStatuses}}{{$lastState.terminated.message}}{{end}}"
```

## Customizing the termination message

Kubernetes retrieves termination messages from the termination message file specified in the `terminationMessagePath` field of a Container, which has a default value of `/dev/termination-log`. By customizing this field, you can tell Kubernetes to use a different file. Kubernetes uses the contents from the specified file to populate the Container's status message on both success and failure.

In the following example, the container writes termination messages to `/tmp/my-log` for Kubernetes to retrieve:

```
apiVersion: v1
kind: Pod
metadata:
  name: msg-path-demo
spec:
  containers:
    - name: msg-path-demo-container
      image: debian
      terminationMessagePath: "/tmp/my-log"
```

Moreover, users can set the `terminationMessagePolicy` field of a Container for further customization. This field defaults to "File" which means the termination messages are retrieved only from the termination message file. By setting the `terminationMessagePolicy` to "FallbackToLogsOnError", you can tell Kubernetes to use the last chunk of container log output if the termination message file is empty and the container exited with an error. The log output is limited to 2048 bytes or 80 lines, whichever is smaller.

## What's next

- See the `terminationMessagePath` field in [Container](#).
- Learn about [retrieving logs](#).
- Learn about [Go templates](#).

## Feedback

Was this page helpful?

Yes No

Thanks for the feedback. If you have a specific, answerable question about how to use Kubernetes, ask it on [Stack Overflow](#). Open an issue

in the GitHub repo if you want to [report a problem](#) or [suggest an improvement](#).

---

[Create an Issue](#) [Edit This Page](#)

Page last modified on June 20, 2019 at 4:48 AM PST by [specify Pod not to display redundant messages \(#14936\)](#) ([Page History](#))

[Edit This Page](#)

# Developing and debugging services locally

Kubernetes applications usually consist of multiple, separate services, each running in its own container. Developing and debugging these services on a remote Kubernetes cluster can be cumbersome, requiring

you to [get a shell on a running container](#) and running your tools inside the remote shell.

telepresence is a tool to ease the process of developing and debugging services locally, while proxying the service to a remote Kubernetes cluster. Using telepresence allows you to use custom tools, such as a debugger and IDE, for a local service and provides the service full access to ConfigMap, secrets, and the services running on the remote cluster.

This document describes using telepresence to develop and debug services running on a remote cluster locally.

- [Before you begin](#)
- [Getting a shell on a remote cluster](#)
- [Developing or debugging an existing service](#)
- [What's next](#)

## Before you begin

- Kubernetes cluster is installed
- kubectl is configured to communicate with the cluster
- [Telepresence](#) is installed

## Getting a shell on a remote cluster

Open a terminal and run telepresence with no arguments to get a telepresence shell. This shell runs locally, giving you full access to your local filesystem.

The telepresence shell can be used in a variety of ways. For example, write a shell script on your laptop, and run it directly from the shell in real time. You can do this on a remote shell as well, but you might not be able to use your preferred code editor, and the script is deleted when the container is terminated.

Enter `exit` to quit and close the shell.

## Developing or debugging an existing service

When developing an application on Kubernetes, you typically program or debug a single service. The service might require access to other services for testing and debugging. One option is to use the continuous deployment pipeline, but even the fastest deployment pipeline introduces a delay in the program or debug cycle.

Use the `--swap-deployment` option to swap an existing deployment with the Telepresence proxy. Swapping allows you to run a service

locally and connect to the remote Kubernetes cluster. The services in the remote cluster can now access the locally running instance.

To run telepresence with `--swap-deployment`, enter:

```
telepresence --swap-deployment $DEPLOYMENT_NAME
```

where `$DEPLOYMENT_NAME` is the name of your existing deployment.

Running this command spawns a shell. In the shell, start your service. You can then make edits to the source code locally, save, and see the changes take effect immediately. You can also run your service in a debugger, or any other local development tool.

## What's next

If you're interested in a hands-on tutorial, check out [this tutorial](#) that walks through locally developing the Guestbook application on Google Kubernetes Engine.

Telepresence has [numerous proxying options](#), depending on your situation.

For further reading, visit the [Telepresence website](#).

## Feedback

Was this page helpful?

Yes No

Thanks for the feedback. If you have a specific, answerable question about how to use Kubernetes, ask it on [Stack Overflow](#). Open an issue in the GitHub repo if you want to [report a problem](#) or [suggest an improvement](#).

---

[Create an Issue](#) [Edit This Page](#)

Page last modified on March 20, 2019 at 7:05 PM PST by [Fix relative links issue in English content \(#13307\)](#) ([Page History](#))

[Edit This Page](#)

## Events in Stackdriver

Kubernetes events are objects that provide insight into what is happening inside a cluster, such as what decisions were made by scheduler or why some pods were evicted from the node. You can read more about using events for debugging your application in the [Application Introspection and Debugging](#) section.

Since events are API objects, they are stored in the apiserver on master. To avoid filling up master's disk, a retention policy is enforced: events are removed one hour after the last occurrence. To provide

longer history and aggregation capabilities, a third party solution should be installed to capture events.

This article describes a solution that exports Kubernetes events to Stackdriver Logging, where they can be processed and analyzed.

**Note:** It is not guaranteed that all events happening in a cluster will be exported to Stackdriver. One possible scenario when events will not be exported is when event exporter is not running (e.g. during restart or upgrade). In most cases it's fine to use events for purposes like setting up [metrics](#) and [alerts](#), but you should be aware of the potential inaccuracy.

- [Deployment](#)
- [User Guide](#)

## Deployment

### Google Kubernetes Engine

In Google Kubernetes Engine, if cloud logging is enabled, event exporter is deployed by default to the clusters with master running version 1.7 and higher. To prevent disturbing your workloads, event exporter does not have resources set and is in the best effort QOS class, which means that it will be the first to be killed in the case of resource starvation. If you want your events to be exported, make sure you have enough resources to facilitate the event exporter pod. This may vary depending on the workload, but on average, approximately 100Mb RAM and 100m CPU is needed.

### Deploying to the Existing Cluster

Deploy event exporter to your cluster using the following command:

```
kubectl apply -f https://k8s.io/examples/debug/event-exporter.yaml
```

Since event exporter accesses the Kubernetes API, it requires permissions to do so. The following deployment is configured to work with RBAC authorization. It sets up a service account and a cluster role binding to allow event exporter to read events. To make sure that event exporter pod will not be evicted from the node, you can additionally set up resource requests. As mentioned earlier, 100Mb RAM and 100m CPU should be enough.

## [\*\*debug/event-exporter.yaml\*\*](#)

```
apiVersion: v1
kind: ServiceAccount
metadata:
  name: event-exporter-sa
  namespace: default
  labels:
    app: event-exporter
---
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRoleBinding
metadata:
  name: event-exporter-rb
  labels:
    app: event-exporter
roleRef:
  apiGroup: rbac.authorization.k8s.io
  kind: ClusterRole
  name: view
subjects:
- kind: ServiceAccount
  name: event-exporter-sa
  namespace: default
---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: event-exporter-v0.2.3
  namespace: default
  labels:
    app: event-exporter
spec:
  selector:
    matchLabels:
      app: event-exporter
  replicas: 1
  template:
    metadata:
      labels:
        app: event-exporter
    spec:
      serviceAccountName: event-exporter-sa
      containers:
        - name: event-exporter
          image: k8s.gcr.io/event-exporter:v0.2.3
          command:
            - '/event-exporter'
      terminationGracePeriodSeconds: 30
```

# User Guide

Events are exported to the GKE Cluster resource in Stackdriver Logging. You can find them by selecting an appropriate option from a drop-down menu of available resources:

The screenshot shows the Stackdriver Logging interface. At the top, there is a search bar labeled "Filter by label or text search". Below it are three dropdown menus: "GKE Cluster" (which is selected), "All logs", and "Any log level". The main area displays a list of log entries. On the left, a sidebar lists various Google Cloud resources with checkboxes next to them. The "GKE Cluster" checkbox is checked. The log entries themselves are listed on the right, each preceded by a small icon and some log text. At the bottom of the log list, there is a timestamp "19:38:50.000" followed by the message "Started container". To the right of this message is a yellow bar containing the text "No newer entries found matching current fil".

You can filter based on the event object fields using Stackdriver Logging [filtering mechanism](#). For example, the following query will show events from the scheduler about pods from deployment nginx-deployment:

```
resource.type="gke_cluster"
jsonPayload.kind="Event"
jsonPayload.source.component="default-scheduler"
jsonPayload.involvedObject.name:"nginx-deployment"
```

```
1 resource.type="gke_cluster"
2 jsonPayload.kind="Event"
3 jsonPayload.source.component="default-scheduler"
4 jsonPayload.involvedObject.name:"nginx-deployment"
```

[Submit Filter](#)

[Jump to date ▾](#)

2017-06-21 CEST



- ▶ i 19:38:41.000 Successfully assigned nginx-deployment-3088474477-9bbgf
- ▶ i 19:38:41.000 Successfully assigned nginx-deployment-3088474477-w3hzb



## Feedback

Was this page helpful?

Yes No

Thanks for the feedback. If you have a specific, answerable question about how to use Kubernetes, ask it on [Stack Overflow](#). Open an issue in the GitHub repo if you want to [report a problem](#) or [suggest an improvement](#).

---

[Create an Issue](#) [Edit This Page](#)

Page last modified on October 14, 2019 at 11:52 AM PST by [Fixing links in markdown for `logs` and `metrics` \(#16843\)](#) ([Page History](#))

[Edit This Page](#)

# Get a Shell to a Running Container

This page shows how to use `kubectl exec` to get a shell to a running Container.

- [Before you begin](#)
- [Getting a shell to a Container](#)
- [Writing the root page for nginx](#)
- [Running individual commands in a Container](#)
- [Opening a shell when a Pod has more than one Container](#)

- [What's next](#)

## Before you begin

You need to have a Kubernetes cluster, and the kubectl command-line tool must be configured to communicate with your cluster. 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`.

## Getting a shell to a Container

In this exercise, you create a Pod that has one Container. The Container runs the nginx image. Here is the configuration file for the Pod:

```
application/shell-demo.yaml  
  
apiVersion: v1  
kind: Pod  
metadata:  
  name: shell-demo  
spec:  
  volumes:  
    - name: shared-data  
      emptyDir: {}  
  containers:  
    - name: nginx  
      image: nginx  
      volumeMounts:  
        - name: shared-data  
          mountPath: /usr/share/nginx/html  
hostNetwork: true  
dnsPolicy: Default
```

Create the Pod:

```
kubectl apply -f https://k8s.io/examples/application/shell-  
demo.yaml
```

Verify that the Container is running:

```
kubectl get pod shell-demo
```

Get a shell to the running Container:

```
kubectl exec -it shell-demo -- /bin/bash
```

**Note:** The double dash symbol "--" is used to separate the arguments you want to pass to the command from the kubectl arguments.

In your shell, list the root directory:

```
root@shell-demo:/# ls /
```

In your shell, experiment with other commands. Here are some examples:

```
root@shell-demo:/# ls /
root@shell-demo:/# cat /proc/mounts
root@shell-demo:/# cat /proc/1/maps
root@shell-demo:/# apt-get update
root@shell-demo:/# apt-get install -y tcpdump
root@shell-demo:/# tcpdump
root@shell-demo:/# apt-get install -y lsof
root@shell-demo:/# lsof
root@shell-demo:/# apt-get install -y procps
root@shell-demo:/# ps aux
root@shell-demo:/# ps aux | grep nginx
```

## Writing the root page for nginx

Look again at the configuration file for your Pod. The Pod has an empty Dir volume, and the Container mounts the volume at /usr/share/nginx/html.

In your shell, create an `index.html` file in the `/usr/share/nginx/html` directory:

```
root@shell-demo:/# echo Hello shell demo > /usr/share/nginx/
html/index.html
```

In your shell, send a GET request to the nginx server:

```
root@shell-demo:/# apt-get update
root@shell-demo:/# apt-get install curl
root@shell-demo:/# curl localhost
```

The output shows the text that you wrote to the `index.html` file:

```
Hello shell demo
```

When you are finished with your shell, enter `exit`.

# Running individual commands in a Container

In an ordinary command window, not your shell, list the environment variables in the running Container:

```
kubectl exec shell-demo env
```

Experiment running other commands. Here are some examples:

```
kubectl exec shell-demo ps aux  
kubectl exec shell-demo ls /  
kubectl exec shell-demo cat /proc/1/mounts
```

## Opening a shell when a Pod has more than one Container

If a Pod has more than one Container, use `--container` or `-c` to specify a Container in the `kubectl exec` command. For example, suppose you have a Pod named `my-pod`, and the Pod has two containers named `main-app` and `helper-app`. The following command would open a shell to the `main-app` Container.

```
kubectl exec -it my-pod --container main-app -- /bin/bash
```

## What's next

- [kubectl exec](#)

## Feedback

Was this page helpful?

Yes No

Thanks for the feedback. If you have a specific, answerable question about how to use Kubernetes, ask it on [Stack Overflow](#). Open an issue in the GitHub repo if you want to [report a problem](#) or [suggest an improvement](#).

---

[Create an Issue](#) [Edit This Page](#)

Page last modified on March 25, 2019 at 5:06 PM PST by [Official 1.14](#)  
[Release Docs \(#13174\)](#) ([Page History](#))

[Edit This Page](#)

# Logging Using Elasticsearch and Kibana

On the Google Compute Engine (GCE) platform, the default logging support targets [Stackdriver Logging](#), which is described in detail in the [Logging With Stackdriver Logging](#).

This article describes how to set up a cluster to ingest logs into [Elasticsearch](#) and view them using [Kibana](#), as an alternative to Stackdriver Logging when running on GCE.

**Note:** You cannot automatically deploy Elasticsearch and Kibana in the Kubernetes cluster hosted on Google Kubernetes Engine. You have to deploy them manually.

- [What's next](#)

To use Elasticsearch and Kibana for cluster logging, you should set the following environment variable as shown below when creating your cluster with kube-up.sh:

```
KUBE_LOGGING_DESTINATION=elasticsearch
```

You should also ensure that `KUBE_ENABLE_NODE_LOGGING=true` (which is the default for the GCE platform).

Now, when you create a cluster, a message will indicate that the Fluentd log collection daemons that run on each node will target Elasticsearch:

