

[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
- Administering Federation
- 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.

Administering Federation

Configure components in a cluster federation.

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](#).

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

[Edit This Page](#)

Installing kubeadm



This page shows how to install the `kubeadm` toolbox. For information how to create a cluster with `kubeadm` once you have performed this installation process, see the [Using kubeadm to Create a Cluster](#) page.

- Before you begin
- Verify the MAC address and product_uuid are unique for every node
- Check network adapters
- Check required ports
- Installing Docker
- Installing kubeadm, kubelet and kubectl
- Configure cgroup driver used by kubelet on Master Node
- Troubleshooting
- What's next

Before you begin

- One or more machines running one of:
 - Ubuntu 16.04+
 - Debian 9
 - CentOS 7
 - RHEL 7
 - Fedora 25/26 (best-effort)
 - HypriotOS v1.0.1+
 - Container Linux (tested with 1576.4.0)
- 2 GB or more of RAM per machine (any less will leave little room for your apps)
- 2 CPUs or more
- Full network connectivity between all machines in the cluster (public or private network is fine)
- Unique hostname, MAC address, and product_uuid for every node. See [here](#) for more details.

- Certain ports are open on your machines. See here for more details.
- Swap disabled. You **MUST** disable swap in order for the kubelet to work properly.

Verify the MAC address and product_uuid are unique for every node

- You can get the MAC address of the network interfaces using the command `ip link` or `ifconfig -a`
- The `product_uuid` can be checked by using the command `sudo cat /sys/class/dmi/id/product_uuid`

It is very likely that hardware devices will have unique addresses, although some virtual machines may have identical values. Kubernetes uses these values to uniquely identify the nodes in the cluster. If these values are not unique to each node, the installation process may fail.

Check network adapters

If you have more than one network adapter, and your Kubernetes components are not reachable on the default route, we recommend you add IP route(s) so Kubernetes cluster addresses go via the appropriate adapter.

Check required ports

Master node(s)

Protocol	Direction	Port Range	Purpose
TCP	Inbound	6443*	Kubernetes API server
TCP	Inbound	2379-2380	etcd server client API
TCP	Inbound	10250	Kubelet API
TCP	Inbound	10251	kube-scheduler
TCP	Inbound	10252	kube-controller-manager
TCP	Inbound	10255	Read-only Kubelet API

Worker node(s)

Protocol	Direction	Port Range	Purpose
TCP	Inbound	10250	Kubelet API
TCP	Inbound	10255	Read-only Kubelet API

Protocol	Direction	Port Range	Purpose
TCP	Inbound	30000-32767	NodePort Services**

** Default port range for NodePort Services.

Any port numbers marked with * are overridable, so you will need to ensure any custom ports you provide are also open.

Although etcd ports are included in master nodes, you can also host your own etcd cluster externally or on custom ports.

The pod network plugin you use (see below) may also require certain ports to be open. Since this differs with each pod network plugin, please see the documentation for the plugins about what port(s) those need.

Installing Docker

On each of your machines, install Docker. Version 17.03 is recommended, but 1.11, 1.12 and 1.13 are known to work as well. Versions 17.06+ *might work*, but have not yet been tested and verified by the Kubernetes node team. Keep track of the latest verified Docker version in the Kubernetes release notes.

Please proceed with executing the following commands based on your OS as root. You may become the root user by executing `sudo -i` after SSH-ing to each host.

If you already have the required versions of the Docker installed, you can move on to next section. If not, you can use the following commands to install Docker on your system:

- Ubuntu, Debian or HypriotOS
- CentOS, RHEL or Fedora
- Container Linux

Install Docker from Ubuntu's repositories:

```
apt-get update
apt-get install -y docker.io
```

or install Docker CE 17.03 from Docker's repositories for Ubuntu or Debian:

```
apt-get update
apt-get install -y apt-transport-https ca-certificates curl software-properties-common
curl -fsSL https://download.docker.com/linux/ubuntu/gpg | apt-key add -
add-apt-repository "deb https://download.docker.com/linux/$(. /etc/os-release; echo "$ID") stable"
apt-get update && apt-get install -y docker-ce=$(apt-cache madison docker-ce | grep 17.03 |
```

Install Docker using your operating system's bundled package:

```
yum install -y docker  
systemctl enable docker && systemctl start docker
```

Enable and start Docker:

```
systemctl enable docker && systemctl start docker
```

Refer to the official Docker installation guides for more information.

Installing kubeadm, kubelet and kubectl

You will install these packages on all of your machines:

- **kubeadm**: the command to bootstrap the cluster.
- **kubelet**: the component that runs on all of the machines in your cluster and does things like starting pods and containers.
- **kubectl**: the command line util to talk to your cluster.

kubeadm **will not** install or manage **kubelet** or **kubectl** for you, so you will need to ensure they match the version of the Kubernetes control panel you want kubeadm to install for you. If you do not, there is a risk of a version skew occurring that can lead to unexpected, buggy behaviour. However, *one* minor version skew between the kubelet and the control plane is supported, but the kubelet version may never exceed the API server version. For example, kubelets running 1.7.0 should be fully compatible with a 1.8.0 API server, but not vice versa.

For more information on version skews, please read our version skew policy.

- Ubuntu, Debian or HypriotOS
- CentOS, RHEL or Fedora
- Container Linux

```
apt-get update && apt-get install -y apt-transport-https curl  
curl -s https://packages.cloud.google.com/apt/doc/apt-key.gpg | apt-key add -  
cat <<EOF >/etc/apt/sources.list.d/kubernetes.list  
deb http://apt.kubernetes.io/ kubernetes-xenial main  
EOF  
apt-get update  
apt-get install -y kubelet kubeadm kubectl  
  
cat <<EOF > /etc/yum.repos.d/kubernetes.repo  
[kubernetes]  
name=Kubernetes  
baseurl=https://packages.cloud.google.com/yum/repos/kubernetes-el7-x86_64  
enabled=1  
gpgcheck=1  
repo_gpgcheck=1
```

```
gpgkey=https://packages.cloud.google.com/yum/doc/yum-key.gpg https://packages.cloud.google.com/yum/doc/yum-key.asc
EOF
setenforce 0
yum install -y kubelet kubeadm kubectl
systemctl enable kubelet && systemctl start kubelet
```

Note:

- Disabling SELinux by running `setenforce 0` is required to allow containers to access the host filesystem, which is required by pod networks for example. You have to do this until SELinux support is improved in the kubelet.
- Some users on RHEL/CentOS 7 have reported issues with traffic being routed incorrectly due to iptables being bypassed. You should ensure `net.bridge.bridge-nf-call-iptables` is set to 1 in your `sysctl` config, e.g.

```
cat <<EOF > /etc/sysctl.d/k8s.conf
net.bridge.bridge-nf-call-ip6tables = 1
net.bridge.bridge-nf-call-iptables = 1
EOF
sysctl --system
```

Install CNI plugins (required for most pod network):

```
CNI_VERSION="v0.6.0"
mkdir -p /opt/cni/bin
curl -L "https://github.com/containernetworking/plugins/releases/download/${CNI_VERSION}/cni-plugins-v${CNI_VERSION}.tgz" | tar -x -C /opt/cni/bin
```

Install kubeadm, kubelet, kubectl and add a kubelet systemd service:

```
RELEASE="$(curl -sSL https://dl.k8s.io/release/stable.txt)"
```

```
mkdir -p /opt/bin
cd /opt/bin
curl -L --remote-name-all https://storage.googleapis.com/kubernetes-release/release/${RELEASE}/bin/linux/amd64/{kubeadm,kubelet,kubectl}
chmod +x {kubeadm,kubelet,kubectl}

curl -sSL "https://raw.githubusercontent.com/kubernetes/kubernetes/${RELEASE}/build/debs/kubelet_${RELEASE}_amd64.deb"
curl -sSL "https://raw.githubusercontent.com/kubernetes/kubernetes/${RELEASE}/build/debs/kubelet.service.d/kubelet.conf"
curl -sSL "https://raw.githubusercontent.com/kubernetes/kubernetes/${RELEASE}/build/debs/100-kubelet.conf"
```

Enable and start kubelet:

```
systemctl enable kubelet && systemctl start kubelet
```

The kubelet is now restarting every few seconds, as it waits in a crashloop for kubeadm to tell it what to do.

Configure cgroup driver used by kubelet on Master Node

Make sure that the cgroup driver used by kubelet is the same as the one used by Docker. Verify that your Docker cgroup driver matches the kubelet config:

```
docker info | grep -i cgroup  
cat /etc/systemd/system/kubelet.service.d/10-kubeadm.conf
```

If the Docker cgroup driver and the kubelet config don't match, change the kubelet config to match the Docker cgroup driver. The flag you need to change is `--cgroup-driver`. If it's already set, you can update like so:

```
sed -i "s/cgroup-driver=systemd/cgroup-driver=cgroupfs/g" /etc/systemd/system/kubelet.service
```

Otherwise, you will need to open the systemd file and add the flag to an existing environment line.

Then restart kubelet:

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

Troubleshooting

If you are running into difficulties with kubeadm, please consult our troubleshooting docs.

What's next

- Using kubeadm to Create a Cluster

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

[Edit This Page](#)

Install and Set Up kubectl

Use the Kubernetes command-line tool, kubectl, to deploy and manage applications on Kubernetes. Using kubectl, you can inspect cluster resources; create, delete, and update components; and look at your new cluster and bring up example apps.

- Before you begin
- Install kubectl
- Install kubectl binary via native package management

- Install with snap on Ubuntu
- Install with Homebrew on macOS
- Install with Powershell from PSGallery
- Install with Chocolatey on Windows
- Download as part of the Google Cloud SDK
- Install kubectl binary via curl
- Configure kubectl
- Check the kubectl configuration
- Enabling shell autocompletion
- What's next

Before you begin

Use a version of kubectl that is the same version as your server or later. Using an older kubectl with a newer server might produce validation errors.

Install kubectl

Here are a few methods to install kubectl.

Install kubectl binary via 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 -
sudo touch /etc/apt/sources.list.d/kubernetes.list
echo "deb http://apt.kubernetes.io/ kubernetes-xenial main" | sudo tee -a /etc/apt/sources.list
sudo apt-get update
sudo apt-get install -y kubectl

cat <<EOF > /etc/yum.repos.d/kubernetes.repo
[kubernetes]
name=Kubernetes
baseurl=https://packages.cloud.google.com/yum/repos/kubernetes-el7-x86_64
enabled=1
gpgcheck=1
repo_gpgcheck=1
gpgkey=https://packages.cloud.google.com/yum/doc/yum-key.gpg https://packages.cloud.google.com/yum/doc/yum-key.gpg
EOF
yum install -y kubectl
```

Install with snap on Ubuntu

kubectl is available as a snap application.

1. If you are on Ubuntu or one of other Linux distributions that support snap package manager, you can install with:

```
sudo snap install kubectl --classic
```

2. Run `kubectl version` to verify that the version you've installed is sufficiently up-to-date.

Install with Homebrew on macOS

1. If you are on macOS and using Homebrew package manager, you can install with:

```
brew install kubectl
```

2. Run `kubectl version` to verify that the version you've installed is sufficiently up-to-date.

Install with Powershell from PSGallery

1. If you are on Windows and using Powershell Gallery package manager, you can install and update with:

```
Install-Script -Name install-kubectl -Scope CurrentUser -Force  
install-kubectl.ps1 [-DownloadLocation <path>]
```

If no Downloadlocation is specified, kubectl will be installed in users temp Directory 2. The installer creates \$HOME/.kube and instructs it to create a config file 3. Updating re-run Install-Script to update the installer re-run install-kubectl.ps1 to install latest binaries

Install with Chocolatey on Windows

1. If you are on Windows and using Chocolatey package manager, you can install with:

```
choco install kubernetes-cli
```

2. Run `kubectl version` to verify that the version you've installed is sufficiently up-to-date.

3. Configure kubectl to use a remote Kubernetes cluster:

```
cd C:\users\yourusername (Or wherever your %HOME% directory is)
mkdir .kube
cd .kube
New-Item config -type file
```

Edit the config file with a text editor of your choice, such as Notepad for example.

[Download as part of the Google Cloud SDK](#)

kubectl can be installed as part of the Google Cloud SDK.

1. Install the Google Cloud SDK.
 2. Run the following command to install kubectl:

```
gcloud components install kubectl
```

3. Run `kubectl version` to verify that the version you've installed is sufficiently up-to-date.

Install kubectl binary via curl

- macOS
 - Linux
 - Windows

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)
```

To download a specific version, replace the `$(curl` portion of the command with the specific version.

For example, to download version v1.10.3 on MacOS, type:

```
curl -LO https://storage.googleapis.com/kubernetes-release/release/v1.10.3/bin/darwin/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
```

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)
```

To download a specific version, replace the `$(curl` portion of the command with the specific version.

For example, to download version v1.10.3 on Linux, type:

```

curl -LO https://storage.googleapis.com/kubernetes-release/release/v1.10.3/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
1. Download the latest release v1.10.3 from this link.
    Or if you have curl installed, use this command:
    curl -LO https://storage.googleapis.com/kubernetes-release/release/v1.10.3/bin/windows/amd64/kubectl.exe
    To find out the latest stable version (for example, for scripting), take a look
    at https://storage.googleapis.com/kubernetes-release/release/stable.txt.
2. Add the binary in to your PATH.

```

Configure kubectl

In order for kubectl to find and access a Kubernetes cluster, it needs a kubeconfig file, which is created automatically when you create a cluster using kube-up.sh or successfully deploy a Minikube cluster. See the getting started guides for more about creating clusters. If you need access to a cluster you didn't create, see the Sharing Cluster Access document. By default, kubectl configuration is located at `~/.kube/config`.

Check the kubectl configuration

Check that kubectl is properly configured by getting the cluster state:

```
kubectl cluster-info
```

If you see a URL response, kubectl is correctly configured to access your cluster.

If you see a message similar to the following, kubectl is not correctly configured or not able to connect to a Kubernetes cluster.

```
The connection to the server <server-name:port> was refused - did you specify the right host?
```

For example, if you are intending to run a Kubernetes cluster on your laptop (locally), you will need a tool like minikube to be installed first and then re-run the commands stated above.

If kubectl cluster-info returns the url response but you can't access your cluster, to check whether it is configured properly, use:

```
kubectl cluster-info dump
```

Enabling shell completion

kubectl includes completion support, which can save a lot of typing!

The completion script itself is generated by kubectl, so you typically just need to invoke it from your profile.

Common examples are provided here. For more details, consult `kubectl completion -h`.

On Linux, using bash

On CentOS Linux, you may need to install the bash-completion package which is not installed by default.

```
yum install bash-completion -y
```

To add kubectl completion to your current shell, run `source <(kubectl completion bash)`.

To add kubectl completion to your profile, so it is automatically loaded in future shells run:

```
echo "source <(kubectl completion bash)" >> ~/.bashrc
```

On macOS, using bash

On macOS, you will need to install bash-completion support via Homebrew first:

```
## If running Bash 3.2 included with macOS
brew install bash-completion
## or, if running Bash 4.1+
brew install bash-completion@2
```

Follow the “caveats” section of brew’s output to add the appropriate bash completion path to your local `.bashrc`.

If you’ve installed kubectl using the Homebrew instructions then kubectl completion should start working immediately.

If you have installed kubectl manually, you need to add kubectl completion to the bash-completion:

```
kubectl completion bash > $(brew --prefix)/etc/bash_completion.d/kubectl
```

The Homebrew project is independent from Kubernetes, so the bash-completion packages are not guaranteed to work.

Using Zsh

If you are using zsh edit the `~/.zshrc` file and add the following code to enable kubectl autocompletion:

```
if [ $commands[kubectl] ]; then
    source <(kubectl completion zsh)
fi
```

Or when using Oh-My-Zsh, edit the `~/.zshrc` file and update the `plugins=` line to include the kubectl plugin.