```
cluster/kube-up.sh
```

```
...
Project: kubernetes-satnam
Zone: us-central1-b
... calling kube-up
Project: kubernetes-satnam
Zone: us-central1-b
+++ Staging server tars to Google Storage: gs://kubernetes-
staging-e6d0e81793-devel
+++ kubernetes-server-linux-amd64.tar.gz uploaded (sha1 =
6987c098277871b6d69623141276924ab687f89d)
+++ kubernetes-salt.tar.gz uploaded (sha1 =
bdfc83ed6b60fa9e3bff9004b542fcfc643464cd0)
Looking for already existing resources
Starting master and configuring firewalls
Created [https://www.googleapis.com/compute/v1/projects/
kubernetes-satnam/zones/us-central1-b/disks/kubernetes-
master-pd].
NAME          ZONE      SIZE_GB TYPE    STATUS
kubernetes-master-pd us-central1-b 20      pd-ssd  READY
Created [https://www.googleapis.com/compute/v1/projects/
kubernetes-satnam/regions/us-central1/addresses/kubernetes-
master-ip].
+++ Logging using Fluentd to elasticsearch
```

The per-node Fluentd pods, the Elasticsearch pods, and the Kibana pods should all be running in the kube-system namespace soon after the cluster comes to life.

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

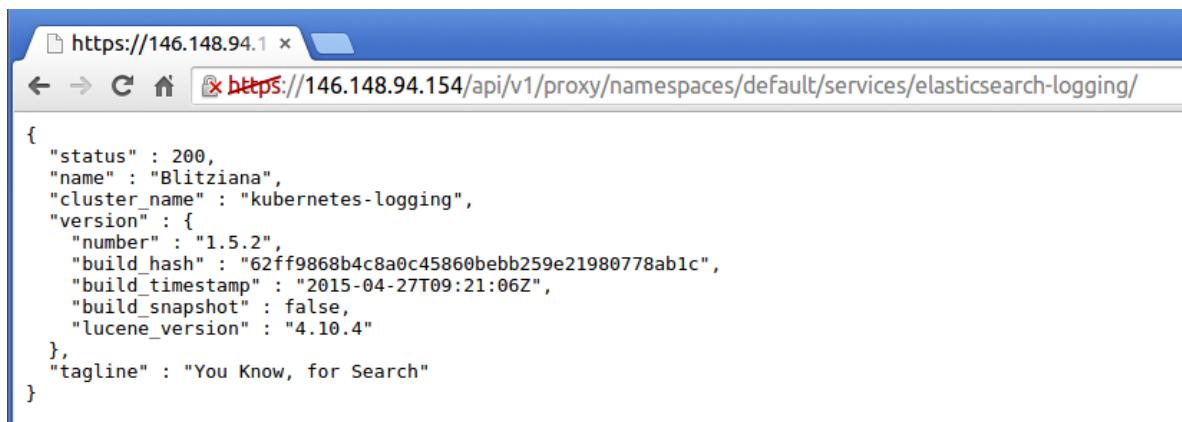
NAME	READY
STATUS	RESTARTS AGE

elasticsearch-logging-v1-78nog	1/1
Running 0 2h	
elasticsearch-logging-v1-nj2nb	1/1
Running 0 2h	
fluentd-elasticsearch-kubernetes-node-5oq0	1/1
Running 0 2h	
fluentd-elasticsearch-kubernetes-node-6896	1/1
Running 0 2h	
fluentd-elasticsearch-kubernetes-node-l1lds	1/1
Running 0 2h	
fluentd-elasticsearch-kubernetes-node-lz9j	1/1
Running 0 2h	
kibana-logging-v1-bhp08	1/1
Running 0 2h	
kube-dns-v3-7r1l9	3/3
Running 0 2h	
monitoring-heapster-v4-yl332	1/1
Running 1 2h	
monitoring-influx-grafana-v1-o79xf	2/2
Running 0 2h	

The fluentd-elasticsearch pods gather logs from each node and send them to the elasticsearch-logging pods, which are part of a [service](#) named elasticsearch-logging. These Elasticsearch pods store the logs and expose them via a REST API. The kibana-logging pod provides a web UI for reading the logs stored in Elasticsearch, and is part of a service named kibana-logging.

The Elasticsearch and Kibana services are both in the kube-system namespace and are not directly exposed via a publicly reachable IP address. To reach them, follow the instructions for [Accessing services running in a cluster](#).

If you try accessing the elasticsearch-logging service in your browser, you'll see a status page that looks something like this:



```

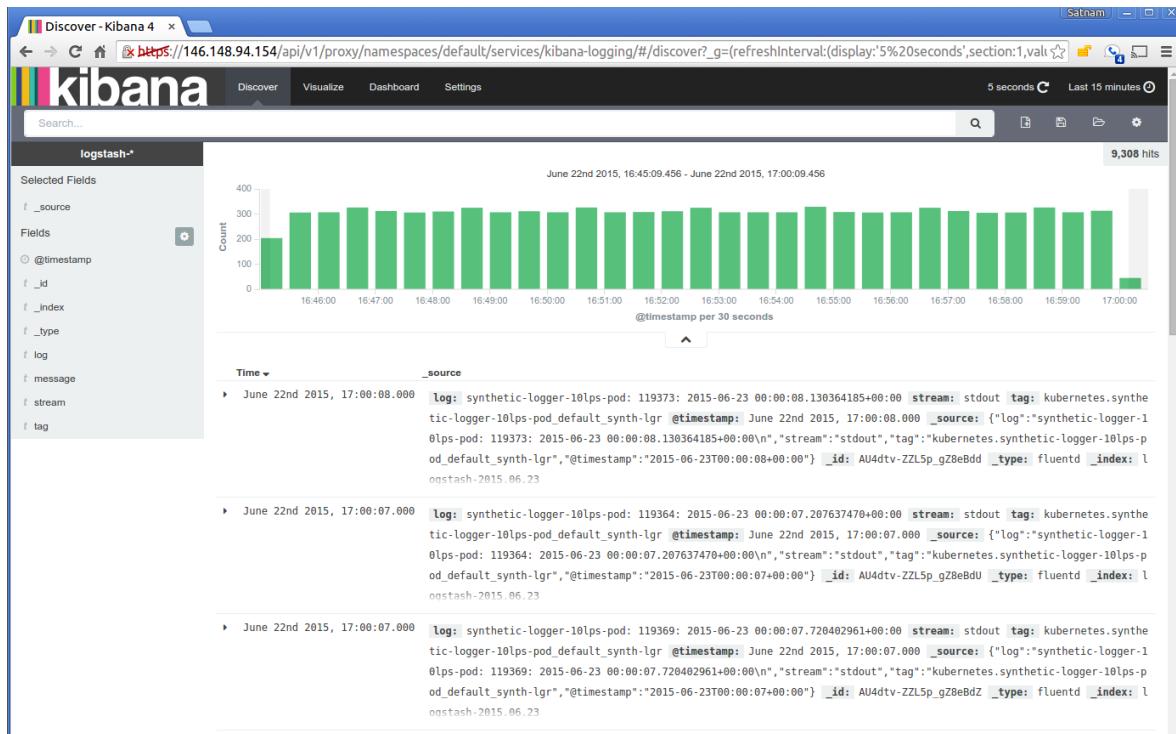
https://146.148.94.154/api/v1/proxy/namespaces/default/services/elasticsearch-logging/
{
  "status" : 200,
  "name" : "Blitziana",
  "cluster_name" : "kubernetes-logging",
  "version" : {
    "number" : "1.5.2",
    "build_hash" : "62ff9868b4c8a0c45860bebb259e21980778ab1c",
    "build_timestamp" : "2015-04-27T09:21:06Z",
    "build_snapshot" : false,
    "lucene_version" : "4.10.4"
  },
  "tagline" : "You Know, for Search"
}

```

You can now type Elasticsearch queries directly into the browser, if you'd like. See [Elasticsearch's documentation](#) for more details on how to do so.

Alternatively, you can view your cluster's logs using Kibana (again using the [instructions for accessing a service running in the cluster](#)). The first time you visit the Kibana URL you will be presented with a page that asks you to configure your view of the ingested logs. Select the option for timeseries values and select @timestamp. On the following page select the Discover tab and then you should be able to see the ingested logs. You can set the refresh interval to 5 seconds to have the logs regularly refreshed.

Here is a typical view of ingested logs from the Kibana viewer:



## What's next

Kibana opens up all sorts of powerful options for exploring your logs! For some ideas on how to dig into it, check out [Kibana's documentation](#).

## Feedback

Was this page helpful?

Yes No

Thanks for the feedback. If you have a specific, answerable question about how to use Kubernetes, ask it on [Stack Overflow](#). Open an issue in the GitHub repo if you want to [report a problem](#) or [suggest an improvement](#).

[Edit This Page](#)

Page last modified on March 07, 2019 at 3:01 PM PST by [Code snippets shouldn't include the command prompt \(#12779\)](#) ([Page History](#))

[Edit This Page](#)

# Logging Using Stackdriver

Before reading this page, it's highly recommended to familiarize yourself with the [overview of logging in Kubernetes](#).

**Note:** By default, Stackdriver logging collects only your container's standard output and standard error streams. To collect any logs your application writes to a file (for example), see the [sidecar approach](#) in the Kubernetes logging overview.

- [Deploying](#)

- [Verifying your Logging Agent Deployment](#)
- [Viewing logs](#)
- [Configuring Stackdriver Logging Agents](#)

# Deploying

To ingest logs, you must deploy the Stackdriver Logging agent to each node in your cluster. The agent is a configured fluentd instance, where the configuration is stored in a ConfigMap and the instances are managed using a Kubernetes DaemonSet. The actual deployment of the ConfigMap and DaemonSet for your cluster depends on your individual cluster setup.

## Deploying to a new cluster

### Google Kubernetes Engine

Stackdriver is the default logging solution for clusters deployed on Google Kubernetes Engine. Stackdriver Logging is deployed to a new cluster by default unless you explicitly opt-out.

### Other platforms

To deploy Stackdriver Logging on a *new* cluster that you're creating using `kube-up.sh`, do the following:

1. Set the `KUBE_LOGGING_DESTINATION` environment variable to `gcp`.
2. **If not running on GCE**, include the `beta.kubernetes.io/fluentd-ds-ready=true` in the `KUBE_NODE_LABELS` variable.

Once your cluster has started, each node should be running the Stackdriver Logging agent. The DaemonSet and ConfigMap are configured as addons. If you're not using `kube-up.sh`, consider starting a cluster without a pre-configured logging solution and then deploying Stackdriver Logging agents to the running cluster.

**Warning:** The Stackdriver logging daemon has known issues on platforms other than Google Kubernetes Engine. Proceed at your own risk.

## Deploying to an existing cluster

1. Apply a label on each node, if not already present.

The Stackdriver Logging agent deployment uses node labels to determine to which nodes it should be allocated. These labels were introduced to distinguish nodes with the Kubernetes version 1.6 or higher. If the cluster was created with Stackdriver Logging configured and node has version 1.5.X or lower, it will have fluentd as static pod. Node cannot have more than one instance of fluentd, therefore only apply labels to the nodes that don't have fluentd pod

allocated already. You can ensure that your node is labelled properly by running `kubectl describe` as follows:

```
kubectl describe node $NODE_NAME
```

The output should be similar to this:

Name:	NODE_NAME
Role:	
Labels:	beta.kubernetes.io/fluentd-ds-ready=true
...	

Ensure that the output contains the label `beta.kubernetes.io/fluentd-ds-ready=true`. If it is not present, you can add it using the `kubectl label` command as follows:

```
kubectl label node $NODE_NAME beta.kubernetes.io/fluentd-ds-ready=true
```

**Note:** If a node fails and has to be recreated, you must re-apply the label to the recreated node. To make this easier, you can use Kubelet's command-line parameter for applying node labels in your node startup script.

2. Deploy a ConfigMap with the logging agent configuration by running the following command:

```
kubectl apply -f https://k8s.io/examples/debug/fluentd-gcp-configmap.yaml
```

The command creates the ConfigMap in the default namespace. You can download the file manually and change it before creating the ConfigMap object.

3. Deploy the logging agent DaemonSet by running the following command:

```
kubectl apply -f https://k8s.io/examples/debug/fluentd-gcp-ds.yaml
```

You can download and edit this file before using it as well.

## Verifying your Logging Agent Deployment

After Stackdriver DaemonSet is deployed, you can discover logging agent deployment status by running the following command:

```
kubectl get ds --all-namespaces
```

If you have 3 nodes in the cluster, the output should look similar to this:

NAMESPACE	NAME	DESIRED	CURRENT	AGE
READY	NODE-SELECTOR			
...				
default	fluentd-gcp-v2.0	3	3	
3	beta.kubernetes.io/fluentd-ds-ready=true			5m
...				

To understand how logging with Stackdriver works, consider the following synthetic log generator pod specification [counter-pod.yaml](#):

#### [debug/counter-pod.yaml](#)

```
apiVersion: v1
kind: Pod
metadata:
  name: counter
spec:
  containers:
    - name: count
      image: busybox
      args: [/bin/sh, -c,
              'i=0; while true; do echo "$i: $(date)"; i=$((i+1)); sleep 1; done']
```

This pod specification has one container that runs a bash script that writes out the value of a counter and the datetime once per second, and runs indefinitely. Let's create this pod in the default namespace.

```
kubectl apply -f https://k8s.io/examples/debug/counter-pod.yaml
```

You can observe the running pod:

```
kubectl get pods
```

NAME	READY		
STATUS	RESTARTS	AGE	
counter			1/1
Running	0	5m	

For a short period of time you can observe the 'Pending' pod status, because the kubelet has to download the container image first. When the pod status changes to Running you can use the `kubectl logs` command to view the output of this counter pod.

```
kubectl logs counter
```

```
0: Mon Jan  1 00:00:00 UTC 2001
1: Mon Jan  1 00:00:01 UTC 2001
2: Mon Jan  1 00:00:02 UTC 2001
...
```

As described in the logging overview, this command fetches log entries from the container log file. If the container is killed and then restarted by Kubernetes, you can still access logs from the previous container. However, if the pod is evicted from the node, log files are lost. Let's demonstrate this by deleting the currently running counter container:

```
kubectl delete pod counter
```

```
pod "counter" deleted
```

and then recreating it:

```
kubectl create -f https://k8s.io/examples/debug/counter-pod.yaml
```

```
pod/counter created
```

After some time, you can access logs from the counter pod again:

```
kubectl logs counter
```

```
0: Mon Jan  1 00:01:00 UTC 2001
1: Mon Jan  1 00:01:01 UTC 2001
2: Mon Jan  1 00:01:02 UTC 2001
...
```

As expected, only recent log lines are present. However, for a real-world application you will likely want to be able to access logs from all containers, especially for the debug purposes. This is exactly when the previously enabled Stackdriver Logging can help.

## Viewing logs

Stackdriver Logging agent attaches metadata to each log entry, for you to use later in queries to select only the messages you're interested in: for example, the messages from a particular pod.

The most important pieces of metadata are the resource type and log name. The resource type of a container log is `container`, which is named `GKE Containers` in the UI (even if the Kubernetes cluster is not on Google Kubernetes Engine). The log name is the name of the container, so that if you have a pod with two containers, named `container_1` and `container_2` in the spec, their logs will have log names `container_1` and `container_2` respectively.

System components have resource type `compute`, which is named `GCE VM Instance` in the interface. Log names for system components are fixed. For a Google Kubernetes Engine node, every log entry from a system component has one of the following log names:

- `docker`
- `kubelet`
- `kube-proxy`

You can learn more about viewing logs on [the dedicated Stackdriver page](#).

One of the possible ways to view logs is using the [gcloud logging](#) command line interface from the [Google Cloud SDK](#). It uses Stackdriver Logging [filtering syntax](#) to query specific logs. For example, you can run the following command:

```
gcloud beta logging read 'logName="projects/$YOUR_PROJECT_ID/logs/count"' --format json | jq '.[].textPayload'
```