```
source <(kubectl completion zsh)
```

What's next

Learn how to launch and expose your application.

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

[Edit This Page](#)

Install Minikube

This page shows how to install Minikube.

- Before you begin
- Install a Hypervisor
- Install kubectl
- Install Minikube
- What's next

Before you begin

VT-x or AMD-v virtualization must be enabled in your computer's BIOS.

Install a Hypervisor

If you do not already have a hypervisor installed, install one now.

- For OS X, install VirtualBox or VMware Fusion, or HyperKit.
- For Linux, install VirtualBox or KVM.

Note: Minikube also supports a `--vm-driver=none` option that runs the Kubernetes components on the host and not in a VM. Docker is required to use this driver but a hypervisor is not required.

- For Windows, install VirtualBox or Hyper-V.

Install kubectl

- Install kubectl.

Install Minikube

- Install Minikube according to the instructions for the latest release.

What's next

- Running Kubernetes Locally via Minikube

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

[Edit This Page](#)

Installing kubeadm



This page shows how to install the `kubeadm` toolbox. For information how to create a cluster with `kubeadm` once you have performed this installation process, see the [Using `kubeadm` to Create a Cluster](#) page.

- Before you begin
- Verify the MAC address and product_uuid are unique for every node

- Check network adapters
- Check required ports
- Installing Docker
- Installing kubeadm, kubelet and kubectl
- Configure cgroup driver used by kubelet on Master Node
- Troubleshooting
- What's next

Before you begin

- One or more machines running one of:
 - Ubuntu 16.04+
 - Debian 9
 - CentOS 7
 - RHEL 7
 - Fedora 25/26 (best-effort)
 - HypriotOS v1.0.1+
 - Container Linux (tested with 1576.4.0)
- 2 GB or more of RAM per machine (any less will leave little room for your apps)
- 2 CPUs or more
- Full network connectivity between all machines in the cluster (public or private network is fine)
- Unique hostname, MAC address, and product_uuid for every node. See here for more details.
- Certain ports are open on your machines. See here for more details.
- Swap disabled. You **MUST** disable swap in order for the kubelet to work properly.

Verify the MAC address and product_uuid are unique for every node

- You can get the MAC address of the network interfaces using the command `ip link` or `ifconfig -a`
- The product_uuid can be checked by using the command `sudo cat /sys/class/dmi/id/product_uuid`

It is very likely that hardware devices will have unique addresses, although some virtual machines may have identical values. Kubernetes uses these values to uniquely identify the nodes in the cluster. If these values are not unique to each node, the installation process may fail.

Check network adapters

If you have more than one network adapter, and your Kubernetes components are not reachable on the default route, we recommend you add IP route(s) so Kubernetes cluster addresses go via the appropriate adapter.

Check required ports

Master node(s)

Protocol	Direction	Port Range	Purpose
TCP	Inbound	6443*	Kubernetes API server
TCP	Inbound	2379-2380	etcd server client API
TCP	Inbound	10250	Kubelet API
TCP	Inbound	10251	kube-scheduler
TCP	Inbound	10252	kube-controller-manager
TCP	Inbound	10255	Read-only Kubelet API

Worker node(s)

Protocol	Direction	Port Range	Purpose
TCP	Inbound	10250	Kubelet API
TCP	Inbound	10255	Read-only Kubelet API
TCP	Inbound	30000-32767	NodePort Services**

** Default port range for NodePort Services.

Any port numbers marked with * are overridable, so you will need to ensure any custom ports you provide are also open.

Although etcd ports are included in master nodes, you can also host your own etcd cluster externally or on custom ports.

The pod network plugin you use (see below) may also require certain ports to be open. Since this differs with each pod network plugin, please see the documentation for the plugins about what port(s) those need.

Installing Docker

On each of your machines, install Docker. Version 17.03 is recommended, but 1.11, 1.12 and 1.13 are known to work as well. Versions 17.06+ *might work*, but

have not yet been tested and verified by the Kubernetes node team. Keep track of the latest verified Docker version in the Kubernetes release notes.

Please proceed with executing the following commands based on your OS as root. You may become the root user by executing `sudo -i` after SSH-ing to each host.

If you already have the required versions of the Docker installed, you can move on to next section. If not, you can use the following commands to install Docker on your system:

- Ubuntu, Debian or HypriotOS
- CentOS, RHEL or Fedora
- Container Linux

Install Docker from Ubuntu's repositories:

```
apt-get update  
apt-get install -y docker.io
```

or install Docker CE 17.03 from Docker's repositories for Ubuntu or Debian:

```
apt-get update  
apt-get install -y apt-transport-https ca-certificates curl software-properties-common  
curl -fsSL https://download.docker.com/linux/ubuntu/gpg | apt-key add -  
add-apt-repository "deb https://download.docker.com/linux/$(. /etc/os-release; echo "$ID")" 9  
apt-get update && apt-get install -y docker-ce=$(apt-cache madison docker-ce | grep 17.03 |
```

Install Docker using your operating system's bundled package:

```
yum install -y docker  
systemctl enable docker && systemctl start docker
```

Enable and start Docker:

```
systemctl enable docker && systemctl start docker
```

Refer to the official Docker installation guides for more information.

Installing kubeadm, kubelet and kubectl

You will install these packages on all of your machines:

- **kubeadm**: the command to bootstrap the cluster.
- **kubelet**: the component that runs on all of the machines in your cluster and does things like starting pods and containers.
- **kubectl**: the command line util to talk to your cluster.

kubeadm **will not** install or manage **kubelet** or **kubectl** for you, so you will need to ensure they match the version of the Kubernetes control panel you want kubeadm to install for you. If you do not, there is a risk of a version skew

occurring that can lead to unexpected, buggy behaviour. However, *one* minor version skew between the kubelet and the control plane is supported, but the kubelet version may never exceed the API server version. For example, kubelets running 1.7.0 should be fully compatible with a 1.8.0 API server, but not vice versa.

For more information on version skews, please read our version skew policy.

- Ubuntu, Debian or HypriotOS
- CentOS, RHEL or Fedora
- Container Linux

```
apt-get update && apt-get install -y apt-transport-https curl
curl -s https://packages.cloud.google.com/apt/doc/apt-key.gpg | apt-key add -
cat <<EOF >/etc/apt/sources.list.d/kubernetes.list
deb http://apt.kubernetes.io/ kubernetes-xenial main
EOF
apt-get update
apt-get install -y kubelet kubeadm kubectl
cat <<EOF > /etc/yum.repos.d/kubernetes.repo
[kubernetes]
name=Kubernetes
baseurl=https://packages.cloud.google.com/yum/repos/kubernetes-e17-x86_64
enabled=1
gpgcheck=1
repo_gpgcheck=1
gpgkey=https://packages.cloud.google.com/yum/doc/yum-key.gpg https://packages.cloud.google.com/yum/doc/yum-key.gpg
EOF
setenforce 0
yum install -y kubelet kubeadm kubectl
systemctl enable kubelet && systemctl start kubelet
```

Note:

- Disabling SELinux by running `setenforce 0` is required to allow containers to access the host filesystem, which is required by pod networks for example. You have to do this until SELinux support is improved in the kubelet.
- Some users on RHEL/CentOS 7 have reported issues with traffic being routed incorrectly due to iptables being bypassed. You should ensure `net.bridge.bridge-nf-call-iptables` is set to 1 in your `sysctl` config, e.g.

```
cat <<EOF > /etc/sysctl.d/k8s.conf
net.bridge.bridge-nf-call-ip6tables = 1
net.bridge.bridge-nf-call-iptables = 1
EOF
sysctl --system
```

Install CNI plugins (required for most pod network):

```
CNI_VERSION="v0.6.0"
mkdir -p /opt/cni/bin
curl -L "https://github.com/containernetworking/plugins/releases/download/${CNI_VERSION}/cni-plugins-v${CNI_VERSION}.tgz" | tar -x -C /opt/cni/bin
```

Install kubeadm, kubelet, kubectl and add a kubelet systemd service:

```
RELEASE=$(curl -sSL https://dl.k8s.io/release/stable.txt)"
```

```
mkdir -p /opt/bin
cd /opt/bin
curl -L --remote-name-all https://storage.googleapis.com/kubernetes-release/release/${RELEASE}/bin/linux/amd64/{kubeadm,kubelet,kubectl}
chmod +x {kubeadm,kubelet,kubectl}
```

```
curl -sSL "https://raw.githubusercontent.com/kubernetes/kubernetes/${RELEASE}/build/debs/kubelet_${RELEASE}_amd64.deb"
mkdir -p /etc/systemd/system/kubelet.service.d
curl -sSL "https://raw.githubusercontent.com/kubernetes/kubernetes/${RELEASE}/build/debs/10-kubeadm.conf"
```

Enable and start kubelet:

```
systemctl enable kubelet && systemctl start kubelet
```

The kubelet is now restarting every few seconds, as it waits in a crashloop for kubeadm to tell it what to do.

Configure cgroup driver used by kubelet on Master Node

Make sure that the cgroup driver used by kubelet is the same as the one used by Docker. Verify that your Docker cgroup driver matches the kubelet config:

```
docker info | grep -i cgroup
cat /etc/systemd/system/kubelet.service.d/10-kubeadm.conf
```

If the Docker cgroup driver and the kubelet config don't match, change the kubelet config to match the Docker cgroup driver. The flag you need to change is `--cgroup-driver`. If it's already set, you can update like so:

```
sed -i "s/cgroup-driver=systemd/cgroup-driver=cgroupfs/g" /etc/systemd/system/kubelet.service
```

Otherwise, you will need to open the systemd file and add the flag to an existing environment line.

Then restart kubelet:

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

Troubleshooting

If you are running into difficulties with kubeadm, please consult our troubleshooting docs.

What's next

- Using kubeadm to Create a Cluster

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

[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. A Container is guaranteed to have as much CPU as it requests, but is not allowed to use more CPU than its limit.

- 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 don't 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 that the Heapster service is running in your cluster. But if you don't have Heapster running, you can do most of the steps, and it won't be a problem if you skip the Heapster steps.

If you are running minikube, run the following command to enable heapster:

```
minikube addons enable heapster
```

To see whether the Heapster service is running, enter this command:

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

If the heapster service is running, it shows in the output:

NAMESPACE	NAME	CLUSTER-IP	EXTERNAL-IP	PORT(S)	AGE
kube-system	heapster	10.11.240.9	<none>	80/TCP	6d

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 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's 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 CPU request of 0.5 cpu and a CPU limit of 1 cpu. Here's the configuration file for the Pod:

```
cpu-request-limit.yaml docs/tasks/configure-pod-container
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"
```

In the configuration file, the `args` section 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 create -f https://k8s.io/docs/tasks/configure-pod-container/cpu-request-limit.yaml
```

Verify that the Pod's Container 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
```

Start a proxy so that you can call the heapster service:

```
kubectl proxy
```

In another command window, get the CPU usage rate from the heapster service:

```
curl http://localhost:8001/api/v1/namespaces/kube-system/services/heapster/proxy/api/v1/mode
```

The 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's configuration file.

```
{  
  "timestamp": "2017-06-22T18:48:00Z",  
  "value": 974  
}
```

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: There's another possible explanation for the CPU throttling. The Node might not have enough CPU resources available. Recall that the prerequisites for this exercise require that each of your Nodes has at least 1 cpu. If your Container is running 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's 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.

```
cpu-request-limit-2.yaml docs/tasks/configure-pod-container
```

```
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 create -f https://k8s.io/docs/tasks/configure-pod-container/cpu-request-limit-2.yaml
```

View the Pod's status:

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

The output shows that the Pod's 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:: Insu

Delete your Pod:

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

If you don't specify a CPU limit

If you don't 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's Nodes. By keeping a Pod's 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

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

[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 don't 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 that the Heapster service is running in your cluster. But if you don't have Heapster running, you can do most of the steps, and it won't be a problem if you skip the Heapster steps.

If you are running minikube, run the following command to enable heapster:

```
minikube addons enable heapster
```

To see whether the Heapster service is running, enter this command:

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

If the Heapster service is running, it shows in the output:

NAMESPACE	NAME	CLUSTER-IP	EXTERNAL-IP	PORT(S)	AGE
kube-system	heapster	10.11.240.9	<none>	80/TCP	6d

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:

```
memory-request-limit.yaml docs/tasks/configure-pod-container
```

```
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"]
```

In the configuration file, the `args` section 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 create -f https://k8s.io/docs/tasks/configure-pod-container/memory-request-limit.yaml
```

Verify that the Pod's 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  
  ...
```

Start a proxy so that you can call the Heapster service:

```
kubectl proxy
```

In another command window, get the memory usage from the Heapster service:

```
curl http://localhost:8001/api/v1/namespaces/kube-system/services/heapster/proxy/api/v1/mode
```

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.

```
{  
  "timestamp": "2017-06-20T18:54:00Z",  
  "value": 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 is restartable, the kubelet will restart 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. The Container has a memory request of 50 MiB and a memory limit of 100 MiB.

```
memory-request-limit-2.yaml docs/tasks/configure-pod-container
```

```
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 configuration file, in the `args` section, 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 create -f https://k8s.io/docs/tasks/configure-pod-container/memory-request-limit-2.yaml
```

View detailed information about the Pod:

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

At this point, the Container might be running, or it might have been killed. If the Container has not yet been killed, repeat the preceding command until you see that the Container has been killed:

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

Get a more detailed view of the Container's status:

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

The output shows that the Container has been killed because it is out of memory (OOM).

```
lastState:
  terminated:
    containerID: docker://65183c1877aaec2e8427bc95609cc52677a454b56fc24340dbd22917c23b10f
```

```
exitCode: 137
finishedAt: 2017-06-20T20:52:19Z
reason: OOMKilled
startedAt: null
```

The Container in this exercise is restartable, so the kubelet will restart it. Enter this command several times to see that the Container gets repeatedly killed and restarted:

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

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

```
stevepe@sperry-1:~/steveperry-53.github.io$ kubectl get pod memory-demo-2 --namespace=mem-example
NAME        READY   STATUS    RESTARTS   AGE
memory-demo-2  0/1     OOMKilled  1          37s
stevepe@sperry-1:~/steveperry-53.github.io$ 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's 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 66a3a20aa7980e61be4922780bf9d24d1a1d8b7395c
... 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
```

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. The Container requests 1000 GiB of memory, which is likely to exceed the capacity of any Node in your cluster.

```
memory-request-limit-3.yaml docs/tasks/configure-pod-container
```

```
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 create -f https://k8s.io/docs/tasks/configure-pod-container/memory-request-limit-3.yaml
```

View the Pod's status:

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

The output shows that the Pod's 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 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:

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 don't specify a memory limit

If you don't specify a memory limit for a Container, then one of these 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.
- 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

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

[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. A Container is guaranteed to have as much CPU as it requests, but is not allowed to use more CPU than its limit.

- 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 don't 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 that the Heapster service is running in your cluster. But if you don't have Heapster running, you can do most of the steps, and it won't be a problem if you skip the Heapster steps.

If you are running minikube, run the following command to enable heapster:

```
minikube addons enable heapster
```

To see whether the Heapster service is running, enter this command:

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

If the heapster service is running, it shows in the output:

NAMESPACE	NAME	CLUSTER-IP	EXTERNAL-IP	PORT(S)	AGE
kube-system	heapster	10.11.240.9	<none>	80/TCP	6d

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 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's 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 CPU request of 0.5 cpu and a CPU limit of 1 cpu. Here's the configuration file for the Pod:

```
cpu-request-limit.yaml docs/tasks/configure-pod-container
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"
```

In the configuration file, the `args` section 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 create -f https://k8s.io/docs/tasks/configure-pod-container/cpu-request-limit.yaml
```

Verify that the Pod's Container 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
```

Start a proxy so that you can call the heapster service:

```
kubectl proxy
```

In another command window, get the CPU usage rate from the heapster service:

```
curl http://localhost:8001/api/v1/namespaces/kube-system/services/heapster/proxy/api/v1/mode
```

The 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's configuration file.

```
{  
  "timestamp": "2017-06-22T18:48:00Z",  
  "value": 974  
}
```

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: There's another possible explanation for the CPU throttling. The Node might not have enough CPU resources available. Recall that the prerequisites for this exercise require that each of your Nodes has at least 1 cpu. If your Container is running 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's 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.

```
cpu-request-limit-2.yaml docs/tasks/configure-pod-container
```

```
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 create -f https://k8s.io/docs/tasks/configure-pod-container/cpu-request-limit-2.yaml
```

View the Pod's status:

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

The output shows that the Pod's 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:: Insu

Delete your Pod:

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

If you don't specify a CPU limit

If you don't 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's Nodes. By keeping a Pod's 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

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

[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:

```
qos-pod.yaml docs/tasks/configure-pod-container
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 create -f https://k8s.io/docs/tasks/configure-pod-container/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's 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 mem-

ory 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.

```
qos-pod-2.yaml docs/tasks/configure-pod-container
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 create -f https://k8s.io/docs/tasks/configure-pod-container/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:

```
qos-pod-3.yaml docs/tasks/configure-pod-container
```

```
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 create -f https://k8s.io/docs/tasks/configure-pod-container/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.

```
qos-pod-4.yaml docs/tasks/configure-pod-container
```

```
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 create -f https://k8s.io/docs/tasks/configure-pod-container/qos-pod-4.yaml --namespa
```

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

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

[Edit This Page](#)

Assign Extended Resources to a Container

This page shows how to assign extended resources to a Container.

FEATURE STATE: Kubernetes v1.10 stable

This feature is *stable*, meaning:

- The version name is vX where X is an integer.
- Stable versions of features will appear in released software for many subsequent versions.
- Before you begin
- Assign an extended resource to a Pod
- Attempt to create a second 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`.

Before you do this exercise, do the exercise in Advertise Extended Resources for a Node. That will configure one of your Nodes to advertise a dongle resource.

Assign an extended resource to a Pod

To request an extended resource, include the `resources:requests` field in your Container manifest. Extended resources are fully qualified with any domain outside of `*.kubernetes.io/`. Valid extended resource names have the form

`example.com/foo` where `example.com` is replaced with your organization's domain and `foo` is a descriptive resource name.

Here is the configuration file for a Pod that has one Container:

```
extended-resource-pod.yaml docs/tasks/configure-pod-container
```

```
apiVersion: v1
kind: Pod
metadata:
  name: extended-resource-demo
spec:
  containers:
  - name: extended-resource-demo-ctr
    image: nginx
    resources:
      requests:
        example.com/dongle: 3
      limits:
        example.com/dongle: 3
```

In the configuration file, you can see that the Container requests 3 dongles.

Create a Pod:

```
kubectl create -f https://k8s.io/docs/tasks/configure-pod-container/extended-resource-pod.ya
```

Verify that the Pod is running:

```
kubectl get pod extended-resource-demo
```

Describe the Pod:

```
kubectl describe pod extended-resource-demo
```

The output shows dongle requests:

```
Limits:
  example.com/dongle: 3
Requests:
  example.com/dongle: 3
```

Attempt to create a second Pod

Here is the configuration file for a Pod that has one Container. The Container requests two dongles.

```
extended-resource-pod-2.yaml docs/tasks/configure-pod-container
```

```
apiVersion: v1
kind: Pod
metadata:
  name: extended-resource-demo-2
spec:
  containers:
  - name: extended-resource-demo-2-ctr
    image: nginx
    resources:
      requests:
        example.com/dongle: 2
      limits:
        example.com/dongle: 2
```

Kubernetes will not be able to satisfy the request for two dongles, because the first Pod used three of the four available dongles.

Attempt to create a Pod:

```
kubectl create -f https://k8s.io/docs/tasks/configure-pod-container/extended-resource-pod-2
```

Describe the Pod

```
kubectl describe pod extended-resource-demo-2
```

The output shows that the Pod cannot be scheduled, because there is no Node that has 2 dongles available:

Conditions:

Type	Status
PodScheduled	False

...

Events:

```
...
... Warning FailedScheduling pod (extended-resource-demo-2) failed to fit in any node
fit failure summary on nodes : Insufficient example.com/dongle (1)
```

View the Pod status:

```
kubectl get pod extended-resource-demo-2
```

The output shows that the Pod was created, but not scheduled to run on a Node. It has a status of Pending:

NAME	READY	STATUS	RESTARTS	AGE
------	-------	--------	----------	-----

```
extended-resource-demo-2  0/1        Pending    0          6m
```

Clean up

Delete the Pod that you created for this exercise:

```
kubectl delete pod extended-resource-demo-2
```

What's next

For application developers

- Assign Memory Resources to Containers and Pods
- Assign CPU Resources to Containers and Pods

For cluster administrators

- Advertise Extended Resources for a Node

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

[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, changes to the filesystem 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 and databases. For example, Redis is a key-value cache and store.

- 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:

```
pod-redis.yaml docs/tasks/configure-pod-container
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 create -f https://k8s.io/docs/tasks/configure-pod-container/pod-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 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# 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
```

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
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, goto `/data/redis`, and verify that `test-file` is still there.

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.

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

[Edit This Page](#)

Configure a Pod to Use a PersistentVolume for Storage

This page shows how to configure a Pod to use a PersistentVolumeClaim for storage. Here is a summary of the process:

1. A cluster administrator creates a PersistentVolume that is backed by physical storage. The administrator does not associate the volume with any Pod.
2. A cluster user creates a PersistentVolumeClaim, which gets automatically bound to a suitable PersistentVolume.
3. The user creates a Pod that uses the PersistentVolumeClaim as storage.
 - Before you begin
 - Create an index.html file on your Node
 - Create a PersistentVolume
 - Create a PersistentVolumeClaim
 - Create a Pod
 - 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 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 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, create a `/mnt/data` directory:

```
mkdir /mnt/data
```

In the `/mnt/data` directory, create an `index.html` file:

```
echo 'Hello from Kubernetes storage' > /mnt/data/index.html
```

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:

```
task-pv-volume.yaml docs/tasks/configure-pod-container
kind: PersistentVolume
apiVersion: v1
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 create -f https://k8s.io/docs/tasks/configure-pod-container/task-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
task-pv-volume	10Gi	RWO	Retain	Available		manual

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:

```
task-pv-claim.yaml docs/tasks/configure-pod-container
kind: PersistentVolumeClaim
apiVersion: v1
metadata:
  name: task-pv-claim
spec:
  storageClassName: manual
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: 3Gi
```

Create the PersistentVolumeClaim:

```
kubectl create -f https://k8s.io/docs/tasks/configure-pod-container/task-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	STATUS	CLAIM
task-pv-volume	10Gi	RWO	Retain	Bound	default/task-pv-claim

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	RWO	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:

```
task-pv-pod.yaml docs/tasks/configure-pod-container
kind: Pod
apiVersion: v1
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 create -f https://k8s.io/docs/tasks/configure-pod-container/task-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:

```
root@task-pv-pod:/# apt-get update  
root@task-pv-pod:/# apt-get install curl  
root@task-pv-pod:/# curl localhost
```

The output shows the text that you wrote to the `index.html` file on the hostPath volume:

```
Hello from Kubernetes storage
```

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:

```
kind: PersistentVolume  
apiVersion: v1  
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

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

[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`, and `downwardAPI` volumes can be projected.

- Before you begin
- Configure a projected 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 projected volume for a pod

In this exercise, you create username and password Secrets 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:

```
projected-volume.yaml docs/tasks/configure-pod-containerer
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 create -f https://k8s.io/docs/tasks/configure-pod-container/projected-volume
```

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		0	14s	Running

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/
```

What's next

- Learn more about `projected` volumes.
- Read the all-in-one volume design document.

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

[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
- 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:

```
security-context.yaml docs/tasks/configure-pod-container
apiVersion: v1
kind: Pod
metadata:
  name: security-context-demo
spec:
  securityContext:
    runAsUser: 1000
    fsGroup: 2000
  volumes:
  - name: sec-ctx-vol
    emptyDir: {}
  containers:
  - name: sec-ctx-demo
    image: gcr.io/google-samples/node-hello:1.0
    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, the first process runs with user ID 1000. The `fsGroup` field specifies that group ID 2000 is associated with all Containers in the Pod. Group ID 2000 is also associated with the volume mounted at `/data/demo` and with any files created in that volume.

Create the Pod:

```
kubectl create -f https://k8s.io/docs/tasks/configure-pod-container/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 aux
```

The output shows that the processes are running as user 1000, which is the value of `runAsUser`:

USER	PID	%CPU	%MEM	VSZ	RSS	TTY	STAT	START	TIME	COMMAND
------	-----	------	------	-----	-----	-----	------	-------	------	---------

```
1000      1  0.0  0.0   4336   724 ?      Ss  18:16  0:00 /bin/sh -c node server.js
1000      5  0.2  0.6 772124 22768 ?      S1  18:16  0:00 node server.js
...
```

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
```

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:

```
security-context-2.yaml docs/tasks/configure-pod-container
```

```
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 create -f https://k8s.io/docs/tasks/configure-pod-container/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:

```
security-context-3.yaml docs/tasks/configure-pod-container
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 create -f https://k8s.io/docs/tasks/configure-pod-container/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	?	S1	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:

```
security-context-4.yaml docs/tasks/configure-pod-container
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 create -f https://k8s.io/docs/tasks/configure-pod-container/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.

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

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

[Edit This Page](#)

Configure Service Accounts for Pods

A service account provides an identity for processes that run in a Pod.

This is a user introduction to Service Accounts. See also the Cluster Admin Guide to Service Accounts.

Note: This document 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

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 a service account has 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
NAME      SECRETS   AGE
default    1          1d
```

You can create additional ServiceAccount objects like this:

```
$ cat > /tmp/serviceaccount.yaml <<EOF
apiVersion: v1
kind: ServiceAccount
metadata:
  name: build-robot
EOF
$ kubectl create -f /tmp/serviceaccount.yaml
serviceaccount "build-robot" created
```

If you get a complete dump of the service account object, like this:

```
$ kubectl get serviceaccounts/build-robot -o yaml
apiVersion: v1
kind: ServiceAccount
metadata:
  creationTimestamp: 2015-06-16T00:12:59Z
  name: build-robot
  namespace: default
  resourceVersion: "272500"
  selfLink: /api/v1/namespaces/default/serviceaccounts/build-robot
  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.

```
$ cat > /tmp/build-robot-secret.yaml <<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
$ kubectl create -f /tmp/build-robot-secret.yaml
secret "build-robot-secret" created
```

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
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
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 requiring manual edit:

```
$ kubectl get serviceaccounts default -o yaml > ./sa.yaml
$ cat sa.yaml
apiVersion: v1
kind: ServiceAccount
metadata:
  creationTimestamp: 2015-08-07T22:02:39Z
  name: default
  namespace: default
  resourceVersion: "243024"
  selfLink: /api/v1/namespaces/default/serviceaccounts/default
  uid: 052fb0f4-3d50-11e5-b066-42010af0d7b6
secrets:
- name: default-token-uudge
$ vi sa.yaml
[editor session not shown]
[delete line with key "resourceVersion"]
[add lines with "imagePullSecrets:"]
$ cat sa.yaml
apiVersion: v1
kind: ServiceAccount
metadata:
  creationTimestamp: 2015-08-07T22:02:39Z
  name: default
  namespace: default
  selfLink: /api/v1/namespaces/default/serviceaccounts/default
  uid: 052fb0f4-3d50-11e5-b066-42010af0d7b6
secrets:
- name: default-token-uudge
imagePullSecrets:
- name: myregistrykey
$ kubectl replace serviceaccount default -f ./sa.yaml
serviceaccounts/default
```

Now, any new pods created in the current namespace will have this added to their spec:

```
spec:
  imagePullSecrets:
```

```
- name: myregistrykey
```

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

[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 in the cluster that holds your authorization token
- 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 in the cluster that holds your authorization token

A Kubernetes cluster uses the Secret of `docker-registry` type to authenticate with a container registry to pull a private image.

Create this Secret, naming it `regcred`:

```
kubectl create secret docker-registry regcred --docker-server=<your-registry-server> --docke
```

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 `regcred`.

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  
data:  
  .dockerconfigjson: eyJodHRwczovL2luZGV4L ... JOQU16RTIifX0=  
kind: Secret
```

```
metadata:  
  ...  
  name: regcred  
  ...  
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 -d
```

The output is similar to this:

```
{"auths":{"yourprivateregistry.com":{"username":"janedoe","password":"xxxxxxxxxxxx","email":}}
```

To understand what is in the `auth` field, convert the base64-encoded data to a readable format:

```
echo "c3R...zE2" | base64 -d
```

The output, username and password concatenated with a `:`, is similar to this:

```
janedoe:xxxxxxxxxx
```

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`:

```
private-reg-pod.yaml docs/tasks/configure-pod-container
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/docs/tasks/configure-pod-container/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:

```
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 create -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.
- See `kubectl create secret docker-registry`.
- See Secret.
- See the `imagePullSecrets` field of PodSpec.

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

[Edit This Page](#)

Configure Liveness and Readiness Probes

This page shows how to configure liveness and readiness 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.

- Before you begin
- Define a liveness command
- Define a liveness HTTP request
- Define a TCP liveness probe
- Use a named port
- 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:

```
exec-liveness.yaml docs/tasks/configure-pod-container
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 600
    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 create -f https://k8s.io/docs/tasks/configure-pod-container/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	LastSeen	Count	From	SubobjectPath	Type	Reason
24s	24s	1	{default-scheduler }		Normal	Scheduled Success
23s	23s	1	{kubelet worker0}	spec.containers{liveness}	Normal	Pulling
23s	23s	1	{kubelet worker0}	spec.containers{liveness}	Normal	Pulled
23s	23s	1	{kubelet worker0}	spec.containers{liveness}	Normal	Created
23s	23s	1	{kubelet worker0}	spec.containers{liveness}	Normal	Started

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	LastSeen	Count	From	SubobjectPath	Type	Reason
37s	37s	1	{default-scheduler }		Normal	Scheduled Success
36s	36s	1	{kubelet worker0}	spec.containers{liveness}	Normal	Pulling
36s	36s	1	{kubelet worker0}	spec.containers{liveness}	Normal	Pulled
36s	36s	1	{kubelet worker0}	spec.containers{liveness}	Normal	Created
36s	36s	1	{kubelet worker0}	spec.containers{liveness}	Normal	Started
2s	2s	1	{kubelet worker0}	spec.containers{liveness}	Warning	Unhealthy

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.

```
http-liveness.yaml docs/tasks/configure-pod-container
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: X-Custom-Header
          value: Awesome
    initialDelaySeconds: 3
    periodSeconds: 3
```

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 create -f https://k8s.io/docs/tasks/configure-pod-container/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
```

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.

```
tcp-liveness-readiness.yaml docs/tasks/configure-pod-container
```

```
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 create -f https://k8s.io/docs/tasks/configure-pod-container/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
```

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. 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.

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.
- **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 Pod. 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`.

What’s next

- Learn more about Container Probes.

Reference

- Pod
- Container

- Probe

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

[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
- 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:

```
kubectl get nodes
```

The output is similar to this:

NAME	STATUS	AGE	VERSION
worker0	Ready	1d	v1.6.0+fff5156
worker1	Ready	1d	v1.6.0+fff5156
worker2	Ready	1d	v1.6.0+fff5156

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	AGE	VERSION	LABELS
worker0	Ready	1d	v1.6.0+fff5156	...,disktype=ssd,kubernetes.io/hostname=
worker1	Ready	1d	v1.6.0+fff5156	...,kubernetes.io/hostname=worker1
worker2	Ready	1d	v1.6.0+fff5156	...,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.

`pod.yaml` [docs/tasks/configure-pod-container/pod.yaml](https://k8s.io/docs/tasks/configure-pod-container/pod.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 create -f https://k8s.io/docs/tasks/configure-pod-container/pod.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

What's next

Learn more about labels and selectors.