```
...
"2: Mon Jan 1 00:01:02 UTC 2001\n"
"1: Mon Jan 1 00:01:01 UTC 2001\n"
"0: Mon Jan 1 00:01:00 UTC 2001\n"
...
"2: Mon Jan 1 00:00:02 UTC 2001\n"
"1: Mon Jan 1 00:00:01 UTC 2001\n"
"0: Mon Jan 1 00:00:00 UTC 2001\n"
```

As you can see, it outputs messages for the count container from both the first and second runs, despite the fact that the kubelet already deleted the logs for the first container.

## Exporting logs

You can export logs to [Google Cloud Storage](#) or to [BigQuery](#) to run further analysis. Stackdriver Logging offers the concept of sinks, where you can specify the destination of log entries. More information is available on the Stackdriver [Exporting Logs page](#).

# Configuring Stackdriver Logging Agents

Sometimes the default installation of Stackdriver Logging may not suit your needs, for example:

- You may want to add more resources because default performance doesn't suit your needs.
- You may want to introduce additional parsing to extract more metadata from your log messages, like severity or source code reference.
- You may want to send logs not only to Stackdriver or send it to Stackdriver only partially.

In this case you need to be able to change the parameters of DaemonSet and ConfigMap.

## Prerequisites

If you're using GKE and Stackdriver Logging is enabled in your cluster, you cannot change its configuration, because it's managed and

supported by GKE. However, you can disable the default integration and deploy your own.

**Note:** You will have to support and maintain a newly deployed configuration yourself: update the image and configuration, adjust the resources and so on.

To disable the default logging integration, use the following command:

```
gcloud beta container clusters update --logging-service=none  
CLUSTER
```

You can find notes on how to then install Stackdriver Logging agents into a running cluster in the [Deploying section](#).

## Changing DaemonSet parameters

When you have the Stackdriver Logging DaemonSet in your cluster, you can just modify the template field in its spec, daemonset controller will update the pods for you. For example, let's assume you've just installed the Stackdriver Logging as described above. Now you want to change the memory limit to give fluentd more memory to safely process more logs.

Get the spec of DaemonSet running in your cluster:

```
kubectl get ds fluentd-gcp-v2.0 --namespace kube-system -o  
yaml > fluentd-gcp-ds.yaml
```

Then edit resource requirements in the spec file and update the Daemon Set object in the apiserver using the following command:

```
kubectl replace -f fluentd-gcp-ds.yaml
```

After some time, Stackdriver Logging agent pods will be restarted with the new configuration.

## Changing fluentd parameters

Fluentd configuration is stored in the ConfigMap object. It is effectively a set of configuration files that are merged together. You can learn about fluentd configuration on the [official site](#).

Imagine you want to add a new parsing logic to the configuration, so that fluentd can understand default Python logging format. An appropriate fluentd filter looks similar to this:

```
<filter reform.*>  
  type parser  
  format /^(?<severity>\w):(?:<logger_name>\w):(?:<log>.*)>/  
  reserve_data true  
  suppress_parse_error_log true
```

```
    key_name log  
</filter>
```

Now you have to put it in the configuration and make Stackdriver Logging agents pick it up. Get the current version of the Stackdriver Logging ConfigMap in your cluster by running the following command:

```
kubectl get cm fluentd-gcp-config --namespace kube-system -o yaml > fluentd-gcp-configmap.yaml
```

Then in the value of the key `containers.input.conf` insert a new filter right after the `source` section.

**Note:** Order is important.

Updating ConfigMap in the apiserver is more complicated than updating DaemonSet. It's better to consider ConfigMap to be immutable. Then, in order to update the configuration, you should create ConfigMap with a new name and then change DaemonSet to point to it using [guide above](#).

## Adding fluentd plugins

Fluentd is written in Ruby and allows to extend its capabilities using [plugins](#). If you want to use a plugin, which is not included in the default Stackdriver Logging container image, you have to build a custom image. Imagine you want to add Kafka sink for messages from a particular container for additional processing. You can re-use the default [container image sources](#) with minor changes:

- Change Makefile to point to your container repository, e.g. `PREFIX =gcr.io/<your-project-id>`.
- Add your dependency to the Gemfile, for example `gem 'fluent-plugin-kafka'`.

Then run `make build push` from this directory. After updating DaemonSet to pick up the new image, you can use the plugin you installed in the fluentd configuration.

## Feedback

Was this page helpful?

Yes No

Thanks for the feedback. If you have a specific, answerable question about how to use Kubernetes, ask it on [Stack Overflow](#). Open an issue in the GitHub repo if you want to [report a problem](#) or [suggest an improvement](#).

[Create an Issue](#) [Edit This Page](#)

Page last modified on March 25, 2019 at 5:06 PM PST by [Official 1.14 Release Docs \(#13174\)](#) ([Page History](#))

[Edit This Page](#)

# Monitor Node Health

*Node problem detector* is a [DaemonSet](#) monitoring the node health. It collects node problems from various daemons and reports them to the apiserver as [NodeCondition](#) and [Event](#).

It supports some known kernel issue detection now, and will detect more and more node problems over time.

Currently Kubernetes won't take any action on the node conditions and events generated by node problem detector. In the future, a remedy system could be introduced to deal with node problems.

See more information [here](#).

- [Before you begin](#)
- [Limitations](#)
- [Enable/Disable in GCE cluster](#)
- [Use in Other Environment](#)
- [Overwrite the Configuration](#)
- [Kernel Monitor](#)
- [Caveats](#)

## Before you begin

You need to have a Kubernetes cluster, and the kubectl command-line tool must be configured to communicate with your cluster. 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`.

## Limitations

- The kernel issue detection of node problem detector only supports file based kernel log now. It doesn't support log tools like journald.
- The kernel issue detection of node problem detector has assumption on kernel log format, and now it only works on Ubuntu and Debian. However, it is easy to extend it to [support other log format](#).

## Enable/Disable in GCE cluster

Node problem detector is [running as a cluster addon](#) enabled by default in the gce cluster.

You can enable/disable it by setting the environment variable `KUBE_ENABLE_NODE_PROBLEM_DETECTOR` before `kube-up.sh`.

## Use in Other Environment

To enable node problem detector in other environment outside of GCE, you can use either `kubectl` or `addon pod`.

### Kubectl

This is the recommended way to start node problem detector outside of GCE. It provides more flexible management, such as overwriting the

default configuration to fit it into your environment or detect customized node problems.

- **Step 1: node-problem-detector.yaml:**

[\*\*debug/node-problem-detector.yaml\*\*](#)

```
apiVersion: apps/v1
kind: DaemonSet
metadata:
  name: node-problem-detector-v0.1
  namespace: kube-system
  labels:
    k8s-app: node-problem-detector
    version: v0.1
    kubernetes.io/cluster-service: "true"
spec:
  selector:
    matchLabels:
      k8s-app: node-problem-detector
      version: v0.1
      kubernetes.io/cluster-service: "true"
  template:
    metadata:
      labels:
        k8s-app: node-problem-detector
        version: v0.1
        kubernetes.io/cluster-service: "true"
    spec:
      hostNetwork: true
      containers:
        - name: node-problem-detector
          image: k8s.gcr.io/node-problem-detector:v0.1
          securityContext:
            privileged: true
          resources:
            limits:
              cpu: "200m"
              memory: "100Mi"
            requests:
              cpu: "20m"
              memory: "20Mi"
          volumeMounts:
            - name: log
              mountPath: /log
              readOnly: true
          volumes:
            - name: log
              hostPath:
                path: /var/log/
```

**Notice that you should make sure the system log directory is right for your OS distro.**

- **Step 2:** Start node problem detector with kubectl:

```
kubectl apply -f https://k8s.io/examples/debug/node-problem-detector.yaml
```

## Addon Pod

This is for those who have their own cluster bootstrap solution, and don't need to overwrite the default configuration. They could leverage the addon pod to further automate the deployment.

Just create `node-problem-detector.yaml`, and put it under the addon pods directory `/etc/kubernetes/addons/node-problem-detector` on master node.

## Overwrite the Configuration

The [default configuration](#) is embedded when building the Docker image of node problem detector.

However, you can use [ConfigMap](#) to overwrite it following the steps:

- **Step 1:** Change the config files in `config/`.
- **Step 2:** Create the ConfigMap `node-problem-detector-config` with `kubectl create configmap node-problem-detector-config --from-file=config/`.
- **Step 3:** Change the `node-problem-detector.yaml` to use the ConfigMap:

```
apiVersion: apps/v1
kind: DaemonSet
metadata:
  name: node-problem-detector-v0.1
  namespace: kube-system
  labels:
    k8s-app: node-problem-detector
    version: v0.1
    kubernetes.io/cluster-service: "true"
spec:
  selector:
    matchLabels:
      k8s-app: node-problem-detector
      version: v0.1
      kubernetes.io/cluster-service: "true"
  template:
    metadata:
      labels:
        k8s-app: node-problem-detector
        version: v0.1
        kubernetes.io/cluster-service: "true"
    spec:
      hostNetwork: true
      containers:
        - name: node-problem-detector
          image: k8s.gcr.io/node-problem-detector:v0.1
          securityContext:
            privileged: true
          resources:
            limits:
              cpu: "200m"
              memory: "100Mi"
            requests:
              cpu: "20m"
              memory: "20Mi"
          volumeMounts:
            - name: log
              mountPath: /log
              readOnly: true
            - name: config # Overwrite the config/ directory with ConfigMap volume
              mountPath: /config
              readOnly: true
          volumes:
            - name: log
              hostPath:
                path: /var/log/
            - name: config # Define ConfigMap volume
              configMap:
                name: node-problem-detector-config
```

- **Step 4:** Re-create the node problem detector with the new yaml file:

```
kubectl delete -f https://k8s.io/examples/debug/node-
problem-detector.yaml # If you have a node-problem-
detector running
kubectl apply -f https://k8s.io/examples/debug/node-
problem-detector-configmap.yaml
```

**Notice that this approach only applies to node problem detector started with kubectl.**

For node problem detector running as cluster addon, because addon manager doesn't support ConfigMap, configuration overwriting is not supported now.

## Kernel Monitor

*Kernel Monitor* is a problem daemon in node problem detector. It monitors kernel log and detects known kernel issues following predefined rules.

The Kernel Monitor matches kernel issues according to a set of predefined rule list in [config/kernel-monitor.json](#). The rule list is extensible, and you can always extend it by overwriting the configuration.

### Add New NodeConditions

To support new node conditions, you can extend the `conditions` field in `config/kernel-monitor.json` with new condition definition:

```
{
  "type": "NodeConditionType",
  "reason": "CamelCaseDefaultNodeConditionReason",
  "message": "arbitrary default node condition message"
}
```

### Detect New Problems

To detect new problems, you can extend the `rules` field in `config/kernel-monitor.json` with new rule definition:

```
{
  "type": "temporary/permanent",
  "condition": "NodeConditionOfPermanentIssue",
  "reason": "CamelCaseShortReason",
  "message": "regexp matching the issue in the kernel log"
}
```

## Change Log Path

Kernel log in different OS distros may locate in different path. The `log` field in `config/kernel-monitor.json` is the log path inside the container. You can always configure it to match your OS distro.

## Support Other Log Format

Kernel monitor uses [Translator](#) plugin to translate kernel log the internal data structure. It is easy to implement a new translator for a new log format.

## Caveats

It is recommended to run the node problem detector in your cluster to monitor the node health. However, you should be aware that this will introduce extra resource overhead on each node. Usually this is fine, because:

- The kernel log is generated relatively slowly.
- Resource limit is set for node problem detector.
- Even under high load, the resource usage is acceptable. (see [benchmark result](#))

## Feedback

Was this page helpful?

Yes No

Thanks for the feedback. If you have a specific, answerable question about how to use Kubernetes, ask it on [Stack Overflow](#). Open an issue in the GitHub repo if you want to [report a problem](#) or [suggest an improvement](#).

---

[Create an Issue](#) [Edit This Page](#)

Page last modified on June 12, 2019 at 5:27 PM PST by [Restructure the left navigation pane of setup \(#14826\)](#) ([Page History](#))

[Edit This Page](#)

# Resource metrics pipeline

Resource usage metrics, such as container CPU and memory usage, are available in Kubernetes through the Metrics API. These metrics can be either accessed directly by user, for example by using `kubectl top` command, or used by a controller in the cluster, e.g. Horizontal Pod Autoscaler, to make decisions.

- [The Metrics API](#)
- [Measuring Resource Usage](#)
- [Metrics Server](#)

# The Metrics API

Through the Metrics API you can get the amount of resource currently used by a given node or a given pod. This API doesn't store the metric values, so it's not possible for example to get the amount of resources used by a given node 10 minutes ago.

The API is no different from any other API:

- it is discoverable through the same endpoint as the other Kubernetes APIs under `/apis/metrics.k8s.io/` path
- it offers the same security, scalability and reliability guarantees

The API is defined in [k8s.io/metrics](#) repository. You can find more information about the API there.

**Note:** The API requires metrics server to be deployed in the cluster. Otherwise it will be not available.

## Measuring Resource Usage

### CPU

CPU is reported as the average usage, in [CPU cores](#), over a period of time. This value is derived by taking a rate over a cumulative CPU counter provided by the kernel (in both Linux and Windows kernels). The kubelet chooses the window for the rate calculation.

### Memory

Memory is reported as the working set, in bytes, at the instant the metric was collected. In an ideal world, the "working set" is the amount of memory in-use that cannot be freed under memory pressure. However, calculation of the working set varies by host OS, and generally makes heavy use of heuristics to produce an estimate. It includes all anonymous (non-file-backed) memory since kubernetes does not support swap. The metric typically also includes some cached (file-backed) memory, because the host OS cannot always reclaim such pages.

## Metrics Server

[Metrics Server](#) is a cluster-wide aggregator of resource usage data. It is deployed by default in clusters created by `kube-up.sh` script as a Deployment object. If you use a different Kubernetes setup mechanism you can deploy it using the provided [deployment yaml](#).

Metric server collects metrics from the Summary API, exposed by [Kubelet](#) on each node.

Metrics Server registered in the main API server through [Kubernetes aggregator](#).

Learn more about the metrics server in [the design doc](#).

## Feedback

Was this page helpful?

Yes No

Thanks for the feedback. If you have a specific, answerable question about how to use Kubernetes, ask it on [Stack Overflow](#). Open an issue in the GitHub repo if you want to [report a problem](#) or [suggest an improvement](#).

---

[Edit This Page](#)

# Tools for Monitoring Resources

To scale an application and provide a reliable service, you need to understand how the application behaves when it is deployed. You can examine application performance in a Kubernetes cluster by examining the containers, [pods](#), [services](#), and the characteristics of the overall cluster. Kubernetes provides detailed information about an application's resource usage at each of these levels. This information allows you to evaluate your application's performance and where bottlenecks can be removed to improve overall performance.

- [Resource metrics pipeline](#)
- [Full metrics pipeline](#)

In Kubernetes, application monitoring does not depend on a single monitoring solution. On new clusters, you can use [resource metrics](#) or [full metrics](#) pipelines to collect monitoring statistics.

## Resource metrics pipeline

The resource metrics pipeline provides a limited set of metrics related to cluster components such as the [Horizontal Pod Autoscaler](#) controller, as well as the `kubectl top` utility. These metrics are collected by the lightweight, short-term, in-memory [metrics-server](#) and are exposed via the `metrics.k8s.io` API.

metrics-server discovers all nodes on the cluster and queries each node's [kubelet](#) for CPU and memory usage. The kubelet acts as a bridge between the Kubernetes master and the nodes, managing the pods and containers running on a machine. The kubelet translates each pod into its constituent containers and fetches individual container usage statistics from the container runtime through the container runtime interface. The kubelet fetches this information from the integrated cAdvisor for the legacy Docker integration. It then exposes the aggregated pod resource usage statistics through the metrics-server Resource Metrics API. This API is served at `/metrics/resource/v1beta1` on the kubelet's authenticated and read-only ports.

## Full metrics pipeline

A full metrics pipeline gives you access to richer metrics. Kubernetes can respond to these metrics by automatically scaling or adapting the cluster based on its current state, using mechanisms such as the Horizontal Pod Autoscaler. The monitoring pipeline fetches metrics from the kubelet and then exposes them to Kubernetes via an adapter

by implementing either the `custom.metrics.k8s.io` or `external.metrics.k8s.io` API.

[Prometheus](#), a CNCF project, can natively monitor Kubernetes, nodes, and Prometheus itself. Full metrics pipeline projects that are not part of the CNCF are outside the scope of Kubernetes documentation.

## Feedback

Was this page helpful?

Yes No

Thanks for the feedback. If you have a specific, answerable question about how to use Kubernetes, ask it on [Stack Overflow](#). Open an issue in the GitHub repo if you want to [report a problem](#) or [suggest an improvement](#).

---

[Create an Issue](#) [Edit This Page](#)

Page last modified on August 20, 2019 at 3:23 AM PST by [Add link to HorzPodAuto; add that there are other full pipelines \(#15863\)](#) ([Page History](#))

[Edit This Page](#)

# Troubleshoot Applications

This guide is to help users debug applications that are deployed into Kubernetes and not behaving correctly. This is *not* a guide for people who want to debug their cluster. For that you should check out [this guide](#).

- [Diagnosing the problem](#)
- [What's next](#)

## Diagnosing the problem

The first step in troubleshooting is triage. What is the problem? Is it your Pods, your Replication Controller or your Service?

- [Debugging Pods](#)
- [Debugging Replication Controllers](#)
- [Debugging Services](#)

## Debugging Pods

The first step in debugging a Pod is taking a look at it. Check the current state of the Pod and recent events with the following command:

```
kubectl describe pods ${POD_NAME}
```

Look at the state of the containers in the pod. Are they all Running? Have there been recent restarts?

Continue debugging depending on the state of the pods.

## My pod stays pending

If a Pod is stuck in Pending it means that it can not be scheduled onto a node. Generally this is because there are insufficient resources of one type or another that prevent scheduling. Look at the output of the `kube ctl describe ...` command above. There should be messages from the scheduler about why it can not schedule your pod. Reasons include:

- **You don't have enough resources:** You may have exhausted the supply of CPU or Memory in your cluster, in this case you need to delete Pods, adjust resource requests, or add new nodes to your cluster. See [Compute Resources document](#) for more information.

- **You are using hostPort:** When you bind a Pod to a hostPort there are a limited number of places that pod can be scheduled. In most cases, hostPort is unnecessary, try using a Service object to expose your Pod. If you do require hostPort then you can only schedule as many Pods as there are nodes in your Kubernetes cluster.

## My pod stays waiting

If a Pod is stuck in the Waiting state, then it has been scheduled to a worker node, but it can't run on that machine. Again, the information from `kubectl describe ...` should be informative. The most common cause of Waiting pods is a failure to pull the image. There are three things to check:

- Make sure that you have the name of the image correct.
- Have you pushed the image to the repository?
- Run a manual `docker pull <image>` on your machine to see if the image can be pulled.

## My pod is crashing or otherwise unhealthy

First, take a look at the logs of the current container:

```
kubectl logs ${POD_NAME} ${CONTAINER_NAME}
```

If your container has previously crashed, you can access the previous container's crash log with:

```
kubectl logs --previous ${POD_NAME} ${CONTAINER_NAME}
```

Alternately, you can run commands inside that container with `exec`:

```
kubectl exec ${POD_NAME} -c ${CONTAINER_NAME} -- ${CMD} ${ARG1} ${ARG2} ... ${ARGN}
```

**Note:** `-c ${CONTAINER_NAME}` is optional. You can omit it for Pods that only contain a single container.

As an example, to look at the logs from a running Cassandra pod, you might run

```
kubectl exec cassandra -- cat /var/log/cassandra/system.log
```

If none of these approaches work, you can find the host machine that the pod is running on and SSH into that host, but this should generally not be necessary given tools in the Kubernetes API. Therefore, if you find yourself needing to ssh into a machine, please file a feature request on GitHub describing your use case and why these tools are insufficient.

## My pod is running but not doing what I told it to do

If your pod is not behaving as you expected, it may be that there was an error in your pod description (e.g. `mypod.yaml` file on your local machine), and that the error was silently ignored when you created the pod. Often a section of the pod description is nested incorrectly, or a key name is typed incorrectly, and so the key is ignored. For example, if you misspelled `command` as `commnd` then the pod will be created but will not use the command line you intended it to use.

The first thing to do is to delete your pod and try creating it again with the `--validate` option. For example, run `kubectl apply --validate -f mypod.yaml`. If you misspelled `command` as `commnd` then will give an error like this:

```
I0805 10:43:25.129850    46757 schema.go:126] unknown field:  
commnd  
I0805 10:43:25.129973    46757 schema.go:129] this may be a fa  
lse alarm, see https://github.com/kubernetes/kubernetes/issues/6842  
pods/mypod
```

The next thing to check is whether the pod on the apiserver matches the pod you meant to create (e.g. in a yaml file on your local machine). For example, run `kubectl get pods/mypod -o yaml > mypod-on-apiserver.yaml` and then manually compare the original pod description, `mypod.yaml` with the one you got back from apiserver, `mypod-on-apiserver.yaml`. There will typically be some lines on the "apiserver" version that are not on the original version. This is expected. However, if there are lines on the original that are not on the apiserver version, then this may indicate a problem with your pod spec.

## Debugging Replication Controllers

Replication controllers are fairly straightforward. They can either create Pods or they can't. If they can't create pods, then please refer to the [instructions above](#) to debug your pods.

You can also use `kubectl describe rc ${CONTROLLER_NAME}` to introspect events related to the replication controller.

## Debugging Services

Services provide load balancing across a set of pods. There are several common problems that can make Services not work properly. The following instructions should help debug Service problems.

First, verify that there are endpoints for the service. For every Service object, the apiserver makes an `endpoints` resource available.

You can view this resource with:

```
kubectl get endpoints ${SERVICE_NAME}
```

Make sure that the endpoints match up with the number of containers that you expect to be a member of your service. For example, if your Service is for an nginx container with 3 replicas, you would expect to see three different IP addresses in the Service's endpoints.

## My service is missing endpoints

If you are missing endpoints, try listing pods using the labels that Service uses. Imagine that you have a Service where the labels are:

```
...  
spec:  
- selector:  
  name: nginx  
  type: frontend
```

You can use:

```
kubectl get pods --selector=name=nginx,type=frontend
```

to list pods that match this selector. Verify that the list matches the Pods that you expect to provide your Service.

If the list of pods matches expectations, but your endpoints are still empty, it's possible that you don't have the right ports exposed. If your service has a `containerPort` specified, but the Pods that are selected don't have that port listed, then they won't be added to the endpoints list.

Verify that the pod's `containerPort` matches up with the Service's `targetPort`

## Network traffic is not forwarded

If you can connect to the service, but the connection is immediately dropped, and there are endpoints in the endpoints list, it's likely that the proxy can't contact your pods.

There are three things to check:

- Are your pods working correctly? Look for restart count, and [debug pods](#).
- Can you connect to your pods directly? Get the IP address for the Pod, and try to connect directly to that IP.
- Is your application serving on the port that you configured? Kubernetes doesn't do port remapping, so if your application serves on 8080, the `containerPort` field needs to be 8080.

## What's next

If none of the above solves your problem, follow the instructions in [Debugging Service document](#) to make sure that your Service is running, has Endpoints, and your Pods are actually serving; you have DNS working, iptables rules installed, and kube-proxy does not seem to be misbehaving.

You may also visit [troubleshooting document](#) for more information.

## Feedback

Was this page helpful?

Yes No

Thanks for the feedback. If you have a specific, answerable question about how to use Kubernetes, ask it on [Stack Overflow](#). Open an issue in the GitHub repo if you want to [report a problem](#) or [suggest an improvement](#).

---

[Create an Issue](#) [Edit This Page](#)

Page last modified on November 26, 2019 at 6:31 AM PST by [Update debug-application.md \(#17066\)](#) ([Page History](#))

[Edit This Page](#)

# Troubleshoot Clusters

This doc is about cluster troubleshooting; we assume you have already ruled out your application as the root cause of the problem you are experiencing. See the [application troubleshooting guide](#) for tips on application debugging. You may also visit [troubleshooting document](#) for more information.

- [Listing your cluster](#)
- [Looking at logs](#)
- [A general overview of cluster failure modes](#)

# **Listing your cluster**

The first thing to debug in your cluster is if your nodes are all registered correctly.

Run

```
kubectl get nodes
```

And verify that all of the nodes you expect to see are present and that they are all in the Ready state.

# **Looking at logs**

For now, digging deeper into the cluster requires logging into the relevant machines. Here are the locations of the relevant log files. (note that on systemd-based systems, you may need to use `journalctl` instead)

## **Master**

- `/var/log/kube-apiserver.log` - API Server, responsible for serving the API
- `/var/log/kube-scheduler.log` - Scheduler, responsible for making scheduling decisions
- `/var/log/kube-controller-manager.log` - Controller that manages replication controllers

## **Worker Nodes**

- `/var/log/kubelet.log` - Kubelet, responsible for running containers on the node
- `/var/log/kube-proxy.log` - Kube Proxy, responsible for service load balancing

# **A general overview of cluster failure modes**

This is an incomplete list of things that could go wrong, and how to adjust your cluster setup to mitigate the problems.

## **Root causes:**

- VM(s) shutdown
- Network partition within cluster, or between cluster and users
- Crashes in Kubernetes software
- Data loss or unavailability of persistent storage (e.g. GCE PD or AWS EBS volume)
- Operator error, e.g. misconfigured Kubernetes software or application software

## **Specific scenarios:**

- Apiserver VM shutdown or apiserver crashing
  - Results
    - unable to stop, update, or start new pods, services, replication controller
    - existing pods and services should continue to work normally, unless they depend on the Kubernetes API
- Apiserver backing storage lost
  - Results
    - apiserver should fail to come up
    - kubelets will not be able to reach it but will continue to run the same pods and provide the same service proxying
    - manual recovery or recreation of apiserver state necessary before apiserver is restarted
- Supporting services (node controller, replication controller manager, scheduler, etc) VM shutdown or crashes
  - currently those are colocated with the apiserver, and their unavailability has similar consequences as apiserver
  - in future, these will be replicated as well and may not be co-located
  - they do not have their own persistent state
- Individual node (VM or physical machine) shuts down
  - Results
    - pods on that Node stop running
- Network partition
  - Results
    - partition A thinks the nodes in partition B are down; partition B thinks the apiserver is down. (Assuming the master VM ends up in partition A.)
- Kubelet software fault
  - Results
    - crashing kubelet cannot start new pods on the node
    - kubelet might delete the pods or not
    - node marked unhealthy
    - replication controllers start new pods elsewhere
- Cluster operator error
  - Results
    - loss of pods, services, etc
    - lost of apiserver backing store
    - users unable to read API
    - etc.

## **Mitigations:**

- Action: Use IaaS provider's automatic VM restarting feature for IaaS VMs
  - Mitigates: Apiserver VM shutdown or apiserver crashing
  - Mitigates: Supporting services VM shutdown or crashes

- Action: Use IaaS providers reliable storage (e.g. GCE PD or AWS EBS volume) for VMs with apiserver+etcd
  - Mitigates: Apiserver backing storage lost
- Action: Use [high-availability](#) configuration
  - Mitigates: Control plane node shutdown or control plane components (scheduler, API server, controller-manager) crashing
  - Will tolerate one or more simultaneous node or component failures
  - Mitigates: API server backing storage (i.e., etcd's data directory) lost
  - Assumes HA (highly-available) etcd configuration
- Action: Snapshot apiserver PDs/EBS-volumes periodically
  - Mitigates: Apiserver backing storage lost
  - Mitigates: Some cases of operator error
  - Mitigates: Some cases of Kubernetes software fault
- Action: use replication controller and services in front of pods
  - Mitigates: Node shutdown
  - Mitigates: Kubelet software fault
- Action: applications (containers) designed to tolerate unexpected restarts
  - Mitigates: Node shutdown
  - Mitigates: Kubelet software fault
- Action: [Multiple independent clusters](#) (and avoid making risky changes to all clusters at once)
  - Mitigates: Everything listed above.

## Feedback

Was this page helpful?

Yes No

Thanks for the feedback. If you have a specific, answerable question about how to use Kubernetes, ask it on [Stack Overflow](#). Open an issue in the GitHub repo if you want to [report a problem](#) or [suggest an improvement](#).

---

[Create an Issue](#) [Edit This Page](#)

Page last modified on November 07, 2019 at 10:16 PM PST by [Remove 'experimental' from high-availability, and ':' from etcd bullet \(#17254\)](#)  
[\(Page History\)](#)

[Edit This Page](#)

# Troubleshooting

Sometimes things go wrong. This guide is aimed at making them right.  
It has two sections:

- [Troubleshooting your application](#) - Useful for users who are deploying code into Kubernetes and wondering why it is not working.
- [Troubleshooting your cluster](#) - Useful for cluster administrators and people whose Kubernetes cluster is unhappy.

You should also check the known issues for the [release](#) you're using.

- [Getting help](#)
- [Help! My question isn't covered! I need help now!](#)

## Getting help

If your problem isn't answered by any of the guides above, there are variety of ways for you to get help from the Kubernetes team.

### Questions

The documentation on this site has been structured to provide answers to a wide range of questions. [Concepts](#) explain the Kubernetes architecture and how each component works, while [Setup](#) provides practical instructions for getting started. [Tasks](#) show how to accomplish commonly used tasks, and [Tutorials](#) are more comprehensive walkthroughs of real-world, industry-specific, or end-to-end development scenarios. The [Reference](#) section provides detailed documentation on the [Kubernetes API](#) and command-line interfaces (CLIs), such as [kubectl](#).

You may also find the Stack Overflow topics relevant:

- [Kubernetes](#)
- [Google Kubernetes Engine](#)

## Help! My question isn't covered! I need help now!

### Stack Overflow

Someone else from the community may have already asked a similar question or may be able to help with your problem. The Kubernetes team will also monitor [posts tagged Kubernetes](#). If there aren't any existing questions that help, please [ask a new one!](#)

### Slack

The Kubernetes team hangs out on Slack in the `#kubernetes-users` channel. You can participate in discussion with the Kubernetes team [here](#). Slack requires registration, but the Kubernetes team is open invitation to anyone to register [here](#). Feel free to come and ask any and all questions.

Once registered, browse the growing list of channels for various subjects of interest. For example, people new to Kubernetes may also want to join the `#kubernetes-novice` channel. As another example, developers should join the `#kubernetes-dev` channel.

There are also many country specific/local language channels. Feel free to join these channels for localized support and info:

- China: #cn-users, #cn-events
- Finland: #fi-users
- France: #fr-users, #fr-events
- Germany: #de-users, #de-events
- India: #in-users, #in-events
- Italy: #it-users, #it-events
- Japan: #jp-users, #jp-events
- Korea: #kr-users
- Netherlands: #nl-users
- Norway: #norw-users
- Poland: #pl-users
- Russia: #ru-users
- Spain: #es-users
- Sweden: #se-users
- Turkey: #tr-users, #tr-events

## Forum

The Kubernetes Official Forum [discuss.kubernetes.io](https://discuss.kubernetes.io)

## Bugs and Feature requests

If you have what looks like a bug, or you would like to make a feature request, please use the [GitHub issue tracking system](#).

Before you file an issue, please search existing issues to see if your issue is already covered.

If filing a bug, please include detailed information about how to reproduce the problem, such as:

- Kubernetes version: `kubectl version`
- Cloud provider, OS distro, network configuration, and Docker version
- Steps to reproduce the problem

## Feedback

Was this page helpful?

Yes No

Thanks for the feedback. If you have a specific, answerable question about how to use Kubernetes, ask it on [Stack Overflow](#). Open an issue in the GitHub repo if you want to [report a problem](#) or [suggest an improvement](#).

---

[Create an Issue](#) [Edit This Page](#)

Page last modified on May 27, 2019 at 3:09 PM PST by [Add more language specific slack channels \(#14506\)](#) ([Page History](#))

[Edit This Page](#)

# Configure the Aggregation Layer

Configuring the [aggregation layer](#) allows the Kubernetes apiserver to be extended with additional APIs, which are not part of the core Kubernetes APIs.

- [Before you begin](#)
- [Authentication Flow](#)
- [Enable Kubernetes Apiserver flags](#)
- [What's next](#)

## Before you begin

You need to have a Kubernetes cluster, and the kubectl command-line tool must be configured to communicate with your cluster. 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`.

**Note:** There are a few setup requirements for getting the aggregation layer working in your environment to support mutual TLS auth between the proxy and extension apiservers. Kubernetes and the kube-apiserver have multiple CAs, so make sure that the proxy is signed by the aggregation layer CA and not by something else, like the master CA.

[Edit This Page](#)

# Extend the Kubernetes API with CustomResourceDefinitions

This page shows how to install a [custom resource](#) into the Kubernetes API by creating a [CustomResourceDefinition](#).

- [Before you begin](#)
- [Create a CustomResourceDefinition](#)
- [Create custom objects](#)
- [Delete a CustomResourceDefinition](#)
- [Specifying a structural schema](#)
- [Serving multiple versions of a CRD](#)
- [Advanced topics](#)
- [What's next](#)

## Before you begin

You need to have a Kubernetes cluster, and the kubectl command-line tool must be configured to communicate with your cluster. 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`.

- Make sure your Kubernetes cluster has a master version of 1.16.0 or higher to use `apiextensions.k8s.io/v1`, or 1.7.0 or higher for `apiextensions.k8s.io/v1beta1`.
- Read about [custom resources](#).

## Create a CustomResourceDefinition

When you create a new CustomResourceDefinition (CRD), the Kubernetes API Server creates a new RESTful resource path for each version you specify. The CRD can be either namespaced or cluster-scoped, as specified in the CRD's `scope` field. As with existing built-in objects, deleting a namespace deletes all custom objects in that namespace. CustomResourceDefinitions themselves are non-namespaced and are available to all namespaces.

For example, if you save the following CustomResourceDefinition to `resourcedefinition.yaml`:

- [apiextensions.k8s.io/v1](#)
- [apiextensions.k8s.io/v1beta1](#)