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

[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:

```
init-containers.yaml docs/tasks/configure-pod-container
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 create -f https://k8s.io/docs/tasks/configure-pod-container/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

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

[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:

```
lifecycle-events.yaml docs/tasks/configure-pod-container
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: ["/usr/sbin/nginx", "-s", "quit"]
```

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 create -f https://k8s.io/docs/tasks/configure-pod-container/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

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

[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 Pod environment variables using ConfigMap data
- Configure all key-value pairs in a ConfigMap as Pod 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

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.

The data source corresponds to a key-value pair in the ConfigMap, where

- key = the file name or the key you provided on the command line, and
- value = the file contents or the literal value you provided on the command line.

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.

For example:

```
mkdir -p configure-pod-container/configmap/kubectl/
wget https://k8s.io/docs/tasks/configure-pod-container/configmap/kubectl/game.properties -O game.properties
wget https://k8s.io/docs/tasks/configure-pod-container/configmap/kubectl/ui.properties -O ui.properties
kubectl create configmap game-config --from-file=configure-pod-container/configmap/kubectl/
```

combines the contents of the `configure-pod-container/configmap/kubectl/` directory

```
ls configure-pod-container/configmap/kubectl/
game.properties
ui.properties
```

into the following ConfigMap:

```
kubectl describe configmaps game-config
Name:          game-config
Namespace:     default
Labels:        <none>
Annotations:   <none>

Data
=====
game.properties:      158 bytes
ui.properties:        83 bytes
```

The `game.properties` and `ui.properties` files in the `configure-pod-container/configmap/kubectl/` directory are represented in the data section of the ConfigMap.

```
kubectl get configmaps game-config -o yaml
apiVersion: v1
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
kind: ConfigMap
metadata:
  creationTimestamp: 2016-02-18T18:52:05Z
  name: game-config
  namespace: default
  resourceVersion: "516"
  selfLink: /api/v1/namespaces/default/configmaps/game-config
  uid: b4952dc3-d670-11e5-8cd0-68f728db1985
```

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/kubectl
```

would produce the following ConfigMap:

```
kubectl describe configmaps game-config-2
Name:          game-config-2
Namespace:     default
Labels:        <none>
Annotations:   <none>

Data
=====
game.properties:      158 bytes
```

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/kubectl
kubectl describe configmaps game-config-2
Name:          game-config-2
Namespace:     default
Labels:        <none>
Annotations:   <none>

Data
=====
game.properties:      158 bytes
ui.properties:       83 bytes
```

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)

wget https://k8s.io/docs/tasks/configure-pod-container/configmap/kubectl/game-env-file.properties
cat configure-pod-container/configmap/kubectl/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/kubectl/game-env-file.properties
```

would produce the following ConfigMap:

```
kubectl get configmap game-config-env-file -o yaml
```

```
apiVersion: v1
data:
  allowed: '"true"'
  enemies: aliens
  lives: "3"
kind: ConfigMap
metadata:
  creationTimestamp: 2017-12-27T18:36:28Z
  name: game-config-env-file
  namespace: default
  resourceVersion: "809965"
  selfLink: /api/v1/namespaces/default/configmaps/game-config-env-file
  uid: d9d1ca5b-eb34-11e7-887b-42010a8002b8
```

When passing `--from-env-file` multiple times to create a ConfigMap from multiple data sources, only the last env-file is used:

```
wget https://k8s.io/docs/tasks/configure-pod-container/configmap/kubectl/ui-env-file.properties
kubectl create configmap config-multi-env-files \
--from-env-file=configure-pod-container/configmap/kubectl/game-env-file.properties \
--from-env-file=configure-pod-container/configmap/kubectl/ui-env-file.properties
```

would produce the following ConfigMap:

```
kubectl get configmap config-multi-env-files -o yaml
```

```
apiVersion: v1
data:
  color: purple
  how: fairlyNice
  textmode: "true"
kind: ConfigMap
metadata:
  creationTimestamp: 2017-12-27T18:38:34Z
  name: config-multi-env-files
  namespace: default
  resourceVersion: "810136"
  selfLink: /api/v1/namespaces/default/configmaps/config-multi-env-files
  uid: 252c4572-eb35-11e7-887b-42010a8002b8
```

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
```

```
kubectl get configmaps game-config-3 -o yaml
```

```
apiVersion: v1
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
kind: ConfigMap
metadata:
  creationTimestamp: 2016-02-18T18:54:22Z
  name: game-config-3
  namespace: default
  resourceVersion: "530"
  selfLink: /api/v1/namespaces/default/configmaps/game-config-3
  uid: 05f8da22-d671-11e5-8cd0-68f728db1985
```

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=speci
```

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
apiVersion: v1
data:
  special.how: very
```

```

    special.type: charm
kind: ConfigMap
metadata:
  creationTimestamp: 2016-02-18T19:14:38Z
  name: special-config
  namespace: default
  resourceVersion: "651"
  selfLink: /api/v1/namespaces/default/configmaps/special-config
  uid: dadce046-d673-11e5-8cd0-68f728db1985

```

Define Pod environment variables using ConfigMap data

Define a Pod 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=special.how=very
```

1. Assign the `special.how` value defined in the ConfigMap to the `SPECIAL_LEVEL_KEY` environment variable in the Pod specification.

```

kubectl edit pod dapi-test-pod

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

```

1. Save the changes to the Pod specification. Now, the Pod's output includes `SPECIAL_LEVEL_KEY=very`.

Define Pod environment variables with data from multiple ConfigMaps

1. As with the previous example, create the ConfigMaps first.

```
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
```

1. Define the environment variables in the Pod specification.

```
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
```

1. Save the changes to the Pod specification. Now, the Pod's output includes `SPECIAL_LEVEL_KEY=very` and `LOG_LEVEL=info`.

Configure all key-value pairs in a ConfigMap as Pod environment variables

Note: This functionality is available to users running Kubernetes v1.6 and later.

1. Create a ConfigMap containing multiple key-value pairs.

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: special-config
  namespace: default
data:
  SPECIAL_LEVEL: very
  SPECIAL_TYPE: charm
```

1. Use `envFrom` to define all of the ConfigMap's data as Pod environment variables. The key from the ConfigMap becomes the environment variable name in the Pod.

```
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
```

1. Save the changes to the Pod specification. Now, the Pod's output includes `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

```

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

```

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.

```

apiVersion: v1
kind: ConfigMap
metadata:
  name: special-config
  namespace: default
data:
  special.level: very
  special.type: charm

```

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 references the `special.level` item stored in the ConfigMap.

```
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
```

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`.

```
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

```

When the pod runs, the command ("cat /etc/config/keys") produces the output below:

very

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 + ttl of ConfigMaps cache in kubelet.

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`.

```
kind: ConfigMap
apiVersion: v1
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

1. 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.
2. 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
LASTSEEN FIRSTSEEN COUNT NAME KIND SUBOBJECT TYPE REASON
0s 0s 1 dapi-test-pod Pod Warning InvalidEnvironmentVaria
```

1. ConfigMaps reside in a specific namespace. A ConfigMap can only be referenced by pods residing in the same namespace.
2. Kubelet doesn't support the use of ConfigMaps for pods not found on the API server. This includes pods created via the Kubelet's `--manifest-url` flag, `--config` flag, or the Kubelet REST API.

Note: These are not commonly-used ways to create pods.

What's next

- Follow a real world example of Configuring Redis using a ConfigMap.

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

[Edit This Page](#)

Share Process Namespace between Containers in a Pod

FEATURE STATE: Kubernetes v1.10 alpha

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

- The version names contain alpha (e.g. v1alpha1).
- Might be buggy. Enabling the feature may expose bugs. Disabled by default.
- Support for feature may be dropped at any time without notice.
- The API may change in incompatible ways in a later software release without notice.
- Recommended for use only in short-lived testing clusters, due to increased risk of bugs and lack of long-term support.

This page shows how to configure process namespace sharing for a pod. When process namespace sharing is enabled, processes in a container are visible to all other containers in that pod.

You can use this feature to configure cooperating containers, such as a log handler sidecar container, or to troubleshoot container images that don't include debugging utilities like a shell.

- Before you begin
- Configure a Pod
- Understanding Process Namespace Sharing

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 version v1.10. To check the version, enter `kubectl version`.

A special **alpha** feature gate `PodShareProcessNamespace` must be set to true across the system: `--feature-gates=PodShareProcessNamespace=true`.

Configure a Pod

Process Namespace Sharing is enabled using the `ShareProcessNamespace` field of `v1.PodSpec`. For example:

```
share-process-namespace.yaml docs/tasks/configure-pod-container
```

```
apiVersion: v1
kind: Pod
metadata:
  name: nginx
spec:
  shareProcessNamespace: true
  containers:
    - name: nginx
      image: nginx
    - name: shell
      image: busybox
      securityContext:
        capabilities:
          add:
            - SYS_PTRACE
      stdin: true
      tty: true
```

1. Create the pod `nginx` on your cluster:

```
$ kubectl create -f https://k8s.io/docs/tasks/configure-pod-container/share-process-nam
```

2. Attach to the shell container and run ps:

```
$ kubectl attach -it nginx -c shell
If you don't see a command prompt, try pressing enter.
/ # ps ax
PID  USER      TIME  COMMAND
 1 root      0:00 /pause
  8 root      0:00 nginx: master process nginx -g daemon off;
 14 101      0:00 nginx: worker process
 15 root      0:00 sh
 21 root      0:00 ps ax
```

You can signal processes in other containers. For example, send SIGHUP to nginx to restart the worker process. This requires the SYS_PTRACE capability.

```
/ # kill -HUP 8
/ # ps ax
PID  USER      TIME  COMMAND
 1 root      0:00 /pause
  8 root      0:00 nginx: master process nginx -g daemon off;
 15 root      0:00 sh
 22 101      0:00 nginx: worker process
 23 root      0:00 ps ax
```

It's even possible to access another container image using the `/proc/$pid/root` link.

```
/ # head /proc/8/root/etc/nginx/nginx.conf

user    nginx;
worker_processes  1;

error_log  /var/log/nginx/error.log warn;
pid        /var/run/nginx.pid;

events {
    worker_connections  1024;
```

Understanding Process Namespace Sharing

Pods share many resources so it makes sense they would also share a process namespace. Some container images may expect to be isolated from other containers, though, so it's important to understand these differences:

1. **The container process no longer has PID 1.** Some container images refuse to start without PID 1 (for example, containers using `systemd`) or run commands like `kill -HUP 1` to signal the container process. In

pods with a shared process namespace, `kill -HUP 1` will signal the pod sandbox. (`/pause` in the above example.)

2. **Processes are visible to other containers in the pod.** This includes all information visible in `/proc`, such as passwords that were passed as arguments or environment variables. These are protected only by regular Unix permissions.
3. **Container filesystems are visible to other containers in the pod through the `/proc/$pid/root` link.** This makes debugging easier, but it also means that filesystem secrets are protected only by filesystem permissions.

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

[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 release

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/v1.1.0/kompose-linux-amd64 -o kompose

# macOS
curl -L https://github.com/kubernetes/kompose/releases/download/v1.1.0/kompose-darwin-amd64 -o kompose

# Windows
curl -L https://github.com/kubernetes/kompose/releases/download/v1.1.0/kompose-windows-amd64 -o kompose

chmod +x kompose
sudo mv ./kompose /usr/local/bin/kompose
```

Alternatively, you can download the tarball.

Go

Installing using `go get` pulls from the master branch with the latest development changes.

```
go get -u github.com/kubernetes/kompose
```

CentOS

Kompose is in EPEL CentOS repository. If you don't have EPEL repository already installed and enabled you can do it by running `sudo yum install epel-release`

If you have EPEL enabled in your system, you can install Kompose like any other package.

```
sudo yum -y install kompose
```

Fedora

Kompose is in Fedora 24, 25 and 26 repositories. You can install it just like any other package.

```
sudo dnf -y install kompose
```

macOS

On macOS you can install latest release via Homebrew:

```
brew install kompose
```

Use Kompose

In just a few steps, we'll take you from Docker Compose to Kubernetes. All you need is an existing `docker-compose.yml` file.

1. Go to the directory containing your `docker-compose.yml` file. If you don't have one, test using this one.

```
version: "2"

services:

  redis-master:
    image: k8s.gcr.io/redis:e2e
    ports:
      - "6379"

  redis-slave:
    image: gcr.io/google_samples/gb-redisslave:v1
    ports:
      - "6379"
    environment:
      - GET_HOSTS_FROM=dns

  frontend:
    image: gcr.io/google-samples/gb-frontend:v4
    ports:
      - "80:80"
    environment:
      - GET_HOSTS_FROM=dns
    labels:
      kompose.service.type: LoadBalancer
```

2. Run the `kompose up` command to deploy to Kubernetes directly, or skip to the next step instead to generate a file to use with `kubectl`.

```
$ kompose up
We are going to create Kubernetes Deployments, Services and PersistentVolumeClaims fo
If you need different kind of resources, use the 'kompose convert' and 'kubectl creat
```

```
INFO Successfully created Service: redis
INFO Successfully created Service: web
INFO Successfully created Deployment: redis
INFO Successfully created Deployment: web
```

Your application has been deployed to Kubernetes. You can run '`kubectl get deployment`' to see your services.

3. To convert the `docker-compose.yml` file to files that you can use with `kubectl`, run `kompose convert` and then `kubectl create -f <output file>`.

```
$ kompose convert
INFO Kubernetes file "frontend-service.yaml" created
INFO Kubernetes file "redis-master-service.yaml" created
INFO Kubernetes file "redis-slave-service.yaml" created
INFO Kubernetes file "frontend-deployment.yaml" created
INFO Kubernetes file "redis-master-deployment.yaml" created
INFO Kubernetes file "redis-slave-deployment.yaml" created

$ kubectl create -f frontend-service.yaml,redis-master-service.yaml,redis-slave-servi
service "frontend" created
service "redis-master" created
service "redis-slave" created
deployment "frontend" created
deployment "redis-master" created
deployment "redis-slave" created
```

Your deployments are running in Kubernetes.

4. Access your application.

If you're already using `minikube` for your development process:

```
$ minikube service frontend
```

Otherwise, let's look up what IP your service is using!

```
$ kubectl describe svc frontend
Name:           frontend
Namespace:      default
Labels:         service=frontend
Selector:       service=frontend
Type:          LoadBalancer
IP:            10.0.0.183
```

```
LoadBalancer Ingress: 123.45.67.89
Port:                 80      80/TCP
NodePort:              80      31144/TCP
Endpoints:            172.17.0.4:80
Session Affinity:    None
No events.
```

If you're using a cloud provider, your IP will be listed next to `LoadBalancer Ingress`.

```
$ curl http://123.45.67.89
```

User Guide

- CLI
 - `kompose convert`
 - `kompose up`
 - `kompose down`
- Documentation
 - Build and Push Docker Images
 - Alternative Conversions
 - Labels
 - Restart
 - Docker Compose Versions

Kompose has support for two providers: OpenShift and Kubernetes. You can choose a targeted provider using global option `--provider`. If no provider is specified, Kubernetes is set by default.

kompose convert

Kompose supports conversion of V1, V2, and V3 Docker Compose files into Kubernetes and OpenShift objects.

Kubernetes

```
$ kompose --file docker-voting.yml convert
WARN Unsupported key networks - ignoring
WARN Unsupported key build - ignoring
INFO Kubernetes file "worker-svc.yaml" created
INFO Kubernetes file "db-svc.yaml" created
INFO Kubernetes file "redis-svc.yaml" created
INFO Kubernetes file "result-svc.yaml" created
INFO Kubernetes file "vote-svc.yaml" created
INFO Kubernetes file "redis-deployment.yaml" created
```

```

INFO Kubernetes file "result-deployment.yaml" created
INFO Kubernetes file "vote-deployment.yaml" created
INFO Kubernetes file "worker-deployment.yaml" created
INFO Kubernetes file "db-deployment.yaml" created

$ ls
db-deployment.yaml  docker-compose.yml          docker-gitlab.yml  redis-deployment.yaml res
db-svc.yaml         docker-voting.yml        redis-svc.yaml    result-svc.yaml
vot

```

You can also provide multiple docker-compose files at the same time:

```

$ kompose -f docker-compose.yml -f docker-guestbook.yml convert
INFO Kubernetes file "frontend-service.yaml" created
INFO Kubernetes file "mlbparks-service.yaml" created
INFO Kubernetes file "mongodb-service.yaml" created
INFO Kubernetes file "redis-master-service.yaml" created
INFO Kubernetes file "redis-slave-service.yaml" created
INFO Kubernetes file "frontend-deployment.yaml" created
INFO Kubernetes file "mlbparks-deployment.yaml" created
INFO Kubernetes file "mongodb-deployment.yaml" created
INFO Kubernetes file "mongodb-claim0-persistentvolumeclaim.yaml" created
INFO Kubernetes file "redis-master-deployment.yaml" created
INFO Kubernetes file "redis-slave-deployment.yaml" created

$ ls
mlbparks-deployment.yaml  mongodb-service.yaml          redis-slave-service.json
frontend-deployment.yaml  mongodb-claim0-persistentvolumeclaim.yaml  redis-master-service.yaml
frontend-service.yaml      mongodb-deployment.yaml      redis-slave-deployment.yaml
redis-master-deployment.yaml

```

When multiple docker-compose files are provided the configuration is merged.
Any configuration that is common will be over ridden by subsequent file.

OpenShift

```

$ kompose --provider openshift --file docker-voting.yml convert
WARN [worker] Service cannot be created because of missing port.
INFO OpenShift file "vote-service.yaml" created
INFO OpenShift file "db-service.yaml" created
INFO OpenShift file "redis-service.yaml" created
INFO OpenShift file "result-service.yaml" created
INFO OpenShift file "vote-deploymentconfig.yaml" created
INFO OpenShift file "vote-imagestream.yaml" created
INFO OpenShift file "worker-deploymentconfig.yaml" created
INFO OpenShift file "worker-imagestream.yaml" created
INFO OpenShift file "db-deploymentconfig.yaml" created

```

```
INFO OpenShift file "db-imagestream.yaml" created
INFO OpenShift file "redis-deploymentconfig.yaml" created
INFO OpenShift file "redis-imagestream.yaml" created
INFO OpenShift file "result-deploymentconfig.yaml" created
INFO OpenShift file "result-imagestream.yaml" created
```

It also supports creating buildconfig for build directive in a service. By default, it uses the remote repo for the current git branch as the source repo, and the current branch as the source branch for the build. You can specify a different source repo and branch using `--build-repo` and `--build-branch` options respectively.

```
$ kompose --provider openshift --file buildconfig/docker-compose.yml convert
WARN [foo] Service cannot be created because of missing port.
INFO OpenShift Buildconfig using git@github.com:rtnpro/kompose.git::master as source.
INFO OpenShift file "foo-deploymentconfig.yaml" created
INFO OpenShift file "foo-imagestream.yaml" created
INFO OpenShift file "foo-buildconfig.yaml" created
```

Note: If you are manually pushing the Openshift artifacts using `oc create -f`, you need to ensure that you push the imagestream artifact before the buildconfig artifact, to workaround this Openshift issue: <https://github.com/openshift/origin/issues/4518> .

kompose up

Kompose supports a straightforward way to deploy your “composed” application to Kubernetes or OpenShift via `kompose up`.

Kubernetes

```
$ kompose --file ./examples/docker-guestbook.yml up
We are going to create Kubernetes deployments and services for your Dockerized application.
If you need different kind of resources, use the 'kompose convert' and 'kubectl create -f' commands.

INFO Successfully created service: redis-master
INFO Successfully created service: redis-slave
INFO Successfully created service: frontend
INFO Successfully created deployment: redis-master
INFO Successfully created deployment: redis-slave
INFO Successfully created deployment: frontend
```

Your application has been deployed to Kubernetes. You can run 'kubectl get deployment,svc,pods'

```
$ kubectl get deployment,svc,pods
```

NAME	DESIRED	CURRENT	UP-TO-DATE	AVAILABLE	AGE
deploy/frontend	1	1	1	1	4m
deploy/redis-master	1	1	1	1	4m
deploy/redis-slave	1	1	1	1	4m
NAME	CLUSTER-IP	EXTERNAL-IP	PORT(S)	AGE	
svc/frontend	10.0.174.12	<none>	80/TCP	4m	
svc/kubernetes	10.0.0.1	<none>	443/TCP	13d	
svc/redis-master	10.0.202.43	<none>	6379/TCP	4m	
svc/redis-slave	10.0.1.85	<none>	6379/TCP	4m	
NAME	READY	STATUS	RESTARTS	AGE	
po/frontend-2768218532-cs5t5	1/1	Running	0	4m	
po/redis-master-1432129712-63jn8	1/1	Running	0	4m	
po/redis-slave-2504961300-nve7b	1/1	Running	0	4m	

Note: - You must have a running Kubernetes cluster with a pre-configured kubectl context.
 - Only deployments and services are generated and deployed to Kubernetes. If you need different kind of resources, use the 'kompose convert' and 'kubectl create -f' commands instead.

OpenShift

```
$ kompose --file ./examples/docker-guestbook.yml --provider openshift up
We are going to create OpenShift DeploymentConfigs and Services for your Dockerized application
If you need different kind of resources, use the 'kompose convert' and 'oc create -f' command
```

```
INFO Successfully created service: redis-slave
INFO Successfully created service: frontend
INFO Successfully created service: redis-master
INFO Successfully created deployment: redis-slave
INFO Successfully created ImageStream: redis-slave
INFO Successfully created deployment: frontend
INFO Successfully created ImageStream: frontend
INFO Successfully created deployment: redis-master
INFO Successfully created ImageStream: redis-master
```

Your application has been deployed to OpenShift. You can run 'oc get dc,svc,is' for details.

```
$ oc get dc,svc,is
NAME          REVISION           DESIRED   CURRENT   TRIGGERED
dc/frontend   0                  1         0         config,image
dc/redis-master 0                1         0         config,image
dc/redis-slave 0                1         0         config,image
NAME          CLUSTER-IP        EXTERNAL-IP PORT(S)  AGE
```

svc/frontend	172.30.46.64	<none>	80/TCP	8s
svc/redis-master	172.30.144.56	<none>	6379/TCP	8s
svc/redis-slave	172.30.75.245	<none>	6379/TCP	8s
NAME	DOCKER REPO	TAGS		UPDATED
is/frontend	172.30.12.200:5000/fff/frontend			
is/redis-master	172.30.12.200:5000/fff/redis-master			
is/redis-slave	172.30.12.200:5000/fff/redis-slave	v1		

Note: - You must have a running OpenShift cluster with a pre-configured oc context (`oc login`)

kompose down

Once you have deployed “composed” application to Kubernetes, `$ kompose down` will help you to take the application out by deleting its deployments and services. If you need to remove other resources, use the ‘kubectl’ command.

```
$ kompose --file docker-guestbook.yml down
INFO Successfully deleted service: redis-master
INFO Successfully deleted deployment: redis-master
INFO Successfully deleted service: redis-slave
INFO Successfully deleted deployment: redis-slave
INFO Successfully deleted service: frontend
INFO Successfully deleted deployment: frontend
```

Note: - You must have a running Kubernetes cluster with a pre-configured kubectl context.

Build and Push Docker Images

Kompose supports both building and pushing Docker images. When using the `build` key within your Docker Compose file, your image will:

- Automatically be built with Docker using the `image` key specified within your file
- Be pushed to the correct Docker repository using local credentials (located at `.docker/config`)

Using an example Docker Compose file:

```
version: "2"

services:
  foo:
    build: "./build"
    image: docker.io/foo/bar
```

Using kompose up with a build key:

```
$ kompose up
INFO Build key detected. Attempting to build and push image 'docker.io/foo/bar'
INFO Building image 'docker.io/foo/bar' from directory 'build'
INFO Image 'docker.io/foo/bar' from directory 'build' built successfully
INFO Pushing image 'foo/bar:latest' to registry 'docker.io'
INFO Attempting authentication credentials 'https://index.docker.io/v1/'
INFO Successfully pushed image 'foo/bar:latest' to registry 'docker.io'
INFO We are going to create Kubernetes Deployments, Services and PersistentVolumeClaims for

INFO Deploying application in "default" namespace
INFO Successfully created Service: foo
INFO Successfully created Deployment: foo
```

Your application has been deployed to Kubernetes. You can run 'kubectl get deployment,svc,po'

In order to disable the functionality, or choose to use BuildConfig generation
(with OpenShift) --build (local|build-config|none) can be passed.

```
# Disable building/pushing Docker images
$ kompose up --build none

# Generate Build Config artifacts for OpenShift
$ kompose up --provider openshift --build build-config
```

Alternative Conversions

The default `kompose` transformation will generate Kubernetes Deployments and Services, in yaml format. You have alternative option to generate json with `-j`. Also, you can alternatively generate Replication Controllers objects, Daemon Sets, or Helm charts.

```
$ kompose convert -j
INFO Kubernetes file "redis-svc.json" created
INFO Kubernetes file "web-svc.json" created
INFO Kubernetes file "redis-deployment.json" created
INFO Kubernetes file "web-deployment.json" created
```

The `*-deployment.json` files contain the Deployment objects.

```
$ kompose convert --replication-controller
INFO Kubernetes file "redis-svc.yaml" created
INFO Kubernetes file "web-svc.yaml" created
INFO Kubernetes file "redis-replicationcontroller.yaml" created
INFO Kubernetes file "web-replicationcontroller.yaml" created
```

The `*-replicationcontroller.yaml` files contain the Replication Controller objects. If you want to specify replicas (default is 1), use `--replicas` flag:

```
$ kompose convert --replication-controller --replicas 3
```

```
$ kompose convert --daemon-set
INFO Kubernetes file "redis-svc.yaml" created
INFO Kubernetes file "web-svc.yaml" created
INFO Kubernetes file "redis-daemonset.yaml" created
INFO Kubernetes file "web-daemonset.yaml" created
```

The `*-daemonset.yaml` files contain the Daemon Set objects

If you want to generate a Chart to be used with Helm simply do:

```
$ kompose convert -c
INFO Kubernetes file "web-svc.yaml" created
INFO Kubernetes file "redis-svc.yaml" created
INFO Kubernetes file "web-deployment.yaml" created
INFO Kubernetes file "redis-deployment.yaml" created
chart created in "./docker-compose/"

$ tree docker-compose/
docker-compose
  Chart.yaml
  README.md
  templates
    redis-deployment.yaml
    redis-svc.yaml
    web-deployment.yaml
    web-svc.yaml
```

The chart structure is aimed at providing a skeleton for building your Helm charts.

Labels

`kompose` supports Kompose-specific labels within the `docker-compose.yml` file in order to explicitly define a service's behavior upon conversion.

- `kompose.service.type` defines the type of service to be created.

For example:

```
version: "2"
services:
  nginx:
    image: nginx
    dockerfile: foobar
    build: ./foobar
```

```

cap_add:
  - ALL
container_name: foobar
labels:
  kompose.service.type: nodeport

```

- kompose.service.expose defines if the service needs to be made accessible from outside the cluster or not. If the value is set to “true”, the provider sets the endpoint automatically, and for any other value, the value is set as the hostname. If multiple ports are defined in a service, the first one is chosen to be the exposed.
 - For the Kubernetes provider, an ingress resource is created and it is assumed that an ingress controller has already been configured.
 - For the OpenShift provider, a route is created.

For example:

```

version: "2"
services:
  web:
    image: tuna/docker-counter23
    ports:
      - "5000:5000"
    links:
      - redis
    labels:
      kompose.service.expose: "counter.example.com"
  redis:
    image: redis:3.0
    ports:
      - "6379"

```

The currently supported options are:

Key	Value
kompose.service.type	nodeport / clusterip / loadbalancer
kompose.service.expose	true / hostname

Note: kompose.service.type label should be defined with ports only, otherwise kompose will fail.

Restart

If you want to create normal pods without controllers you can use `restart` construct of docker-compose to define that. Follow table below to see what

happens on the `restart` value.

<code>docker-compose restart</code>	object created	Pod <code>restartPolicy</code>
""	controller object	Always
always	controller object	Always
on-failure	Pod	OnFailure
no	Pod	Never

Note: controller object could be `deployment` or `replicationcontroller`, etc.

For e.g. `pival` service will become pod down here. This container calculated value of `pi`.

```
version: '2'

services:
  pival:
    image: perl
    command: ["perl", "-Mbignum=bpi", "-wle", "print bpi(2000)"]
    restart: "on-failure"
```

Warning about Deployment Config's

If the Docker Compose file has a volume specified for a service, the Deployment (Kubernetes) or DeploymentConfig (OpenShift) strategy is changed to “Recreate” instead of “RollingUpdate” (default). This is done to avoid multiple instances of a service from accessing a volume at the same time.

If the Docker Compose file has service name with `_` in it (eg.`web_service`), then it will be replaced by `-` and the service name will be renamed accordingly (eg.`web-service`). Kompose does this because “Kubernetes” doesn’t allow `_` in object name.

Please note that changing service name might break some `docker-compose` files.

Docker Compose Versions

Kompose supports Docker Compose versions: 1, 2 and 3. We have limited support on versions 2.1 and 3.2 due to their experimental nature.

A full list on compatibility between all three versions is listed in our conversion document including a list of all incompatible Docker Compose keys.

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

[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:

```
cpu-constraints.yaml
docs/tasks/administer-cluster/manage-resources
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 create -f https://k8s.io/docs/tasks/administer-cluster/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`.

`cpu-constraints-pod.yaml`

`docs/tasks/administer-cluster/manage-resources`

```
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 create -f https://k8s.io/docs/tasks/administer-cluster/cpu-constraints-pod.yaml --name=constraints-cpu-demo
```

Verify that the Pod's Container is running:

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

View detailed information about the Pod:

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

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

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

Delete the Pod

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

Attempt to create a Pod that exceeds the maximum CPU constraint

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

```
cpu-constraints-pod-2.yaml
docs/tasks/administer-cluster/manage-resources
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 create -f https://k8s.io/docs/tasks/administer-cluster/cpu-constraints-pod-2.yaml --
```

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 "docs/tasks/administer-cluster/cpu-constraints-pod-2.yaml": pods "constraints-cpu-demo-2" is forbidden: maximum cpu usage per Container is 800m, but limit is 1.5
```

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.

```
cpu-constraints-pod-3.yaml
```

```
docs/tasks/administer-cluster/manage-resources
apiVersion: v1
kind: Pod
metadata:
  name: constraints-cpu-demo-4
spec:
  containers:
  - name: constraints-cpu-demo-4-ctr
    image: nginx
    resources:
      limits:
        cpu: "800m"
      requests:
        cpu: "100m"
```

Attempt to create the Pod:

```
kubectl create -f https://k8s.io/docs/tasks/administer-cluster/cpu-constraints-pod-3.yaml --
```

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 "docs/tasks/administer-cluster/cpu-constraints-pod-3.yaml": pods "constraints-cpu-demo-4" is forbidden: minimum cpu usage per Container is 200m, but requested 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.

```
cpu-constraints-pod-4.yaml
```

```
docs/tasks/administer-cluster/manage-resources
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 create -f https://k8s.io/docs/tasks/administer-cluster/cpu-constraints-pod-4.yaml --
```

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

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

[Edit This Page](#)

Upgrading/downgrading kubeadm clusters between v1.8 to v1.9

This guide is for upgrading `kubeadm` clusters from version 1.8.x to 1.9.x, as well as 1.8.x to 1.8.y and 1.9.x to 1.9.y where $y > x$. See also upgrading `kubeadm` clusters from 1.7 to 1.8 if you're on a 1.7 cluster currently.

- Before you begin
- Upgrading your control plane
- Upgrading your master and node packages
- Recovering from a failure state

Before you begin

Before proceeding:

- You need to have a functional `kubeadm` Kubernetes cluster running version 1.8.0 or higher in order to use the process described here. Swap also needs to be disabled.
- Make sure you read the release notes carefully.
- `kubeadm upgrade` now allows you to upgrade etcd. `kubeadm upgrade` will also upgrade of etcd to 3.1.10 as part of upgrading from v1.8 to v1.9 by default. This is due to the fact that etcd 3.1.10 is the officially validated etcd version for Kubernetes v1.9. The upgrade is handled automatically by `kubeadm` for you.
- Note that `kubeadm upgrade` will not touch any of your workloads, only Kubernetes-internal components. As a best-practice you should back up what's important to you. For example, any app-level state, such as a database an app might depend on (like MySQL or MongoDB) must be backed up beforehand.