```
apiVersion: apiextensions.k8s.io/v1
kind: CustomResourceDefinition
metadata:
  # name must match the spec fields below, and be in the
  form: <plural>.<group>
  name: crontabs.stable.example.com
spec:
  # group name to use for REST API: /apis/<group>/<version>
  group: stable.example.com
  # list of versions supported by this
  # CustomResourceDefinition
  versions:
    - name: v1
      # Each version can be enabled/disabled by Served flag.
      served: true
      # One and only one version must be marked as the
      # storage version.
      storage: true
      schema:
        openAPIV3Schema:
          type: object
          properties:
            spec:
              type: object
              properties:
                cronSpec:
                  type: string
                image:
```

```

        type: string
replicas:
        type: integer
# either Namespaced or Cluster
scope: Namespaced
names:
# plural name to be used in the URL: /apis/<group>/<version>/<plural>
plural: crontabs
# singular name to be used as an alias on the CLI and for display
singular: crontab
# kind is normally the CamelCased singular type. Your resource manifests use this.
kind: CronTab
# shortNames allow shorter string to match your resource on the CLI
shortNames:
- ct

```

[Edit This Page](#)

# Versions in CustomResourceDefinitions

This page explains how to add versioning information to [CustomResourceDefinitions](#), to indicate the stability level of your CustomResourceDefinitions or advance your API to a new version with conversion between API representations. It also describes how to upgrade an object from one version to another.

- [Before you begin](#)
- [Overview](#)
- [Specify multiple versions](#)
- [Webhook conversion](#)
- [Webhook request and response](#)
- [Writing, reading, and updating versioned CustomResourceDefinition objects](#)
- [Upgrade existing objects to a new stored version](#)

## Before you begin

You need to have a Kubernetes cluster, and the kubectl command-line tool must be configured to communicate with your cluster. 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](#)

You should have a initial understanding of [custom resources](#).

Your Kubernetes server must be at or later than version v1.16. To check the version, enter `kubectl version`.

## Overview

The CustomResourceDefinition API provides a workflow for introducing and upgrading to new versions of a CustomResourceDefinition.

When a CustomResourceDefinition is created, the first version is set in the CustomResourceDefinition `spec.versions` list to an appropriate stability level and a version number. For example `v1beta1` would indicate that the first version is not yet stable. All custom resource objects will initially be stored at this version.

Once the CustomResourceDefinition is created, clients may begin using the `v1beta1` API.

Later it might be necessary to add new version such as `v1`.

Adding a new version:

1. Pick a conversion strategy. Since custom resource objects need to be able to be served at both versions, that means they will sometimes be served at a different version than their storage version. In order for this to be possible, the custom resource objects must sometimes be converted between the version they are stored at and the version they are served at. If the conversion involves schema changes and requires custom logic, a conversion webhook should be used. If there are no schema changes, the default `None` conversion strategy may be used and only the `apiVersion` field will be modified when serving different versions.
2. If using conversion webhooks, create and deploy the conversion webhook. See the [Webhook conversion](#) for more details.
3. Update the CustomResourceDefinition to include the new version in the `spec.versions` list with `served:true`. Also, set `spec.conversion` field to the selected conversion strategy. If using a conversion webhook, configure `spec.conversion.webhookClientConfig` field to call the webhook.

Once the new version is added, clients may incrementally migrate to the new version. It is perfectly safe for some clients to use the old version while others use the new version.

Migrate stored objects to the new version:

1. See the [upgrade existing objects to a new stored version](#) section.

It is safe for clients to use both the old and new version before, during and after upgrading the objects to a new stored version.

Removing an old version:

1. Ensure all clients are fully migrated to the new version. The kube-apiserver logs can be reviewed to help identify any clients that are still accessing via the old version.
2. Set `served` to `false` for the old version in the `spec.versions` list. If any clients are still unexpectedly using the old version they may begin reporting errors attempting to access the custom resource objects at the old version. If this occurs, switch back to using `served:true` on the old version, migrate the remaining clients to the new version and repeat this step.
3. Ensure the [upgrade of existing objects to the new stored version](#) step has been completed.
  1. Verify that the `stored` is set to `true` for the new version in the `spec.versions` list in the CustomResourceDefinition.
  2. Verify that the old version is no longer listed in the CustomResourceDefinition `status.storedVersions`.
4. Remove the old version from the CustomResourceDefinition `spec.versions` list.
5. Drop conversion support for the old version in conversion webhooks.

## Specify multiple versions

The CustomResourceDefinition API `versions` field can be used to support multiple versions of custom resources that you have developed. Versions can have different schemas, and conversion webhooks can convert custom resources between versions. Webhook conversions should follow the [Kubernetes API conventions](#) wherever applicable. Specifically, See the [API change documentation](#) for a set of useful gotchas and suggestions.

**Note:** In `apiextensions.k8s.io/v1beta1`, there was a `version` field instead of `versions`. The `version` field is deprecated and optional, but if it is not empty, it must match the first item in the `versions` field.

This example shows a CustomResourceDefinition with two versions. For the first example, the assumption is all versions share the same schema with no conversion between them. The comments in the YAML provide more context.

- [apiextensions.k8s.io/v1](#)
- [apiextensions.k8s.io/v1beta1](#)

```
apiVersion: apiextensions.k8s.io/v1
kind: CustomResourceDefinition
metadata:
  # name must match the spec fields below, and be in the
  form: <plural>.<group>
  name: crontabs.example.com
spec:
```

```

# group name to use for REST API: /apis/<group>/<version>
group: example.com
# list of versions supported by this
CustomResourceDefinition
  versions:
    - name: v1beta1
      # Each version can be enabled/disabled by Served flag.
      served: true
      # One and only one version must be marked as the storage
      version.
      storage: true
      # A schema is required
      schema:
        openAPIV3Schema:
          type: object
          properties:
            host:
              type: string
            port:
              type: string
    - name: v1
      served: true
      storage: false
      schema:
        openAPIV3Schema:
          type: object
          properties:
            host:
              type: string
            port:
              type: string
      # The conversion section is introduced in Kubernetes 1.13+
      # with a default value of
      # None conversion (strategy sub-field set to None).
      conversion:
        # None conversion assumes the same schema for all
        # versions and only sets the apiVersion
        # field of custom resources to the proper value
        strategy: None
      # either Namespaced or Cluster
      scope: Namespaced
      names:
        # plural name to be used in the URL: /apis/<group>/<version>/<plural>
        plural: crontabs
        # singular name to be used as an alias on the CLI and
        # for display
        singular: crontab
        # kind is normally the CamelCased singular type. Your
        # resource manifests use this.
        kind: CronTab

```

```
# shortNames allow shorter string to match your resource
on the CLI
shortNames:
- ct
```

[Edit This Page](#)

# Setup an Extension API Server

Setting up an extension API server to work the aggregation layer allows the Kubernetes apiserver to be extended with additional APIs, which are not part of the core Kubernetes APIs.

- [Before you begin](#)
- [Setup an extension api-server to work with the aggregation layer](#)
- [What's next](#)

## Before you begin

You need to have a Kubernetes cluster, and the kubectl command-line tool must be configured to communicate with your cluster. 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`.

- You must [configure the aggregation layer](#) and enable the apiserver flags.

## Setup an extension api-server to work with the aggregation layer

The following steps describe how to set up an extension-apiserver at a *high level*. These steps apply regardless if you're using YAML configs or using APIs. An attempt is made to specifically identify any differences between the two. For a concrete example of how they can be implemented using YAML configs, you can look at the [sample-apiserver](#) in the Kubernetes repo.

Alternatively, you can use an existing 3rd party solution, such as [apiserver-builder](#), which should generate a skeleton and automate all of the following steps for you.

1. Make sure the APIService API is enabled (check `--runtime-config`). It should be on by default, unless it's been deliberately turned off in your cluster.

2. You may need to make an RBAC rule allowing you to add APIService objects, or get your cluster administrator to make one. (Since API extensions affect the entire cluster, it is not recommended to do testing/development/debug of an API extension in a live cluster.)
3. Create the Kubernetes namespace you want to run your extension api-service in.
4. Create/get a CA cert to be used to sign the server cert the extension api-server uses for HTTPS.
5. Create a server cert/key for the api-server to use for HTTPS. This cert should be signed by the above CA. It should also have a CN of the Kube DNS name. This is derived from the Kubernetes service and be of the form <service name>.<service name namespace>.svc
6. Create a Kubernetes secret with the server cert/key in your namespace.
7. Create a Kubernetes deployment for the extension api-server and make sure you are loading the secret as a volume. It should contain a reference to a working image of your extension api-server. The deployment should also be in your namespace.
8. Make sure that your extension-apiserver loads those certs from that volume and that they are used in the HTTPS handshake.
9. Create a Kubernetes service account in your namespace.
10. Create a Kubernetes cluster role for the operations you want to allow on your resources.
11. Create a Kubernetes cluster role binding from the service account in your namespace to the cluster role you just created.
12. Create a Kubernetes cluster role binding from the service account in your namespace to the system:auth-delegator cluster role to delegate auth decisions to the Kubernetes core API server.
13. Create a Kubernetes role binding from the service account in your namespace to the extension-apiserver-authentication-reader role. This allows your extension api-server to access the extension-apiserver-authentication configmap.
14. Create a Kubernetes apiservice. The CA cert above should be base64 encoded, stripped of new lines and used as the spec.caBundle in the apiservice. This should not be namespaced. If using the [kube-aggregator API](#), only pass in the PEM encoded CA bundle because the base 64 encoding is done for you.
15. Use kubectl to get your resource. It should return "No resources found." Which means that everything worked but you currently have no objects of that resource type created yet.

## What's next

- If you haven't already, [configure the aggregation layer](#) and enable the apiserver flags.
- For a high level overview, see [Extending the Kubernetes API with the aggregation layer](#).
- Learn how to [Extend the Kubernetes API Using Custom Resource Definitions](#).

# Feedback

Was this page helpful?

Yes No

Thanks for the feedback. If you have a specific, answerable question about how to use Kubernetes, ask it on [Stack Overflow](#). Open an issue in the GitHub repo if you want to [report a problem](#) or [suggest an improvement](#).

---

[Create an Issue](#) [Edit This Page](#)

Page last modified on October 11, 2018 at 9:21 PM PST by [Fix some "capture prerequisites" errors in docs/tasks. \(#10270\)](#) ([Page History](#))

[Edit This Page](#)

# Use an HTTP Proxy to Access the Kubernetes API

This page shows how to use an HTTP proxy to access the Kubernetes API.

- [Before you begin](#)
- [Using kubectl to start a proxy server](#)
- [Exploring the Kubernetes API](#)
- [What's next](#)

## Before you begin

You need to have a Kubernetes cluster, and the `kubectl` command-line tool must be configured to communicate with your cluster. 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`.

If you do not already have an application running in your cluster, start a Hello world application by entering this command:

```
kubectl run node-hello --image=gcr.io/google-samples/node-hello:1.0 --port=8080
```

## Using `kubectl` to start a proxy server

This command starts a proxy to the Kubernetes API server:

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

## Exploring the Kubernetes API

When the proxy server is running, you can explore the API using `curl`, `wget`, or a browser.

Get the API versions:

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

The output should look similar to this:

```
{
  "kind": "APIVersions",
  "versions": [
```

```
        "v1"
    ],
    "serverAddressByClientCIDRs": [
        {
            "clientCIDR": "0.0.0.0/0",
            "serverAddress": "10.0.2.15:8443"
        }
    ]
}
```

Get a list of pods:

```
curl http://localhost:8080/api/v1/namespaces/default/pods
```

The output should look similar to this:

```
{
  "kind": "PodList",
  "apiVersion": "v1",
  "metadata": {
    "resourceVersion": "33074"
  },
  "items": [
    {
      "metadata": {
        "name": "kubernetes-bootcamp-2321272333-ix8pt",
        "generateName": "kubernetes-bootcamp-2321272333-",
        "namespace": "default",
        "uid": "ba21457c-6b1d-11e6-85f7-1ef9f1dab92b",
        "resourceVersion": "33003",
        "creationTimestamp": "2016-08-25T23:43:30Z",
        "labels": {
          "pod-template-hash": "2321272333",
          "run": "kubernetes-bootcamp"
        },
        ...
      }
    }
}
```

## What's next

Learn more about [kubectl proxy](#).

## Feedback

Was this page helpful?

Yes No

Thanks for the feedback. If you have a specific, answerable question about how to use Kubernetes, ask it on [Stack Overflow](#). Open an issue

in the GitHub repo if you want to [report a problem](#) or [suggest an improvement](#).

---

[Create an Issue](#) [Edit This Page](#)

Page last modified on February 07, 2020 at 8:59 PM PST by [Separated commands from output \(#19023\)](#) ([Page History](#))

[Edit This Page](#)

# Certificate Rotation

This page shows how to enable and configure certificate rotation for the kubelet.

- [Before you begin](#)
- [Overview](#)
- [Enabling client certificate rotation](#)

- [Understanding the certificate rotation configuration](#)

## Before you begin

- Kubernetes version 1.8.0 or later is required

## Overview

The kubelet uses certificates for authenticating to the Kubernetes API. By default, these certificates are issued with one year expiration so that they do not need to be renewed too frequently.

Kubernetes 1.8 contains [kubelet certificate rotation](#), a beta feature that will automatically generate a new key and request a new certificate from the Kubernetes API as the current certificate approaches expiration. Once the new certificate is available, it will be used for authenticating connections to the Kubernetes API.

## Enabling client certificate rotation

The kubelet process accepts an argument `--rotate-certificates` that controls if the kubelet will automatically request a new certificate as the expiration of the certificate currently in use approaches. Since certificate rotation is a beta feature, the feature flag must also be enabled with `--feature-gates=RotateKubeletClientCertificate=true`.

The kube-controller-manager process accepts an argument `--experimental-cluster-signing-duration` that controls how long certificates will be issued for.

## Understanding the certificate rotation configuration

When a kubelet starts up, if it is configured to bootstrap (using the `--bootstrap-kubeconfig` flag), it will use its initial certificate to connect to the Kubernetes API and issue a certificate signing request. You can view the status of certificate signing requests using:

```
kubectl get csr
```

Initially a certificate signing request from the kubelet on a node will have a status of Pending. If the certificate signing requests meets specific criteria, it will be auto approved by the controller manager, then it will have a status of Approved. Next, the controller manager will sign a certificate, issued for the duration specified by the `--experimental-cluster-signing-duration` parameter, and the signed certificate will be attached to the certificate signing requests.

The kubelet will retrieve the signed certificate from the Kubernetes API and write that to disk, in the location specified by `--cert-dir`. Then the kubelet will use the new certificate to connect to the Kubernetes API.

As the expiration of the signed certificate approaches, the kubelet will automatically issue a new certificate signing request, using the Kubernetes API. Again, the controller manager will automatically approve the certificate request and attach a signed certificate to the certificate signing request. The kubelet will retrieve the new signed certificate from the Kubernetes API and write that to disk. Then it will update the connections it has to the Kubernetes API to reconnect using the new certificate.

**FEATURE STATE:** Kubernetes v1.8 [beta](#)

This feature is currently in a *beta* state, meaning:

[Edit This Page](#)

## Manage TLS Certificates in a Cluster

Kubernetes provides a `certificates.k8s.io` API, which lets you provision TLS certificates signed by a Certificate Authority (CA) that you control. These CA and certificates can be used by your workloads to establish trust.

`certificates.k8s.io` API uses a protocol that is similar to the [ACME draft](#).

**Note:** Certificates created using the `certificates.k8s.io` API are signed by a dedicated CA. It is possible to configure your cluster to use the cluster root CA for this purpose, but you should never rely on this. Do not assume that these certificates will validate against the cluster root CA.

- [Before you begin](#)
- [Trusting TLS in a Cluster](#)
- [Requesting a Certificate](#)
- [Download and install CFSSL](#)
- [Create a Certificate Signing Request](#)
- [Create a Certificate Signing Request object to send to the Kubernetes API](#)
- [Get the Certificate Signing Request Approved](#)
- [Download the Certificate and Use It](#)
- [Approving Certificate Signing Requests](#)
- [A Word of Warning on the Approval Permission](#)
- [A Note to Cluster Administrators](#)

# Before you begin

You need to have a Kubernetes cluster, and the kubectl command-line tool must be configured to communicate with your cluster. 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`.

## Trusting TLS in a Cluster

Trusting the custom CA from an application running as a pod usually requires some extra application configuration. You will need to add the CA certificate bundle to the list of CA certificates that the TLS client or server trusts. For example, you would do this with a golang TLS config by parsing the certificate chain and adding the parsed certificates to the `RootCAs` field in the [`tls.Config`](#) struct.

You can distribute the CA certificate as a [ConfigMap](#) that your pods have access to use.

## Requesting a Certificate

The following section demonstrates how to create a TLS certificate for a Kubernetes service accessed through DNS.

**Note:** This tutorial uses CFSSL: Cloudflare's PKI and TLS toolkit [click here](#) to know more.

## Download and install CFSSL

The cfssl tools used in this example can be downloaded at <https://pkg.cfssl.org/>.

## Create a Certificate Signing Request

Generate a private key and certificate signing request (or CSR) by running the following command:

```
cat <<EOF | cfssl genkey - | cfssljson -bare server
{
  "hosts": [
    "my-svc.my-namespace.svc.cluster.local",
    "my-pod.my-namespace.pod.cluster.local",
    "192.0.2.24",
    "10.0.34.2"
```

```
],
  "CN": "my-pod.my-namespace.pod.cluster.local",
  "key": {
    "algo": "ecdsa",
    "size": 256
  }
}
EOF
```

Where 192.0.2.24 is the service's cluster IP, my-svc.my-namespace.svc.cluster.local is the service's DNS name, 10.0.34.2 is the pod's IP and my-pod.my-namespace.pod.cluster.local is the pod's DNS name. You should see the following output:

```
2017/03/21 06:48:17 [INFO] generate received request
2017/03/21 06:48:17 [INFO] received CSR
2017/03/21 06:48:17 [INFO] generating key: ecdsa-256
2017/03/21 06:48:17 [INFO] encoded CSR
```

This command generates two files; it generates server.csr containing the PEM encoded [pkcs#10](#) certification request, and server-key.pem containing the PEM encoded key to the certificate that is still to be created.

## Create a Certificate Signing Request object to send to the Kubernetes API

Generate a CSR yaml blob and send it to the apiserver by running the following command:

```
cat <<EOF | kubectl apply -f -
apiVersion: certificates.k8s.io/v1beta1
kind: CertificateSigningRequest
metadata:
  name: my-svc.my-namespace
spec:
  request: $(cat server.csr | base64 | tr -d '\n')
  usages:
  - digital signature
  - key encipherment
  - server auth
EOF
```

Notice that the server.csr file created in step 1 is base64 encoded and stashed in the .spec.request field. We are also requesting a certificate with the "digital signature", "key encipherment", and "server auth" key usages. We support all key usages and extended key usages listed [here](#) so you can request client certificates and other certificates using this same API.

The CSR should now be visible from the API in a Pending state. You can see it by running:

```
kubectl describe csr my-svc.my-namespace
```

Name:	my-svc.my-namespace
Labels:	<none>
Annotations:	<none>
CreationTimestamp:	Tue, 21 Mar 2017 07:03:51 -0700
Requesting User:	yourname@example.com
Status:	Pending
Subject:	Common Name: my-svc.my-namespace.svc.cluster.local Serial Number:
Subject Alternative Names:	DNS Names: my-svc.my-namespace.svc.cluster.local IP Addresses: 192.0.2.24 10.0.34.2
Events:	<none>

## Get the Certificate Signing Request Approved

Approving the certificate signing request is either done by an automated approval process or on a one off basis by a cluster administrator. More information on what this involves is covered below.

## Download the Certificate and Use It

Once the CSR is signed and approved you should see the following:

```
kubectl get csr
```

NAME	AGE	REQUESTOR
my-svc.my-namespace	10m	yourname@example.com
Approved, Issued		

You can download the issued certificate and save it to a `server.crt` file by running the following:

```
kubectl get csr my-svc.my-namespace -o jsonpath='{.status.certificate}' \
| base64 --decode > server.crt
```

Now you can use `server.crt` and `server-key.pem` as the keypair to start your HTTPS server.

# Approving Certificate Signing Requests

A Kubernetes administrator (with appropriate permissions) can manually approve (or deny) Certificate Signing Requests by using the `kubectl certificate approve` and `kubectl certificate deny` commands. However if you intend to make heavy usage of this API, you might consider writing an automated certificates controller.

Whether a machine or a human using `kubectl` as above, the role of the approver is to verify that the CSR satisfies two requirements:

1. The subject of the CSR controls the private key used to sign the CSR. This addresses the threat of a third party masquerading as an authorized subject. In the above example, this step would be to verify that the pod controls the private key used to generate the CSR.
2. The subject of the CSR is authorized to act in the requested context. This addresses the threat of an undesired subject joining the cluster. In the above example, this step would be to verify that the pod is allowed to participate in the requested service.

If and only if these two requirements are met, the approver should approve the CSR and otherwise should deny the CSR.

## A Word of Warning on the Approval Permission

The ability to approve CSRs decides who trusts who within your environment. The ability to approve CSRs should not be granted broadly or lightly. The requirements of the challenge noted in the previous section and the repercussions of issuing a specific certificate should be fully understood before granting this permission.

## A Note to Cluster Administrators

This tutorial assumes that a signer is setup to serve the certificates API. The Kubernetes controller manager provides a default implementation of a signer. To enable it, pass the `--cluster-signing-cert-file` and `--cluster-signing-key-file` parameters to the controller manager with paths to your Certificate Authority's keypair.

## Feedback

Was this page helpful?

Yes No

Thanks for the feedback. If you have a specific, answerable question about how to use Kubernetes, ask it on [Stack Overflow](#). Open an issue in the GitHub repo if you want to [report a problem](#) or [suggest an improvement](#).

---

[Create an Issue](#) [Edit This Page](#)

Page last modified on August 18, 2019 at 8:48 PM PST by [fix broken link \(#15897\)](#) ([Page History](#))

[Edit This Page](#)

# Perform a Rolling Update on a DaemonSet

This page shows how to perform a rolling update on a DaemonSet.

- [Before you begin](#)
- [DaemonSet Update Strategy](#)
- [Performing a Rolling Update](#)
- [Troubleshooting](#)
- [What's next](#)

## Before you begin

- The DaemonSet rolling update feature is only supported in Kubernetes version 1.6 or later.

## DaemonSet Update Strategy

DaemonSet has two update strategy types:

- OnDelete: With `OnDelete` update strategy, after you update a DaemonSet template, new DaemonSet pods will *only* be created when you manually delete old DaemonSet pods. This is the same behavior of DaemonSet in Kubernetes version 1.5 or before.
- RollingUpdate: This is the default update strategy. With `RollingUpdate` update strategy, after you update a DaemonSet template, old DaemonSet pods will be killed, and new DaemonSet pods will be created automatically, in a controlled fashion.

## Performing a Rolling Update

To enable the rolling update feature of a DaemonSet, you must set its `.spec.updateStrategy.type` to `RollingUpdate`.

You may want to set `.spec.updateStrategy.rollingUpdate.maxUnavailable` (default to 1) and `.spec.minReadySeconds` (default to 0) as well.

### Step 1: Checking DaemonSet RollingUpdate update strategy

First, check the update strategy of your DaemonSet, and make sure it's set to `RollingUpdate`:

```
kubectl get ds/<daemonset-name> -o go-template='{{.spec.updateStrategy.type}}{{"\n"}}'
```

If you haven't created the DaemonSet in the system, check your DaemonSet manifest with the following command instead:

```
kubectl apply -f ds.yaml --dry-run -o go-template='{{.spec.updateStrategy.type}}{{"\n"}}'
```

The output from both commands should be:

```
RollingUpdate
```

If the output isn't RollingUpdate, go back and modify the DaemonSet object or manifest accordingly.

## Step 2: Creating a DaemonSet with RollingUpdate update strategy

If you have already created the DaemonSet, you may skip this step and jump to step 3.

After verifying the update strategy of the DaemonSet manifest, create the DaemonSet:

```
kubectl create -f ds.yaml
```

Alternatively, use `kubectl apply` to create the same DaemonSet if you plan to update the DaemonSet with `kubectl apply`.

```
kubectl apply -f ds.yaml
```

## Step 3: Updating a DaemonSet template

Any updates to a RollingUpdate DaemonSet `.spec.template` will trigger a rolling update. This can be done with several different `kubectl` commands.

### Declarative commands

If you update DaemonSets using [configuration files](#), use `kubectl apply`:

```
kubectl apply -f ds-v2.yaml
```

### Imperative commands

If you update DaemonSets using [imperative commands](#), use `kubectl edit` or `kubectl patch`:

```
kubectl edit ds/<daemonset-name>
```

```
kubectl patch ds/<daemonset-name> -p=<strategic-merge-patch>
```

## Updating only the container image

If you just need to update the container image in the DaemonSet template, i.e. `.spec.template.spec.containers[*].image`, use `kubectl set image`:

```
kubectl set image ds/<daemonset-name> <container-name>=<container-new-image>
```

## Step 4: Watching the rolling update status

Finally, watch the rollout status of the latest DaemonSet rolling update:

```
kubectl rollout status ds/<daemonset-name>
```

When the rollout is complete, the output is similar to this:

```
daemonset "<daemonset-name>" successfully rolled out
```

# Troubleshooting

## DaemonSet rolling update is stuck

Sometimes, a DaemonSet rolling update may be stuck. Here are some possible causes:

### Some nodes run out of resources

The rollout is stuck because new DaemonSet pods can't be scheduled on at least one node. This is possible when the node is [running out of resources](#).

When this happens, find the nodes that don't have the DaemonSet pods scheduled on by comparing the output of `kubectl get nodes` and the output of:

```
kubectl get pods -l <daemonset-selector-key>=<daemonset-selector-value> -o wide
```

Once you've found those nodes, delete some non-DaemonSet pods from the node to make room for new DaemonSet pods.

**Note:** This will cause service disruption when deleted pods are not controlled by any controllers or pods are not replicated. This does not respect [PodDisruptionBudget](#) either.

### Broken rollout

If the recent DaemonSet template update is broken, for example, the container is crash looping, or the container image doesn't exist (often due to a typo), DaemonSet rollout won't progress.

To fix this, just update the DaemonSet template again. New rollout won't be blocked by previous unhealthy rollouts.

## Clock skew

If `.spec.minReadySeconds` is specified in the DaemonSet, clock skew between master and nodes will make DaemonSet unable to detect the right rollout progress.

## What's next

- See [Task: Performing a rollback on a DaemonSet](#)
- See [Concepts: Creating a DaemonSet to adopt existing DaemonSet pods](#)

## Feedback

Was this page helpful?

Yes No

Thanks for the feedback. If you have a specific, answerable question about how to use Kubernetes, ask it on [Stack Overflow](#). Open an issue in the GitHub repo if you want to [report a problem](#) or [suggest an improvement](#).

---

[Create an Issue](#) [Edit This Page](#)

Page last modified on July 21, 2019 at 6:24 AM PST by [Fix command \(#15479\)](#) ([Page History](#))

[Edit This Page](#)

# Perform a Rollback on a DaemonSet

This page shows how to perform a rollback on a DaemonSet.

- [Before you begin](#)
- [Performing a Rollback on a DaemonSet](#)
- [Understanding DaemonSet Revisions](#)
- [Troubleshooting](#)

# Before you begin

- The DaemonSet rollout history and DaemonSet rollback features are only supported in kubectl in Kubernetes version 1.7 or later.
- Make sure you know how to [perform a rolling update on a DaemonSet](#).

## Performing a Rollback on a DaemonSet

### Step 1: Find the DaemonSet revision you want to roll back to

You can skip this step if you just want to roll back to the last revision.

List all revisions of a DaemonSet:

```
kubectl rollout history daemonset <daemonset-name>
```

This returns a list of DaemonSet revisions:

```
daemonsets "<daemonset-name>"  
REVISION      CHANGE-CAUSE  
1            ...  
2            ...  
..."
```

- Change cause is copied from DaemonSet annotation `kubernetes.io/change-cause` to its revisions upon creation. You may specify `--record=true` in `kubectl` to record the command executed in the change cause annotation.

To see the details of a specific revision:

```
kubectl rollout history daemonset <daemonset-name> --revision=1
```

This returns the details of that revision:

```
daemonsets "<daemonset-name>" with revision #1  
Pod Template:  
Labels:      foo=bar  
Containers:  
app:  
  Image:      ...  
  Port:       ...  
  Environment: ...  
  Mounts:     ...  
  Volumes:    ...
```

## Step 2: Roll back to a specific revision

```
# Specify the revision number you get from Step 1 in --to-revision
```

```
kubectl rollout undo daemonset <daemonset-name> --to-revision=<revision>
```

If it succeeds, the command returns:

```
daemonset "<daemonset-name>" rolled back
```

If --to-revision flag is not specified, the last revision will be picked.

## Step 3: Watch the progress of the DaemonSet rollback

`kubectl rollout undo daemonset` tells the server to start rolling back the DaemonSet. The real rollback is done asynchronously on the server side.

To watch the progress of the rollback:

```
kubectl rollout status ds/<daemonset-name>
```

When the rollback is complete, the output is similar to this:

```
daemonset "<daemonset-name>" successfully rolled out
```

## Understanding DaemonSet Revisions

In the previous `kubectl rollout history` step, you got a list of DaemonSet revisions. Each revision is stored in a resource named `ControllerRevision`. `ControllerRevision` is a resource only available in Kubernetes release 1.7 or later.

To see what is stored in each revision, find the DaemonSet revision raw resources:

```
kubectl get controllerrevision -l <daemonset-selector-key>=<daemonset-selector-value>
```

This returns a list of `ControllerRevisions`:

NAME	CONTROLLER	REVISION	AGE
	<daemonset-name>-<revision-hash>	DaemonSet/<daemonset-name>	
	1	1h	
	<daemonset-name>-<revision-hash>	DaemonSet/<daemonset-name>	
	2	1h	

Each `ControllerRevision` stores the annotations and template of a DaemonSet revision.

`kubectl rollout undo` takes a specific ControllerRevision and replaces DaemonSet template with the template stored in the ControllerRevision. `kubectl rollout undo` is equivalent to updating DaemonSet template to a previous revision through other commands, such as `kubectl edit` or `kubectl apply`.

**Note:** DaemonSet revisions only roll forward. That is to say, after a rollback completes, the revision number (`.revision` field) of the ControllerRevision being rolled back to will advance. For example, if you have revision 1 and 2 in the system, and roll back from revision 2 to revision 1, the ControllerRevision with `.revision: 1` will become `.revision: 3`.

## Troubleshooting

- See [troubleshooting DaemonSet rolling update](#).

## Feedback

Was this page helpful?

Yes No

Thanks for the feedback. If you have a specific, answerable question about how to use Kubernetes, ask it on [Stack Overflow](#). Open an issue in the GitHub repo if you want to [report a problem](#) or [suggest an improvement](#).

---

[Create an Issue](#) [Edit This Page](#)

Page last modified on May 20, 2019 at 5:41 PM PST by [Correcting sequence of tasks for manage cluster daemon \(#14383\)](#) ([Page History](#))

[Edit This Page](#)

## Install Service Catalog using Helm

Service Catalog is an extension API that enables applications running in Kubernetes clusters to easily use external managed software offerings, such as a datastore service offered by a cloud provider.

It provides a way to list, provision, and bind with external [Managed Services](#)[A software offering maintained by a third-party provider.](#) from [Service Brokers](#)[An endpoint for a set of Managed Services offered and](#)

[maintained by a third-party](#), without needing detailed knowledge about how those services are created or managed.

Use [Helm](#) to install Service Catalog on your Kubernetes cluster. Up to date information on this process can be found at the [kubernetes-sigs/service-catalog](#) repo.

- [Before you begin](#)
- [Add the service-catalog Helm repository](#)
- [Enable RBAC](#)
- [Install Service Catalog in your Kubernetes cluster](#)
- [What's next](#)

## Before you begin

- Understand the key concepts of [Service Catalog](#).
- Service Catalog requires a Kubernetes cluster running version 1.7 or higher.
- You must have a Kubernetes cluster with cluster DNS enabled.
  - If you are using a cloud-based Kubernetes cluster or [MinikubeA tool for running Kubernetes locally.](#), you may already have cluster DNS enabled.
  - If you are using `hack/local-up-cluster.sh`, ensure that the `KUBE_ENABLE_CLUSTER_DNS` environment variable is set, then run the install script.
- [Install and setup kubectl](#) v1.7 or higher. Make sure it is configured to connect to the Kubernetes cluster.
- Install [Helm](#) v2.7.0 or newer.
  - Follow the [Helm install instructions](#).
  - If you already have an appropriate version of Helm installed, execute `helm init` to install Tiller, the server-side component of Helm.

## Add the service-catalog Helm repository

Once Helm is installed, add the *service-catalog* Helm repository to your local machine by executing the following command:

```
helm repo add svc-cat https://svc-catalog-charts.storage.googleapis.com
```

Check to make sure that it installed successfully by executing the following command:

```
helm search service-catalog
```

If the installation was successful, the command should output the following:

NAME DESCRIPTION	CHART VERSION	APP VERSION
---------------------	---------------	-------------

```
svc-cat/catalog      0.2.1
service-catalog API server and controller-manager helm chart
svc-cat/catalog-v0.2   0.2.2
service-catalog API server and controller-manager helm chart
```

## Enable RBAC

Your Kubernetes cluster must have RBAC enabled, which requires your Tiller Pod(s) to have `cluster-admin` access.

When using Minikube v0.25 or older, you must run Minikube with RBAC explicitly enabled:

```
minikube start --extra-config=apiserver.Authorization.Mode=RBAC
```

When using Minikube v0.26+, run:

```
minikube start
```

With Minikube v0.26+, do not specify `--extra-config`. The flag has since been changed to `-extra-config=apiserver.authorization-mode` and Minikube now uses RBAC by default. Specifying the older flag may cause the start command to hang.

If you are using `hack/local-up-cluster.sh`, set the `AUTHORIZATION_MODE` environment variable with the following values:

```
AUTHORIZATION_MODE=Node,RBAC hack/local-up-cluster.sh -0
```

By default, `helm init` installs the Tiller Pod into the `kube-system` namespace, with Tiller configured to use the `default` service account.

**Note:** If you used the `--tiller-namespace` or `--service-account` flags when running `helm init`, the `--serviceaccount` flag in the following command needs to be adjusted to reference the appropriate namespace and ServiceAccount name.

Configure Tiller to have `cluster-admin` access:

```
kubectl create clusterrolebinding tiller-cluster-admin \
--clusterrole=cluster-admin \
--serviceaccount=kube-system:default
```

## Install Service Catalog in your Kubernetes cluster

Install Service Catalog from the root of the Helm repository using the following command:

- [Helm version 3](#)

- [Helm version 2](#)

```
helm install catalog svc-cat/catalog --namespace catalog
```

[Edit This Page](#)

# Install Service Catalog using SC

Service Catalog is an extension API that enables applications running in Kubernetes clusters to easily use external managed software offerings, such as a datastore service offered by a cloud provider.

It provides a way to list, provision, and bind with external [Managed Services](#) [A software offering maintained by a third-party provider](#), from [Service Brokers](#) [An endpoint for a set of Managed Services offered and maintained by a third-party](#), without needing detailed knowledge about how those services are created or managed.

You can use the GCP [Service Catalog Installer](#) tool to easily install or uninstall Service Catalog on your Kubernetes cluster, linking it to Google Cloud projects.

Service Catalog itself can work with any kind of managed service, not just Google Cloud.

- [Before you begin](#)
- [Install sc in your local environment](#)
- [Install Service Catalog in your Kubernetes cluster](#)
- [Uninstall Service Catalog](#)
- [What's next](#)

## Before you begin

- Understand the key concepts of [Service Catalog](#).
- Install [Go 1.6+](#) and set the GOPATH.
- Install the [cfssl](#) tool needed for generating SSL artifacts.
- Service Catalog requires Kubernetes version 1.7+.
- [Install and setup kubectl](#) so that it is configured to connect to a Kubernetes v1.7+ cluster.
- The kubectl user must be bound to the *cluster-admin* role for it to install Service Catalog. To ensure that this is true, run the following command:

```
kubectl create clusterrolebinding cluster-admin-binding  
--clusterrole=cluster-admin --user=<user-name>
```

## Install sc in your local environment

The installer runs on your local computer as a CLI tool named sc.

Install using go get:

```
go get github.com/GoogleCloudPlatform/k8s-service-catalog/
installer/cmd/sc
```

sc should now be installed in your GOPATH/bin directory.

## Install Service Catalog in your Kubernetes cluster

First, verify that all dependencies have been installed. Run:

```
sc check
```

If the check is successful, it should return:

```
Dependency check passed. You are good to go.
```

Next, run the install command and specify the storageclass that you want to use for the backup:

```
sc install --etcd-backup-storageclass "standard"
```

## Uninstall Service Catalog

If you would like to uninstall Service Catalog from your Kubernetes cluster using the sc tool, run:

```
sc uninstall
```

## What's next

- View [sample service brokers](#).
- Explore the [kubernetes-incubator/service-catalog](#) project.

## Feedback

Was this page helpful?

Yes No

Thanks for the feedback. If you have a specific, answerable question about how to use Kubernetes, ask it on [Stack Overflow](#). Open an issue in the GitHub repo if you want to [report a problem](#) or [suggest an improvement](#).

---

[Create an Issue](#) [Edit This Page](#)

Page last modified on March 27, 2019 at 2:28 PM PST by [Note Google Cloud needs of Service Cloud Installer \(#13243\)](#) ([Page History](#))

[Edit This Page](#)

# Validate IPv4/IPv6 dual-stack

This document shares how to validate IPv4/IPv6 dual-stack enabled Kubernetes clusters.

- [Before you begin](#)
- [Validate addressing](#)
- [Validate Services](#)

# Before you begin

- Provider support for dual-stack networking (Cloud provider or otherwise must be able to provide Kubernetes nodes with routable IPv4/IPv6 network interfaces)
- A [network plugin](#) that supports dual-stack (such as Kubenet or Calico)
- Kube-proxy running in mode IPVS
- [Dual-stack enabled](#) cluster

Your Kubernetes server must be at or later than version v1.16. To check the version, enter `kubectl version`.

## Validate addressing

### Validate node addressing

Each dual-stack Node should have a single IPv4 block and a single IPv6 block allocated. Validate that IPv4/IPv6 Pod address ranges are configured by running the following command. Replace the sample node name with a valid dual-stack Node from your cluster. In this example, the Node's name is k8s-linuxpool1-34450317-0:

```
kubectl get nodes k8s-linuxpool1-34450317-0 -o go-template --template='{{range .spec.podCIDRs}}{{printf "%s\n" .}}{{end}}'  
10.244.1.0/24  
a00:100::/24
```

There should be one IPv4 block and one IPv6 block allocated.

Validate that the node has an IPv4 and IPv6 interface detected (replace node name with a valid node from the cluster. In this example the node name is k8s-linuxpool1-34450317-0):

```
kubectl get nodes k8s-linuxpool1-34450317-0 -o go-template --template='{{range .status.addresses}}{{printf "%s: %s\n" .type .address}}{{end}}'  
Hostname: k8s-linuxpool1-34450317-0  
InternalIP: 10.240.0.5  
InternalIP: 2001:1234:5678:9abc::5
```

### Validate Pod addressing

Validate that a Pod has an IPv4 and IPv6 address assigned. (replace the Pod name with a valid Pod in your cluster. In this example the Pod name is pod01)

```
kubectl get pods pod01 -o go-template --template='{{range .status.podIPs}}{{printf "%s\n" .ip}}{{end}}'
```

```
10.244.1.4  
a00:100::4
```

You can also validate Pod IPs using the Downward API via the `status.podIPs` fieldPath. The following snippet demonstrates how you can expose the Pod IPs via an environment variable called `MY_POD_IPS` within a container.

```
env:  
- name: MY_POD_IPS  
  valueFrom:  
    fieldRef:  
      fieldPath: status.podIPs
```

The following command prints the value of the `MY_POD_IPS` environment variable from within a container. The value is a comma separated list that corresponds to the Pod's IPv4 and IPv6 addresses.

```
kubectl exec -it pod01 -- set | grep MY_POD_IPS
```

```
MY_POD_IPS=10.244.1.4,a00:100::4
```

The Pod's IP addresses will also be written to `/etc/hosts` within a container. The following command executes a cat on `/etc/hosts` on a dual stack Pod. From the output you can verify both the IPv4 and IPv6 IP address for the Pod.

```
kubectl exec -it pod01 -- cat /etc/hosts
```

```
# Kubernetes-managed hosts file.  
127.0.0.1 localhost  
::1 localhost ip6-localhost ip6-loopback  
fe00::0 ip6-localnet  
fe00::0 ip6-mcastprefix  
fe00::1 ip6-allnodes  
fe00::2 ip6-allrouters  
10.244.1.4 pod01  
a00:100::4 pod01
```

## Validate Services

Create the following Service without the `ipFamily` field set. When this field is not set, the Service gets an IP from the first configured range via `--service-cluster-ip-range` flag on the kube-controller-manager.

### [`service/networking/dual-stack-default-svc.yaml`](#)

```
apiVersion: v1
kind: Service
metadata:
  name: my-service
spec:
  selector:
    app: MyApp
  ports:
    - protocol: TCP
      port: 80
      targetPort: 9376
```

By viewing the YAML for the Service you can observe that the Service has the `ipFamily` field has set to reflect the address family of the first configured range set via `--service-cluster-ip-range` flag on kube-controller-manager.