Caution: All the containers will get restarted after the upgrade, due to container spec hash value gets changed.

Also, note that only one minor version upgrade is supported. For example, you can only upgrade from 1.8 to 1.9, not from 1.7 to 1.9.

Upgrading your control plane

Execute these commands on your master node:

1. Install the most recent version of `kubeadm` using `curl` like so:

```
export VERSION=$(curl -sSL https://dl.k8s.io/release/stable.txt) # or manually specify a release
export ARCH=amd64 # or: arm, arm64, ppc64le, s390x
curl -sSL https://dl.k8s.io/release/${VERSION}/bin/linux/${ARCH}/kubeadm > /usr/bin/kubeadm
chmod a+r /usr/bin/kubeadm
```

Caution: Upgrading the `kubeadm` package on your system prior to upgrading the control plane causes a failed upgrade. Even though `kubeadm` ships in the Kubernetes repositories, it's important to install `kubeadm` manually. The `kubeadm` team is working on fixing this limitation.

Verify that this download of `kubeadm` works and has the expected version:

```
kubeadm version
```

1. On the master node, run the following:

```
kubeadm upgrade plan
```

You should see output similar to this:

```
[preflight] Running pre-flight checks
[upgrade] Making sure the cluster is healthy:
[upgrade/health] Checking API Server health: Healthy
[upgrade/health] Checking Node health: All Nodes are healthy
[upgrade/health] Checking Static Pod manifests exists on disk: All manifests exist on disk
[upgrade/config] Making sure the configuration is correct:
[upgrade/config] Reading configuration from the cluster...
[upgrade/config] FYI: You can look at this config file with 'kubectl -n kube-system get cm kubeadm-config -o yaml'
[upgrade] Fetching available versions to upgrade to:
[upgrade/versions] Cluster version: v1.8.1
[upgrade/versions] kubeadm version: v1.9.0
[upgrade/versions] Latest stable version: v1.9.0
[upgrade/versions] Latest version in the v1.8 series: v1.8.6
```

Components that must be upgraded manually after you've upgraded the control plane with 'kubeadm upgrade'

COMPONENT	CURRENT	AVAILABLE
Kubelet	1 x v1.8.1	v1.8.6

Upgrade to the latest version in the v1.8 series:

COMPONENT	CURRENT	AVAILABLE
API Server	v1.8.1	v1.8.6
Controller Manager	v1.8.1	v1.8.6

Scheduler	v1.8.1	v1.8.6
Kube Proxy	v1.8.1	v1.8.6
Kube DNS	1.14.4	1.14.5

You can now apply the upgrade by executing the following command:

```
kubeadm upgrade apply v1.8.6
```

Components that must be upgraded manually after you've upgraded the control plane with 'kubeadm upgrade plan':

COMPONENT	CURRENT	AVAILABLE
Kubelet	1 x v1.8.1	v1.9.0

Upgrade to the latest stable version:

COMPONENT	CURRENT	AVAILABLE
API Server	v1.8.1	v1.9.0
Controller Manager	v1.8.1	v1.9.0
Scheduler	v1.8.1	v1.9.0
Kube Proxy	v1.8.1	v1.9.0
Kube DNS	1.14.5	1.14.7

You can now apply the upgrade by executing the following command:

```
kubeadm upgrade apply v1.9.0
```

Note: Before you do can perform this upgrade, you have to update kubeadm to v1.9.0

The `kubeadm upgrade plan` checks that your cluster is upgradeable and fetches the versions available to upgrade to in an user-friendly way.

To check CoreDNS version, include the `--feature-gates=CoreDNS=true` flag to verify the CoreDNS version which will be installed in place of kube-dns.

1. Pick a version to upgrade to and run. For example:

```
kubeadm upgrade apply v1.9.0
```

You should see output similar to this:

```
[preflight] Running pre-flight checks.
[upgrade] Making sure the cluster is healthy:
[upgrade/config] Making sure the configuration is correct:
[upgrade/config] Reading configuration from the cluster...
[upgrade/config] FYI: You can look at this config file with 'kubectl -n kube-system get cm k
[upgrade/version] You have chosen to upgrade to version "v1.9.0"
```

```

[upgrade/versions] Cluster version: v1.8.1
[upgrade/versions] kubeadm version: v1.9.0
[upgrade/confirm] Are you sure you want to proceed with the upgrade? [y/N]: y
[upgrade/prepull] Will prepull images for components [kube-apiserver kube-controller-manager]
[upgrade/apply] Upgrading your Static Pod-hosted control plane to version "v1.9.0"...
[etcd] Wrote Static Pod manifest for a local etcd instance to "/etc/kubernetes/tmp/kubeadm-...
[upgrade/staticpods] Moved upgraded manifest to "/etc/kubernetes/manifests/etc.yaml" and ba...
[upgrade/staticpods] Waiting for the kubelet to restart the component
[apiclient] Found 1 Pods for label selector component=etcd
[upgrade/staticpods] Component "etcd" upgraded successfully!
[upgrade/staticpods] Writing upgraded Static Pod manifests to "/etc/kubernetes/tmp/kubeadm-...
[controlplane] Wrote Static Pod manifest for component kube-apiserver to "/etc/kubernetes/...
[controlplane] Wrote Static Pod manifest for component kube-controller-manager to "/etc/kube...
[controlplane] Wrote Static Pod manifest for component kube-scheduler to "/etc/kubernetes/tr...
[upgrade/staticpods] Moved upgraded manifest to "/etc/kubernetes/manifests/kube-apiserver.ya...
[upgrade/staticpods] Waiting for the kubelet to restart the component
[apiclient] Found 1 Pods for label selector component=kube-apiserver
[upgrade/staticpods] Component "kube-apiserver" upgraded successfully!
[upgrade/staticpods] Moved upgraded manifest to "/etc/kubernetes/manifests/kube-controller-m...
[upgrade/staticpods] Waiting for the kubelet to restart the component
[apiclient] Found 1 Pods for label selector component=kube-controller-manager
[upgrade/staticpods] Component "kube-controller-manager" upgraded successfully!
[upgrade/staticpods] Moved upgraded manifest to "/etc/kubernetes/manifests/kube-scheduler.ya...
[upgrade/staticpods] Waiting for the kubelet to restart the component
[apiclient] Found 1 Pods for label selector component=kube-scheduler
[upgrade/staticpods] Component "kube-scheduler" upgraded successfully!
[uploadconfig] Storing the configuration used in ConfigMap "kubeadm-config" in the "kube-sys...
[bootstraptoken] Configured RBAC rules to allow Node Bootstrap tokens to post CSRs in order ...
[bootstraptoken] Configured RBAC rules to allow the csapprover controller automatically app...
[bootstraptoken] Configured RBAC rules to allow certificate rotation for all node client ce...
[addons] Applied essential addon: kube-dns
[addons] Applied essential addon: kube-proxy

[upgrade/successful] SUCCESS! Your cluster was upgraded to "v1.9.0". Enjoy!

[upgrade/kubelet] Now that your control plane is upgraded, please proceed with upgrading yo...
To upgrade the cluster with CoreDNS as the default internal DNS, invoke
kubeadm upgrade apply with the --feature-gates=CoreDNS=true flag.
kubeadm upgrade apply does the following:


- Checks that your cluster is in an upgradeable state:
  - The API server is reachable,
  - All nodes are in the Ready state
  - The control plane is healthy
- Enforces the version skew policies.
- Makes sure the control plane images are available or available to pull to

```

the machine.

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

1. Manually upgrade your Software Defined Network (SDN).

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

Upgrading your master and node packages

For each host (referred to as `$HOST` below) in your cluster, upgrade `kubelet` by executing the following commands:

1. Prepare the host for maintenance, marking it unschedulable and evicting the workload:

```
kubectl drain $HOST --ignore-daemonsets
```

When running this command against the master host, this error is expected and can be safely ignored (since there are static pods running on the master):

```
node "master" already cordoned
error: pods not managed by ReplicationController, ReplicaSet, Job, DaemonSet or StatefulSet
```

1. Upgrade the Kubernetes package versions on the `$HOST` node by using a Linux distribution-specific package manager:

If the host is running a Debian-based distro such as Ubuntu, run:

```
apt-get update
apt-get upgrade
```

If the host is running CentOS or the like, run:

```
yum update
```

Now the new version of the `kubelet` should be running on the host. Verify this using the following command on `$HOST`:

```
systemctl status kubelet
```

1. Bring the host back online by marking it schedulable:

```
kubectl uncordon $HOST
```

1. After upgrading `kubelet` on each host in your cluster, verify that all nodes are available again by executing the following (from anywhere, for example, from outside the cluster):

```
kubectl get nodes
```

If the `STATUS` column of the above command shows `Ready` for all of your hosts, you are done.

Recovering from a failure state

If `kubeadm upgrade` somehow fails and fails to roll back, for example due to an unexpected shutdown during execution, you can run `kubeadm upgrade` again as it is idempotent and should eventually make sure the actual state is the desired state you are declaring.

You can use `kubeadm upgrade` to change a running cluster with `x.x.x --> x.x.x` with `--force`, which can be used to recover from a bad state.

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

[Edit This Page](#)

Upgrading kubeadm HA clusters from 1.9.x to 1.9.y

This guide is for upgrading kubeadm HA clusters from version 1.9.x to 1.9.y where $y > x$. The term “kubeadm HA clusters” refers to clusters of more than one master node created with kubeadm. To set up an HA cluster for Kubernetes version 1.9.x kubeadm requires additional manual steps. See Creating HA clusters with kubeadm for instructions on how to do this. The upgrade procedure described here targets clusters created following those very instructions. See Upgrading/downgrading kubeadm clusters between v1.8 to v1.9 for more instructions on how to create an HA cluster with kubeadm.

- Before you begin
- Preparation
- Upgrading your control plane
- Upgrade base software packages
- If something goes wrong

Before you begin

Before proceeding:

- You need to have a functional `kubeadm` HA cluster running version 1.9.0 or higher in order to use the process described here.
- Make sure you read the release notes carefully.
- Note that `kubeadm upgrade` will not touch any of your workloads, only Kubernetes-internal components. As a best-practice you should back up anything important to you. For example, any application-level state, such as a database and application might depend on (like MySQL or MongoDB) should be backed up beforehand.
- Read Upgrading/downgrading kubeadm clusters between v1.8 to v1.9 to learn about the relevant prerequisites.

Preparation

Some preparation is needed prior to starting the upgrade. First download the version of `kubeadm` that matches the version of Kubernetes that you are upgrading to:

```
# Use the latest stable release or manually specify a
# released Kubernetes version
export VERSION=$(curl -sSL https://dl.k8s.io/release/stable.txt)
export ARCH=amd64 # or: arm, arm64, ppc64le, s390x
curl -sSL https://dl.k8s.io/release/${VERSION}/bin/linux/${ARCH}/kubeadm > /tmp/kubeadm
chmod a+r /tmp/kubeadm
```

Copy this file to `/tmp` on your primary master if necessary. Run this command for checking prerequisites and determining the versions you will receive:

```
/tmp/kubeadm upgrade plan
```

If the prerequisites are met you'll get a summary of the software versions kubeadm will upgrade to, like this:

`Upgrade to the latest stable version:`

COMPONENT	CURRENT	AVAILABLE
API Server	v1.9.0	v1.9.2
Controller Manager	v1.9.0	v1.9.2
Scheduler	v1.9.0	v1.9.2
Kube Proxy	v1.9.0	v1.9.2
Kube DNS	1.14.5	1.14.7
Etcd	3.2.7	3.1.11

Caution: Currently the only supported configuration for kubeadm HA clusters requires the use of an externally managed etcd cluster.

Upgrading etcd is not supported as a part of the upgrade. If necessary you will have to upgrade the etcd cluster according to etcd's upgrade instructions, which is beyond the scope of these instructions.

Upgrading your control plane

The following procedure must be applied on a single master node and repeated for each subsequent master node sequentially.

Before initiating the upgrade with `kubeadm configmap/kubeadm-config` needs to be modified for the current master host. Replace any hard reference to a master host name with the current master hosts' name:

```
kubectl get configmap -n kube-system kubeadm-config -o yaml >/tmp/kubeadm-config-cm.yaml  
sed -i 's/^\([ \t]*nodeName:\).*/\1 <CURRENT-MASTER-NAME>/' /tmp/kubeadm-config-cm.yaml  
kubectl apply -f /tmp/kubeadm-config-cm.yaml --force
```

Now the upgrade process can start. Use the target version determined in the preparation step and run the following command (press "y" when prompted):

```
/tmp/kubeadm upgrade apply v<YOUR-CHOSEN-VERSION-HERE>
```

If the operation was successful you'll get a message like this:

```
[upgrade/successful] SUCCESS! Your cluster was upgraded to "v1.9.2". Enjoy!
```

To upgrade the cluster with CoreDNS as the default internal DNS, invoke `kubeadm upgrade apply` with the `--feature-gates=CoreDNS=true` flag.

Next, manually upgrade your CNI provider

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

Note: The `kubeadm upgrade apply` step has been known to fail when run initially on the secondary masters (timed out waiting for the restarted static pods to come up). It should succeed if retried after a minute or two.

Upgrade base software packages

At this point all the static pod manifests in your cluster, for example API Server, Controller Manager, Scheduler, Kube Proxy have been upgraded, however the base software, for example `kubelet`, `kubectl`, `kubeadm` installed on your nodes' OS are still of the old version. For upgrading the base software packages we will upgrade them and restart services on all nodes one by one:

```
# use your distro's package manager, e.g. 'yum' on RH-based systems
# for the versions stick to kubeadm's output (see above)
yum install -y kubelet-<NEW-K8S-VERSION> kubectl-<NEW-K8S-VERSION> kubeadm-<NEW-K8S-VERSION>
systemctl restart kubelet
```

In this example an *rpm*-based system is assumed and `yum` is used for installing the upgraded software. On *deb*-based systems it will be `apt-get update` and then `apt-get install <PACKAGE>=<NEW-K8S-VERSION>` for all packages.

Now the new version of the `kubelet` should be running on the host. Verify this using the following command on the respective host:

```
systemctl status kubelet
```

Verify that the upgraded node is available again by executing the following from wherever you run `kubectl` commands:

```
kubectl get nodes
```

If the `STATUS` column of the above command shows `Ready` for the upgraded host, you can continue (you may have to repeat this for a couple of time before the node gets `Ready`).

If something goes wrong

If the upgrade fails the situation afterwards depends on the phase in which things went wrong:

1. If `/tmp/kubeadm upgrade apply` failed to upgrade the cluster it will try to perform a rollback. Hence if that happened on the first master, chances are pretty good that the cluster is still intact.

You can run `/tmp/kubeadm upgrade apply` again as it is idempotent and should eventually make sure the actual state is the desired state you are declaring. You can use `/tmp/kubeadm upgrade apply` to change a running cluster with `x.x.x --> x.x.x` with `--force`, which can be used to recover from a bad state.

1. If `/tmp/kubeadm upgrade apply` on one of the secondary masters failed you still have a working, upgraded cluster, but with the secondary masters in a somewhat undefined condition. You will have to find out what went wrong and join the secondaries manually. As mentioned above, sometimes upgrading one of the secondary masters fails waiting for the restarted static pods first, but succeeds when the operation is simply repeated after a little pause of one or two minutes.

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

[Edit This Page](#)

Upgrading kubeadm clusters from 1.7 to 1.8

This guide is for upgrading kubeadm clusters from version 1.7.x to 1.8.x, as well as 1.7.x to 1.7.y and 1.8.x to 1.8.y where y > x. See also upgrading kubeadm clusters from 1.6 to 1.7 if you're on a 1.6 cluster currently.

- Before you begin
- Upgrading your control plane
- Upgrading your master and node packages
- Recovering from a bad state

Before you begin

Before proceeding:

- You need to have a functional kubeadm Kubernetes cluster running version 1.7.0 or higher in order to use the process described here.
- Make sure you read the release notes carefully.
- As kubeadm upgrade does not upgrade etcd make sure to back it up. You can, for example, use etcdctl backup to take care of this.
- Note that kubeadm upgrade will not touch any of your workloads, only Kubernetes-internal components. As a best-practice you should back up what's important to you. For example, any app-level state, such as a database an app might depend on (like MySQL or MongoDB) must be backed up beforehand.

Also, note that only one minor version upgrade is supported. That is, you can only upgrade from, say 1.7 to 1.8, not from 1.7 to 1.9.

Upgrading your control plane

You have to carry out the following steps by executing these commands on your master node:

1. Install the most recent version of kubeadm using curl like so:

```
export VERSION=$(curl -sSL https://dl.k8s.io/release/stable.txt) # or manually specify
export ARCH=amd64 # or: arm, arm64, ppc64le, s390x
curl -sSL https://dl.k8s.io/release/${VERSION}/bin/linux/${ARCH}/kubeadm > /usr/bin/kubeadm
chmod a+r /usr/bin/kubeadm
```

Caution: Upgrading the kubeadm package on your system prior to upgrading the control plane causes a failed upgrade. Even though kubeadm is shipped in the Kubernetes repositories, it's important to install kubeadm manually. The kubeadm team is working on fixing this limitation.

Verify that this download of kubeadm works, and has the expected version:

```
kubeadm version
```

1. If this is the first time you use `kubeadm upgrade`, in order to preserve the configuration for future upgrades, do:

Note that for below you will need to recall what CLI args you passed to `kubeadm init` the first time.

If you used flags, do:

```
kubeadm config upload from-flags [flags]
```

Where `flags` can be empty.

If you used a config file, do:

```
kubeadm config upload from-file --config [config]
```

Where the `config` is mandatory.

1. On the master node, run the following:

```
kubeadm upgrade plan
```

You should see output similar to this:

```
[preflight] Running pre-flight checks
[upgrade] Making sure the cluster is healthy:
[upgrade/health] Checking API Server health: Healthy
[upgrade/health] Checking Node health: All Nodes are healthy
[upgrade/health] Checking Static Pod manifests exists on disk: All manifests exist on disk
[upgrade/config] Making sure the configuration is correct:
[upgrade/config] Reading configuration from the cluster...
[upgrade/config] FYI: You can look at this config file with 'kubectl -n kube-system get cm kubeadm-config -o yaml'
[upgrade] Fetching available versions to upgrade to:
[upgrade/versions] Cluster version: v1.7.1
[upgrade/versions] kubeadm version: v1.8.0
[upgrade/versions] Latest stable version: v1.8.0
[upgrade/versions] Latest version in the v1.7 series: v1.7.6
```

```
Components that must be upgraded manually after you've upgraded the control plane with 'kubeadm upgrade plan'
COMPONENT      CURRENT      AVAILABLE
Kubelet        1 x v1.7.1    v1.7.6
```

Upgrade to the latest version in the v1.7 series:

COMPONENT	CURRENT	AVAILABLE
API Server	v1.7.1	v1.7.6
Controller Manager	v1.7.1	v1.7.6
Scheduler	v1.7.1	v1.7.6

Kube Proxy	v1.7.1	v1.7.6
Kube DNS	1.14.4	1.14.4

You can now apply the upgrade by executing the following command:

```
kubeadm upgrade apply v1.7.6
```

Components that must be upgraded manually after you've upgraded the control plane with 'kubeadm upgrade apply'.

COMPONENT	CURRENT	AVAILABLE
Kubelet	1 x v1.7.1	v1.8.0

Upgrade to the latest stable version:

COMPONENT	CURRENT	AVAILABLE
API Server	v1.7.1	v1.8.0
Controller Manager	v1.7.1	v1.8.0
Scheduler	v1.7.1	v1.8.0
Kube Proxy	v1.7.1	v1.8.0
Kube DNS	1.14.4	1.14.4

You can now apply the upgrade by executing the following command:

```
kubeadm upgrade apply v1.8.0
```

Note: Before you do can perform this upgrade, you have to update kubeadm to v1.8.0

The `kubeadm upgrade plan` checks that your cluster is in an upgradeable state and fetches the versions available to upgrade to in an user-friendly way.

1. Pick a version to upgrade to and run, for example, `kubeadm upgrade apply` as follows:

```
kubeadm upgrade apply v1.8.0
```

You should see output similar to this:

```
[preflight] Running pre-flight checks
[upgrade] Making sure the cluster is healthy:
[upgrade/health] Checking API Server health: Healthy
[upgrade/health] Checking Node health: All Nodes are healthy
[upgrade/health] Checking Static Pod manifests exists on disk: All manifests exist on disk
[upgrade/config] Making sure the configuration is correct:
[upgrade/config] Reading configuration from the cluster...
[upgrade/config] FYI: You can look at this config file with 'kubectl -n kube-system get cm kubeadm-config -o yaml'
```

```

[upgrade/version] You have chosen to upgrade to version "v1.8.0"
[upgrade/versions] Cluster version: v1.7.1
[upgrade/versions] kubeadm version: v1.8.0
[upgrade/prepull] Will prepull images for components [kube-apiserver kube-controller-manager]
[upgrade/prepull] Prepulling image for component kube-scheduler.
[upgrade/prepull] Prepulling image for component kube-apiserver.
[upgrade/prepull] Prepulling image for component kube-controller-manager.
[apiclient] Found 0 Pods for label selector k8s-app=upgrade-prepull-kube-scheduler
[apiclient] Found 1 Pods for label selector k8s-app=upgrade-prepull-kube-scheduler
[apiclient] Found 1 Pods for label selector k8s-app=upgrade-prepull-kube-apiserver
[apiclient] Found 1 Pods for label selector k8s-app=upgrade-prepull-kube-controller-manager
[upgrade/prepull] Prepulled image for component kube-apiserver.
[upgrade/prepull] Prepulled image for component kube-controller-manager.
[upgrade/prepull] Prepulled image for component kube-scheduler.
[upgrade/prepull] Successfully prepulled the images for all the control plane components
[upgrade/apply] Upgrading your Static Pod-hosted control plane to version "v1.8.0"...
[upgrade/staticpods] Writing upgraded Static Pod manifests to "/etc/kubernetes/tmp/kubeadm-upgrade/manifests"
[controlplane] Wrote Static Pod manifest for component kube-apiserver to "/etc/kubernetes/manifests/kube-apiserver.yaml"
[controlplane] Wrote Static Pod manifest for component kube-controller-manager to "/etc/kubernetes/manifests/kube-controller-manager.yaml"
[controlplane] Wrote Static Pod manifest for component kube-scheduler to "/etc/kubernetes/manifests/kube-scheduler.yaml"
[upgrade/staticpods] Moved upgraded manifest to "/etc/kubernetes/manifests/kube-apiserver.yaml"
[upgrade/staticpods] Waiting for the kubelet to restart the component
[apiclient] Found 1 Pods for label selector component=kube-apiserver
[upgrade/staticpods] Component "kube-apiserver" upgraded successfully!
[upgrade/staticpods] Moved upgraded manifest to "/etc/kubernetes/manifests/kube-controller-manager.yaml"
[upgrade/staticpods] Waiting for the kubelet to restart the component
[apiclient] Found 1 Pods for label selector component=kube-controller-manager
[upgrade/staticpods] Component "kube-controller-manager" upgraded successfully!
[upgrade/staticpods] Moved upgraded manifest to "/etc/kubernetes/manifests/kube-scheduler.yaml"
[upgrade/staticpods] Waiting for the kubelet to restart the component
[apiclient] Found 1 Pods for label selector component=kube-scheduler
[upgrade/staticpods] Component "kube-scheduler" upgraded successfully!
[uploadconfig] Storing the configuration used in ConfigMap "kubeadm-config" in the "kube-system" namespace
[bootstraptoken] Configured RBAC rules to allow Node Bootstrap tokens to post CSRs in order to upgrade the control plane
[bootstraptoken] Configured RBAC rules to allow the csrapprover controller automatically approve signed CSRs from a Node Bootstrap token
[addons] Applied essential addon: kube-dns
[addons] Applied essential addon: kube-proxy

[upgrade/successful] SUCCESS! Your cluster was upgraded to "v1.8.0". Enjoy!
```

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

kubeadm upgrade apply does the following:

- It checks that your cluster is in an upgradeable state, that is:
 - The API Server is reachable,
 - All nodes are in the Ready state, and

- The control plane is healthy
- It enforces the version skew policies.
- It makes sure the control plane images are available or available to pull to the machine.
- It upgrades the control plane components or rollbacks if any of them fails to come up.
- It applies the new `kube-dns` and `kube-proxy` manifests and enforces that all necessary RBAC rules are created.

1. Manually upgrade your Software Defined Network (SDN).

Your Container Network Interface (CNI) provider might have its own upgrade instructions to follow now. Check the addons page to find your CNI provider and see if there are additional upgrade steps necessary.

1. Add RBAC permissions for automated certificate rotation. In the future, `kubeadm` will perform this step automatically:

```
kubectl create clusterrolebinding kubeadm:node-autoapprove-certificate-rotation --clusterrole=cluster-admin --user=kubeadm --group=kube-node -n kube-system
```

Upgrading your master and node packages

For each host (referred to as `$HOST` below) in your cluster, upgrade `kubelet` by executing the following commands:

1. Prepare the host for maintenance, marking it unschedulable and evicting the workload:

```
kubectl drain $HOST --ignore-daemonsets
```

When running this command against the master host, this error is expected and can be safely ignored (since there are static pods running on the master):

```
node "master" already cordoned
error: pods not managed by ReplicationController, ReplicaSet, Job, DaemonSet or StatefulSet
```

1. Upgrade the Kubernetes package versions on the `$HOST` node by using a Linux distribution-specific package manager:

If the host is running a Debian-based distro such as Ubuntu, run:

```
apt-get update
apt-get upgrade
```

If the host is running CentOS or the like, run:

```
yum update
```

Now the new version of the `kubelet` should be running on the host. Verify this using the following command on `$HOST`:

```
systemctl status kubelet
```

1. Bring the host back online by marking it schedulable:

```
kubectl uncordon $HOST
```

1. After upgrading `kublet` on each host in your cluster, verify that all nodes are available again by executing the following (from anywhere, for example, from outside the cluster):

```
kubectl get nodes
```

If the `STATUS` column of the above command shows `Ready` for all of your hosts, you are done.

Recovering from a bad state

If `kubeadm upgrade` somehow fails and fails to roll back, due to an unexpected shutdown during execution for instance, you may run `kubeadm upgrade` again as it is idempotent and should eventually make sure the actual state is the desired state you are declaring.

You can use `kubeadm upgrade` to change a running cluster with `x.x.x --> x.x.x` with `--force`, which can be used to recover from a bad state.

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

[Edit This Page](#)

Upgrading/downgrading kubeadm clusters between v1.8 to v1.9

This guide is for upgrading kubeadm clusters from version 1.8.x to 1.9.x, as well as 1.8.x to 1.8.y and 1.9.x to 1.9.y where $y > x$. See also upgrading kubeadm clusters from 1.7 to 1.8 if you're on a 1.7 cluster currently.

- Before you begin
- Upgrading your control plane
- Upgrading your master and node packages
- Recovering from a failure state

Before you begin

Before proceeding:

- You need to have a functional `kubeadm` Kubernetes cluster running version 1.8.0 or higher in order to use the process described here. Swap also needs to be disabled.
- Make sure you read the release notes carefully.
- `kubeadm upgrade` now allows you to upgrade etcd. `kubeadm upgrade` will also upgrade of etcd to 3.1.10 as part of upgrading from v1.8 to v1.9 by default. This is due to the fact that etcd 3.1.10 is the officially validated etcd version for Kubernetes v1.9. The upgrade is handled automatically by `kubeadm` for you.
- Note that `kubeadm upgrade` will not touch any of your workloads, only Kubernetes-internal components. As a best-practice you should back up what's important to you. For example, any app-level state, such as a database an app might depend on (like MySQL or MongoDB) must be backed up beforehand.

Caution: All the containers will get restarted after the upgrade, due to container spec hash value gets changed.

Also, note that only one minor version upgrade is supported. For example, you can only upgrade from 1.8 to 1.9, not from 1.7 to 1.9.

Upgrading your control plane

Execute these commands on your master node:

1. Install the most recent version of `kubeadm` using `curl` like so:

```
export VERSION=$(curl -sSL https://dl.k8s.io/release/stable.txt) # or manually specify a release
export ARCH=amd64 # or: arm, arm64, ppc64le, s390x
curl -sSL https://dl.k8s.io/release/${VERSION}/bin/linux/${ARCH}/kubeadm > /usr/bin/kubeadm
chmod a+r /usr/bin/kubeadm
```

Caution: Upgrading the `kubeadm` package on your system prior to upgrading the control plane causes a failed upgrade. Even though `kubeadm` ships in the Kubernetes repositories, it's important to install `kubeadm` manually. The `kubeadm` team is working on fixing this limitation.

Verify that this download of `kubeadm` works and has the expected version:

```
kubeadm version
```

1. On the master node, run the following:

```
kubeadm upgrade plan
```

You should see output similar to this:

```
[preflight] Running pre-flight checks
[upgrade] Making sure the cluster is healthy:
```

```
[upgrade/health] Checking API Server health: Healthy
[upgrade/health] Checking Node health: All Nodes are healthy
[upgrade/health] Checking Static Pod manifests exists on disk: All manifests exist on disk
[upgrade/config] Making sure the configuration is correct:
[upgrade/config] Reading configuration from the cluster...
[upgrade/config] FYI: You can look at this config file with 'kubectl -n kube-system get cm k
[upgrade] Fetching available versions to upgrade to:
[upgrade/versions] Cluster version: v1.8.1
[upgrade/versions] kubeadm version: v1.9.0
[upgrade/versions] Latest stable version: v1.9.0
[upgrade/versions] Latest version in the v1.8 series: v1.8.6
```

Components that must be upgraded manually after you've upgraded the control plane with 'kubeadm upgrade apply v1.8.6'

COMPONENT	CURRENT	AVAILABLE
Kubelet	1 x v1.8.1	v1.8.6

Upgrade to the latest version in the v1.8 series:

COMPONENT	CURRENT	AVAILABLE
API Server	v1.8.1	v1.8.6
Controller Manager	v1.8.1	v1.8.6
Scheduler	v1.8.1	v1.8.6
Kube Proxy	v1.8.1	v1.8.6
Kube DNS	1.14.4	1.14.5

You can now apply the upgrade by executing the following command:

```
kubeadm upgrade apply v1.8.6
```

Components that must be upgraded manually after you've upgraded the control plane with 'kubeadm upgrade apply v1.9.0'

COMPONENT	CURRENT	AVAILABLE
Kubelet	1 x v1.8.1	v1.9.0

Upgrade to the latest stable version:

COMPONENT	CURRENT	AVAILABLE
API Server	v1.8.1	v1.9.0
Controller Manager	v1.8.1	v1.9.0
Scheduler	v1.8.1	v1.9.0
Kube Proxy	v1.8.1	v1.9.0
Kube DNS	1.14.5	1.14.7

You can now apply the upgrade by executing the following command:

```
kubeadm upgrade apply v1.9.0
```

Note: Before you do can perform this upgrade, you have to update kubeadm to v1.9.0

The `kubeadm upgrade plan` checks that your cluster is upgradeable and fetches the versions available to upgrade to in an user-friendly way.

To check CoreDNS version, include the `--feature-gates=CoreDNS=true` flag to verify the CoreDNS version which will be installed in place of kube-dns.

1. Pick a version to upgrade to and run. For example:

```
kubeadm upgrade apply v1.9.0
```

You should see output similar to this:

```
[preflight] Running pre-flight checks.
[upgrade] Making sure the cluster is healthy:
[upgrade/config] Making sure the configuration is correct:
[upgrade/config] Reading configuration from the cluster...
[upgrade/config] FYI: You can look at this config file with 'kubectl -n kube-system get cm kubeadm-config -o yaml'
[upgrade/version] You have chosen to upgrade to version "v1.9.0"
[upgrade/versions] Cluster version: v1.8.1
[upgrade/versions] kubeadm version: v1.9.0
[upgrade/confirm] Are you sure you want to proceed with the upgrade? [y/N]: y
[upgrade/prepull] Will prepull images for components [kube-apiserver kube-controller-manager]
[upgrade/apply] Upgrading your Static Pod-hosted control plane to version "v1.9.0"...
[etcd] Wrote Static Pod manifest for a local etcd instance to "/etc/kubernetes/tmp/kubeadm-...
[upgrade/staticpods] Moved upgraded manifest to "/etc/kubernetes/manifests/etcd.yaml" and ba...
[upgrade/staticpods] Waiting for the kubelet to restart the component
[apiclient] Found 1 Pods for label selector component=etcd
[upgrade/staticpods] Component "etcd" upgraded successfully!
[upgrade/staticpods] Writing upgraded Static Pod manifests to "/etc/kubernetes/tmp/kubeadm-...
[controlplane] Wrote Static Pod manifest for component kube-apiserver to "/etc/kubernetes/...
[controlplane] Wrote Static Pod manifest for component kube-controller-manager to "/etc/kubernetes/...
[controlplane] Wrote Static Pod manifest for component kube-scheduler to "/etc/kubernetes/...
[upgrade/staticpods] Moved upgraded manifest to "/etc/kubernetes/manifests/kube-apiserver.yaml"
[upgrade/staticpods] Waiting for the kubelet to restart the component
[apiclient] Found 1 Pods for label selector component=kube-apiserver
[upgrade/staticpods] Component "kube-apiserver" upgraded successfully!
[upgrade/staticpods] Moved upgraded manifest to "/etc/kubernetes/manifests/kube-controller-m...
[upgrade/staticpods] Waiting for the kubelet to restart the component
[apiclient] Found 1 Pods for label selector component=kube-controller-manager
[upgrade/staticpods] Component "kube-controller-manager" upgraded successfully!
[upgrade/staticpods] Moved upgraded manifest to "/etc/kubernetes/manifests/kube-scheduler.yaml"
[upgrade/staticpods] Waiting for the kubelet to restart the component
[apiclient] Found 1 Pods for label selector component=kube-scheduler
```

```
[upgrade/staticpods] Component "kube-scheduler" upgraded successfully!
[uploadconfig] Storing the configuration used in ConfigMap "kubeadm-config" in the "kube-system" namespace
[bootstraptoken] Configured RBAC rules to allow Node Bootstrap tokens to post CSRs in order to create a Node object
[bootstraptoken] Configured RBAC rules to allow the csrapprover controller automatically approve these CSRs
[bootstraptoken] Configured RBAC rules to allow certificate rotation for all node client certificates
[addons] Applied essential addon: kube-dns
[addons] Applied essential addon: kube-proxy
```

```
[upgrade/successful] SUCCESS! Your cluster was upgraded to "v1.9.0". Enjoy!
```

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

To upgrade the cluster with CoreDNS as the default internal DNS, invoke `kubeadm upgrade apply` with the `--feature-gates=CoreDNS=true` flag. `kubeadm upgrade apply` does the following:

- Checks that your cluster is in an upgradeable state:
 - The API server is reachable,
 - All nodes are in the `Ready` state
 - The control plane is healthy
- Enforces the version skew policies.
- Makes sure the control plane images are available or available to pull to the machine.
- Upgrades the control plane components or rollbacks if any of them fails to come up.
- Applies the new `kube-dns` and `kube-proxy` manifests and enforces that all necessary RBAC rules are created.
- Creates new certificate and key files of apiserver and backs up old files if they're about to expire in 180 days.

1. Manually upgrade your Software Defined Network (SDN).

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

Upgrading your master and node packages

For each host (referred to as `$HOST` below) in your cluster, upgrade `kubelet` by executing the following commands:

1. Prepare the host for maintenance, marking it unschedulable and evicting the workload:

```
kubectl drain $HOST --ignore-daemonsets
```

When running this command against the master host, this error is expected and can be safely ignored (since there are static pods running on the master):

```
node "master" already cordoned
error: pods not managed by ReplicationController, ReplicaSet, Job, DaemonSet or StatefulSet
```

1. Upgrade the Kubernetes package versions on the \$HOST node by using a Linux distribution-specific package manager:

If the host is running a Debian-based distro such as Ubuntu, run:

```
apt-get update
apt-get upgrade
```

If the host is running CentOS or the like, run:

```
yum update
```

Now the new version of the `kubelet` should be running on the host. Verify this using the following command on \$HOST:

```
systemctl status kubelet
```

1. Bring the host back online by marking it schedulable:

```
kubectl uncordon $HOST
```

1. After upgrading `kubelet` on each host in your cluster, verify that all nodes are available again by executing the following (from anywhere, for example, from outside the cluster):

```
kubectl get nodes
```

If the STATUS column of the above command shows Ready for all of your hosts, you are done.

Recovering from a failure state

If `kubeadm upgrade` somehow fails and fails to roll back, for example due to an unexpected shutdown during execution, you can run `kubeadm upgrade` again as it is idempotent and should eventually make sure the actual state is the desired state you are declaring.

You can use `kubeadm upgrade` to change a running cluster with `x.x.x --> x.x.x` with `--force`, which can be used to recover from a bad state.

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

[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:

```
cpu-constraints.yaml
```

```
docs/tasks/administer-cluster/manage-resources
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 create -f https://k8s.io/docs/tasks/administer-cluster/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`.

```
cpu-constraints-pod.yaml
docs/tasks/administer-cluster/manage-resources
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 create -f https://k8s.io/docs/tasks/administer-cluster/cpu-constraints-pod.yaml --name=constraints-cpu-demo
```

Verify that the Pod's Container is running:

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

View detailed information about the Pod:

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

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

```
resources:
```

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

Delete the Pod

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

Attempt to create a Pod that exceeds the maximum CPU constraint

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

```
cpu-constraints-pod-2.yaml
```

```
docs/tasks/administer-cluster/manage-resources  
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 create -f https://k8s.io/docs/tasks/administer-cluster/cpu-constraints-pod-2.yaml --
```

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 "docs/tasks/administer-cluster/cpu-constraints-pod-2.yaml": pods "constraints-cpu-demo-2" is forbidden: maximum cpu usage per Container is 800m, but limit is 1.5m
```

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.

```
cpu-constraints-pod-3.yaml
docs/tasks/administer-cluster/manage-resources
apiVersion: v1
kind: Pod
metadata:
  name: constraints-cpu-demo-4
spec:
  containers:
  - name: constraints-cpu-demo-4-ctr
    image: nginx
    resources:
      limits:
        cpu: "800m"
      requests:
        cpu: "100m"
```

Attempt to create the Pod:

```
kubectl create -f https://k8s.io/docs/tasks/administer-cluster/cpu-constraints-pod-3.yaml --
```

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 "docs/tasks/administer-cluster/cpu-constraints-pod-3.yaml": pods "constraints-cpu-demo-4" is forbidden: minimum cpu usage per Container is 200m, but requested 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.

```
cpu-constraints-pod-4.yaml
```

```
docs/tasks/administer-cluster/manage-resources
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 create -f https://k8s.io/docs/tasks/administer-cluster/cpu-constraints-pod-4.yaml --
```

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

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

[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
- 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.

```
memory-defaults.yaml
docs/tasks/administer-cluster/manage-resources
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 create -f https://k8s.io/docs/tasks/administer-cluster/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.

```
memory-defaults-pod.yaml
docs/tasks/administer-cluster/manage-resources
apiVersion: v1
kind: Pod
metadata:
  name: default-mem-demo
spec:
  containers:
    - name: default-mem-demo-ctr
      image: nginx
```

Create the Pod.

```
kubectl create -f https://k8s.io/docs/tasks/administer-cluster/memory-defaults-pod.yaml --name=mem-demo
```

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:

```
memory-defaults-pod-2.yaml
docs/tasks/administer-cluster/manage-resources
apiVersion: v1
kind: Pod
metadata:
  name: default-mem-demo-2
spec:
  containers:
  - name: defalt-mem-demo-2-ctr
    image: nginx
    resources:
      limits:
        memory: "1Gi"
```

Create the Pod:

```
kubectl create -f https://k8s.io/docs/tasks/administer-cluster/memory-defaults-pod-2.yaml --
```

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:

```
memory-defaults-pod-3.yaml
```

```
docs/tasks/administer-cluster/manage-resources
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 create -f https://k8s.io/docs/tasks/administer-cluster/memory-defaults-pod-3.yaml --
```

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.

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

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

[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
- 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.

```
cpu-defaults.yaml
```

```
docs/tasks/administer-cluster/manage-resources
```

```
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 create -f https://k8s.io/docs/tasks/administer-cluster/cpu-defaults.yaml --namespace
```

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.

```
cpu-defaults-pod.yaml
```

```
docs/tasks/administer-cluster/manage-resources
```

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

Create the Pod.

```
kubectl create -f https://k8s.io/docs/tasks/administer-cluster/cpu-defaults-pod.yaml --name
```

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:

```
cpu-defaults-pod-2.yaml
```

```
docs/tasks/administer-cluster/manage-resources
```

```
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 create -f https://k8s.io/docs/tasks/administer-cluster/cpu-defaults-pod-2.yaml --name
```

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:

```
cpu-defaults-pod-3.yaml
docs/tasks/administer-cluster/manage-resources
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 create -f https://k8s.io/docs/tasks/administer-cluster/cpu-defaults-pod-3.yaml --name=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.

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

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

[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:

```
memory-constraints.yaml
```

```
docs/tasks/administer-cluster/manage-resources
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 create -f https://k8s.io/docs/tasks/administer-cluster/memory-constraints.yaml --name=mem-min-max-demo-lr
```

View detailed information about the LimitRange:

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

The output shows the minimum and maximum memory constraints as expected.

But notice that even though you didn't specify default values in the configuration file for the LimitRange, they were created automatically.

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

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

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

- Verify that the Container has a memory limit that is less than or equal to 1 GiB.

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

```
memory-constraints-pod.yaml
docs/tasks/administer-cluster/manage-resources
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 create -f https://k8s.io/docs/tasks/administer-cluster/memory-constraints-pod.yaml
```

Verify that the Pod's Container is running:

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

View detailed information about the Pod:

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

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

```
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.

```
memory-constraints-pod-2.yaml
```

```
docs/tasks/administer-cluster/manage-resources
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 create -f https://k8s.io/docs/tasks/administer-cluster/memory-constraints-pod-2.yaml
```

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 "docs/tasks/administer-cluster/memory-constraints-pod-2.yaml": pods "constraints-mem-demo-2" is forbidden: maximum memory usage per Container is 1Gi, but 1.5Gi was requested
```

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 200 MiB and a memory limit of 800 MiB.

```
memory-constraints-pod-3.yaml
```

```
docs/tasks/administer-cluster/manage-resources
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 create -f https://k8s.io/docs/tasks/administer-cluster/memory-constraints-pod-3.yaml
```

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 "docs/tasks/administer-cluster/memory-constraints-pod-3.yaml": pods "constraints-mem-demo-3" is forbidden: minimum memory usage per Container is 500Mi, but requested 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.

```
memory-constraints-pod-4.yaml
```

```
docs/tasks/administer-cluster/manage-resources
apiVersion: v1
kind: Pod
metadata:
  name: constraints-mem-demo-4
spec:
  containers:
    - name: constraints-mem-demo-4-ctr
      image: nginx
```

Create the Pod:

```
kubectl create -f https://k8s.io/docs/tasks/administer-cluster/memory-constraints-pod-4.yaml
```

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

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

[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:

```
cpu-constraints.yaml
docs/tasks/administer-cluster/manage-resources
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 create -f https://k8s.io/docs/tasks/administer-cluster/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.

```
cpu-constraints-pod.yaml
```

```
docs/tasks/administer-cluster/manage-resources
```

```
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 create -f https://k8s.io/docs/tasks/administer-cluster/cpu-constraints-pod.yaml --na
```

Verify that the Pod's Container is running:

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

View detailed information about the Pod:

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

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

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

Delete the Pod

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

Attempt to create a Pod that exceeds the maximum CPU constraint

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

```
cpu-constraints-pod-2.yaml
```

```
docs/tasks/administer-cluster/manage-resources
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 create -f https://k8s.io/docs/tasks/administer-cluster/cpu-constraints-pod-2.yaml --
```

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 "docs/tasks/administer-cluster/cpu-constraints-pods "constraints-cpu-demo-2" is forbidden: maximum cpu usage per Container is 800m, but limit
```

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.

```
cpu-constraints-pod-3.yaml
```

```
docs/tasks/administer-cluster/manage-resources
apiVersion: v1
kind: Pod
metadata:
  name: constraints-cpu-demo-4
spec:
  containers:
  - name: constraints-cpu-demo-4-ctr
    image: nginx
    resources:
      limits:
        cpu: "800m"
      requests:
        cpu: "100m"
```

Attempt to create the Pod:

```
kubectl create -f https://k8s.io/docs/tasks/administer-cluster/cpu-constraints-pod-3.yaml --
```

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 "docs/tasks/administer-cluster/cpu-constraints-pods "constraints-cpu-demo-4" is forbidden: minimum cpu usage per Container is 200m, but requested
```

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.

```
cpu-constraints-pod-4.yaml
docs/tasks/administer-cluster/manage-resources
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 create -f https://k8s.io/docs/tasks/administer-cluster/cpu-constraints-pod-4.yaml --
```

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

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

[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:

```
quota-mem-cpu.yaml
```

```
docs/tasks/administer-cluster/manage-resources
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 create -f https://k8s.io/docs/tasks/administer-cluster/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:

```
quota-mem-cpu-pod.yaml
docs/tasks/administer-cluster/manage-resources
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 create -f https://k8s.io/docs/tasks/administer-cluster/quota-mem-cpu-pod.yaml --name=
```

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:

```
quota-mem-cpu-pod-2.yaml  
docs/tasks/administer-cluster/manage-resources  
apiVersion: v1  
kind: Pod  
metadata:  
  name: quota-mem-cpu-demo-2  
spec:  
  containers:  
    - name: quota-mem-cpu-demo-2-ctr  
      image: redis  
      resources:  
        limits:  
          memory: "1Gi"  
          cpu: "800m"  
        requests:  
          memory: "700Mi"  
          cpu: "400m"
```

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

Attempt to create the Pod:

```
kubectl create -f https://k8s.io/docs/tasks/administer-cluster/quota-mem-cpu-pod-2.yaml --n
```

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 "docs/tasks/administer-cluster/quota-mem-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

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

[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:

```
quota-pod.yaml docs/tasks/administer-cluster/manage-resources
```

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

Create the ResourceQuota:

```
kubectl create -f https://k8s.io/docs/tasks/administer-cluster/quota-pod.yaml --namespace=qu
```

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:

```
quota-pod-deployment.yaml
```

```
docs/tasks/administer-cluster/manage-resources
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 create -f https://k8s.io/docs/tasks/administer-cluster/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

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

[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.

Installation in a minikube setup uses a simple “all-in-one” YAML file that includes DaemonSet configurations for Cilium, to connect to the minikube’s etcd instance as well as appropriate RBAC settings:

```
$ kubectl create -f https://raw.githubusercontent.com/cilium/cilium/master/examples/kubernetes/cilium-configmap "cilium-config" created
secret "cilium-etcd-secrets" created
serviceaccount "cilium" created
clusterrolebinding "cilium" created
daemonset "cilium" created
clusterrole "cilium" 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	DESIRED	CURRENT	READY	NODE-SELECTOR	AGE
cilium	1	1	1	<none>	2m
...					

There are two main components to be aware of:

- One `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.
- For production deployments, Cilium should leverage the key-value store cluster (e.g., etcd) used by Kubernetes, which typically runs on the Kubernetes master nodes. The Cilium Kubernetes Installation Guide includes an example DaemonSet which can be customized to point to this key-value store cluster. The simple “all-in-one” DaemonSet for minikube requires no such configuration because it automatically connects to the minikube’s etcd instance.

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.

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

[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.

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

[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.

Installation in a minikube setup uses a simple “all-in-one” YAML file that includes DaemonSet configurations for Cilium, to connect to the minikube’s etcd instance as well as appropriate RBAC settings:

```
$ kubectl create -f https://raw.githubusercontent.com/cilium/cilium/master/examples