```
kubectl get svc my-service -o yaml
```

```
apiVersion: v1
kind: Service
metadata:
  creationTimestamp: "2019-09-03T20:45:13Z"
  labels:
    app: MyApp
  name: my-service
  namespace: default
  resourceVersion: "485836"
  selfLink: /api/v1/namespaces/default/services/my-service
  uid: b6fa83ef-fe7e-47a3-96a1-ac212fa5b030
spec:
  clusterIP: 10.0.29.179
  ipFamily: IPv4
  ports:
    - port: 80
      protocol: TCP
      targetPort: 9376
  selector:
    app: MyApp
  sessionAffinity: None
  type: ClusterIP
status:
  loadBalancer: {}
```

Create the following Service with the `ipFamily` field set to IPv6.

### [service/networking/dual-stack-ipv6-svc.yaml](#)

```
apiVersion: v1
kind: Service
metadata:
  name: my-service
spec:
  ipFamily: IPv6
  selector:
    app: MyApp
  ports:
    - protocol: TCP
      port: 80
      targetPort: 9376
```

Validate that the Service gets a cluster IP address from the IPv6 address block. You may then validate access to the service via the IP and port.

```
kubectl get svc -l app=MyApp
NAME           TYPE        CLUSTER-IP      EXTERNAL-IP
PORT(S)        AGE
my-service     ClusterIP   fe80:20d::d06b <none>       80/
TCP          9s
```

## Create a dual-stack load balanced Service

If the cloud provider supports the provisioning of IPv6 enabled external load balancer, create the following Service with both the `ipFamily` field set to IPv6 and the `type` field set to LoadBalancer

### [service/networking/dual-stack-ipv6-lb-svc.yaml](#)

```
apiVersion: v1
kind: Service
metadata:
  name: my-service
  labels:
    app: MyApp
spec:
  ipFamily: IPv6
  type: LoadBalancer
  selector:
    app: MyApp
  ports:
    - protocol: TCP
      port: 80
      targetPort: 9376
```

Validate that the Service receives a CLUSTER-IP address from the IPv6 address block along with an EXTERNAL-IP. You may then validate access to the service via the IP and port.

```
kubectl get svc -l app=MyApp
NAME      TYPE        CLUSTER-IP      EXTERNAL-
IP          PORT(S)      AGE
my-service ClusterIP  fe80:20d::d06b
2001:db8:f100:4002::9d37:c0d7  80:31868/TCP  30s
```

## Feedback

Was this page helpful?

Yes No

Thanks for the feedback. If you have a specific, answerable question about how to use Kubernetes, ask it on [Stack Overflow](#). Open an issue in the GitHub repo if you want to [report a problem](#) or [suggest an improvement](#).

---

[Create an Issue](#) [Edit This Page](#)

Page last modified on February 03, 2020 at 10:09 AM PST by [Fix that dual-stack does not require Kubenet specifically \(#18924\)](#) ([Page History](#))

[Edit This Page](#)

# Extend kubectl with plugins

This guide demonstrates how to install and write extensions for [kubectl](#). By thinking of core `kubectl` commands as essential building blocks for interacting with a Kubernetes cluster, a cluster administrator can think of plugins as a means of utilizing these building blocks to create more complex behavior. Plugins extend `kubectl` with new sub-commands, allowing for new and custom features not included in the main distribution of `kubectl`.

- [Before you begin](#)

- [Installing kubectl plugins](#)
- [Writing kubectl plugins](#)
- [Distributing kubectl plugins](#)
- [What's next](#)

## Before you begin

You need to have a working `kubectl` binary installed.

## Installing kubectl plugins

A plugin is nothing more than a standalone executable file, whose name begins with `kubectl-`. To install a plugin, simply move its executable file to anywhere on your PATH.

You can also discover and install `kubectl` plugins available in the open source using [Krew](#). Krew is a plugin manager maintained by the Kubernetes SIG CLI community.

**Caution:** `Kubectl` plugins available via the [Krew plugin index](#) are not audited for security. You should install and run third-party plugins at your own risk, since they are arbitrary programs running on your machine.

## Discovering plugins

`kubectl` provides a command `kubectl plugin list` that searches your PATH for valid plugin executables. Executing this command causes a traversal of all files in your PATH. Any files that are executable, and begin with `kubectl-` will show up *in the order in which they are present in your PATH* in this command's output. A warning will be included for any files beginning with `kubectl-` that are *not* executable. A warning will also be included for any valid plugin files that overlap each other's name.

You can use [Krew](#) to discover and install `kubectl` plugins from a community-curated [plugin index](#).

## Limitations

It is currently not possible to create plugins that overwrite existing `kubectl` commands. For example, creating a plugin `kubectl-version` will cause that plugin to never be executed, as the existing `kubectl version` command will always take precedence over it. Due to this limitation, it is also *not* possible to use plugins to add new subcommands to existing `kubectl` commands. For example, adding a subcommand `kubectl create foo` by naming your plugin `kubectl-create-foo` will cause that plugin to be ignored.

`kubectl plugin list` shows warnings for any valid plugins that attempt to do this.

## Writing kubectl plugins

You can write a plugin in any programming language or script that allows you to write command-line commands.

There is no plugin installation or pre-loading required. Plugin executables receive the inherited environment from the `kubectl` binary. A plugin determines which command path it wishes to implement based on its name. For example, a plugin wanting to provide a new command `kubectl foo`, would simply be named `kubectl-foo`, and live somewhere in your PATH.

### Example plugin

```
#!/bin/bash

# optional argument handling
if [[ "$1" == "version" ]]
then
    echo "1.0.0"
    exit 0
fi

# optional argument handling
if [[ "$1" == "config" ]]
then
    echo "$KUBECONFIG"
    exit 0
fi

echo "I am a plugin named kubectl-foo"
```

### Using a plugin

To use the above plugin, simply make it executable:

```
sudo chmod +x ./kubectl-foo
```

and place it anywhere in your PATH:

```
sudo mv ./kubectl-foo /usr/local/bin
```

You may now invoke your plugin as a `kubectl` command:

```
kubectl foo
```

```
I am a plugin named kubectl-foo
```

All args and flags are passed as-is to the executable:

```
kubectl foo version
```

```
1.0.0
```

All environment variables are also passed as-is to the executable:

```
export KUBECONFIG=~/ kube/config  
kubectl foo config
```

```
/home/<user>/ kube/config
```

```
KUBECONFIG=/etc/kube/config kubectl foo config
```

```
/etc/kube/config
```

Additionally, the first argument that is passed to a plugin will always be the full path to the location where it was invoked (`$0` would equal `/usr/local/bin/kubectl-foo` in the example above).

## Naming a plugin

As seen in the example above, a plugin determines the command path that it will implement based on its filename. Every sub-command in the command path that a plugin targets, is separated by a dash (-). For example, a plugin that wishes to be invoked whenever the command `kubectl foo bar baz` is invoked by the user, would have the filename of `kubectl-foo-bar-baz`.

## Flags and argument handling

### Note:

The plugin mechanism does *not* create any custom, plugin-specific values or environment variables for a plugin process.

An older `kubectl` plugin mechanism provided environment variables such as `KUBECTL_PLUGINS_CURRENT_NAMESPACE`; that no longer happens.

`kubectl` plugins must parse and validate all of the arguments passed to them. See [using the command line runtime package](#) for details of a Go library aimed at plugin authors.

Here are some additional cases where users invoke your plugin while providing additional flags and arguments. This builds upon the the `kubectl-foo-bar-baz` plugin from the scenario above.

If you run `kubectl foo bar baz arg1 --flag=value arg2`, `kubectl`'s plugin mechanism will first try to find the plugin with the longest possible name, which in this case would be `kubectl-foo-bar-baz-arg1`. Upon not finding that plugin, `kubectl` then treats the last dash-separated value as an argument (`arg1` in this case), and attempts to find the next longest possible name, `kubectl-foo-bar-baz`. Upon

having found a plugin with this name, kubectl then invokes that plugin, passing all args and flags after the plugin's name as arguments to the plugin process.

Example:

```
# create a plugin
echo -e '#!/bin/bash\n\n#echo "My first command-line argument
was $1"' > kubectl-foo-bar-baz
sudo chmod +x ./kubectl-foo-bar-baz

# "install" your plugin by moving it to a directory in your
$PATH
sudo mv ./kubectl-foo-bar-baz /usr/local/bin

# check that kubectl recognizes your plugin
kubectl plugin list
```

The following kubectl-compatible plugins are available:

```
/usr/local/bin/kubectl-foo-bar-baz
```

```
# test that calling your plugin via a "kubectl" command works
# even when additional arguments and flags are passed to your
# plugin executable by the user.
kubectl foo bar baz arg1 --meaningless-flag=true
```

```
My first command-line argument was arg1
```

As you can see, your plugin was found based on the kubectl command specified by a user, and all extra arguments and flags were passed as-is to the plugin executable once it was found.

## Names with dashes and underscores

Although the kubectl plugin mechanism uses the dash (-) in plugin filenames to separate the sequence of sub-commands processed by the plugin, it is still possible to create a plugin command containing dashes in its commandline invocation by using underscores (\_) in its filename.

Example:

```
# create a plugin containing an underscore in its filename
echo -e '#!/bin/bash\n\n#echo "I am a plugin with a dash in
my name"' > ./kubectl-foo_bar
sudo chmod +x ./kubectl-foo_bar

# move the plugin into your $PATH
sudo mv ./kubectl-foo_bar /usr/local/bin

# You can now invoke your plugin via kubectl:
kubectl foo-bar
```

```
I am a plugin with a dash in my name
```

Note that the introduction of underscores to a plugin filename does not prevent you from having commands such as `kubectl foo_bar`. The command from the above example, can be invoked using either a dash (-) or an underscore (\_):

```
# You can invoke your custom command with a dash  
kubectl foo-bar
```

```
I am a plugin with a dash in my name
```

```
# You can also invoke your custom command with an underscore  
kubectl foo_bar
```

```
I am a plugin with a dash in my name
```

## Name conflicts and overshadowing

It is possible to have multiple plugins with the same filename in different locations throughout your PATH. For example, given a PATH with the following value: `PATH=/usr/local/bin/plugins:/usr/local/bin/moreplugins`, a copy of plugin `kubectl-foo` could exist in `/usr/local/bin/plugins` and `/usr/local/bin/moreplugins`, such that the output of the `kubectl plugin list` command is:

```
PATH=/usr/local/bin/plugins:/usr/local/bin/moreplugins  
kubectl plugin list
```

The following kubectl-compatible plugins are available:

```
/usr/local/bin/plugins/kubectl-foo  
/usr/local/bin/moreplugins/kubectl-foo  
    - warning: /usr/local/bin/moreplugins/kubectl-foo is  
overshadowed by a similarly named plugin: /usr/local/bin/  
plugins/kubectl-foo  
  
error: one plugin warning was found
```

In the above scenario, the warning under `/usr/local/bin/moreplugins/kubectl-foo` tells you that this plugin will never be executed. Instead, the executable that appears first in your PATH, `/usr/local/bin/plugins/kubectl-foo`, will always be found and executed first by the `kubectl` plugin mechanism.

A way to resolve this issue is to ensure that the location of the plugin that you wish to use with `kubectl` always comes first in your PATH. For example, if you want to always use `/usr/local/bin/moreplugins/kubectl-foo` anytime that the `kubectl` command `kubectl foo` was invoked, change the value of your PATH to be `/usr/local/bin/moreplugins:/usr/local/bin/plugins`.

## Invocation of the longest executable filename

There is another kind of overshadowing that can occur with plugin filenames. Given two plugins present in a user's PATH: `kubectl-foo-bar` and `kubectl-foo-bar-baz`, the `kubectl` plugin mechanism will always choose the longest possible plugin name for a given user command. Some examples below, clarify this further:

```
# for a given kubectl command, the plugin with the longest
possible filename will always be preferred
kubectl foo bar baz
```

Plugin `kubectl-foo-bar-baz` is executed

```
kubectl foo bar
```

Plugin `kubectl-foo-bar` is executed

```
kubectl foo bar baz buz
```

Plugin `kubectl-foo-bar-baz` is executed, with "buz" as its first argument

```
kubectl foo bar buz
```

Plugin `kubectl-foo-bar` is executed, with "buz" as its first argument

This design choice ensures that plugin sub-commands can be implemented across multiple files, if needed, and that these sub-commands can be nested under a "parent" plugin command:

```
ls ./plugin_command_tree
```

```
kubectl-parent
kubectl-parent-subcommand
kubectl-parent-subcommand-subsubcommand
```

## Checking for plugin warnings

You can use the aforementioned `kubectl plugin list` command to ensure that your plugin is visible by `kubectl`, and verify that there are no warnings preventing it from being called as a `kubectl` command.

```
kubectl plugin list
```

The following `kubectl`-compatible plugins are available:

```
test/fixtures/pkg/kubectl/plugins/kubectl-foo
/usr/local/bin/kubectl-foo
    - warning: /usr/local/bin/kubectl-foo is overshadowed by a
similarly named plugin: test/fixtures/pkg/kubectl/plugins/
kubectl-foo
```

```
plugins/kubectl-invalid
  - warning: plugins/kubectl-invalid identified as a kubectl
    plugin, but it is not executable

error: 2 plugin warnings were found
```

## Using the command line runtime package

If you're writing a plugin for kubectl and you're using Go, you can make use of the [cli-runtime](#) utility libraries.

These libraries provide helpers for parsing or updating a user's [kubeconfig](#) file, for making REST-style requests to the API server, or to bind flags associated with configuration and printing.

See the [Sample CLI Plugin](#) for an example usage of the tools provided in the CLI Runtime repo.

## Distributing kubectl plugins

If you have developed a plugin for others to use, you should consider how you package it, distribute it and deliver updates to your users.

### Krew

[Krew](#) offers a cross-platform way to package and distribute your plugins. This way, you use a single packaging format for all target platforms (Linux, Windows, macOS etc) and deliver updates to your users. Krew also maintains a [plugin index](#) so that other people can discover your plugin and install it.

### Native / platform specific package management

Alternatively, you can use traditional package managers such as, apt or yum on Linux, Chocolatey on Windows, and Homebrew on macOS. Any package manager will be suitable if it can place new executables placed somewhere in the user's PATH. As a plugin author, if you pick this option then you also have the burden of updating your kubectl plugin's distribution package across multiple platforms for each release.

### Source code

You can publish the source code; for example, as a Git repository. If you choose this option, someone who wants to use that plugin must fetch the code, set up a build environment (if it needs compiling), and deploy the plugin. If you also make compiled packages available, or use Krew, that will make installs easier.

## What's next

- Check the Sample CLI Plugin repository for a [detailed example](#) of a plugin written in Go. In case of any questions, feel free to reach out to the [SIG CLI team](#).
- Read about [Krew](#), a package manager for kubectl plugins.

## Feedback

Was this page helpful?

Yes No

Thanks for the feedback. If you have a specific, answerable question about how to use Kubernetes, ask it on [Stack Overflow](#). Open an issue in the GitHub repo if you want to [report a problem](#) or [suggest an improvement](#).

---

[Create an Issue](#) [Edit This Page](#)

Page last modified on January 12, 2020 at 5:31 PM PST by [kubectl-plugin example: Double-quote to prevent word splitting \(#18081\)](#) ([Page History](#))

[Edit This Page](#)

## Manage HugePages

**FEATURE STATE:** Kubernetes v1.17 [stable](#)

This feature is *stable*, meaning:

[Edit This Page](#)

## Schedule GPUs

**FEATURE STATE:** Kubernetes 1.10 [beta](#)

This feature is currently in a *beta* state, meaning: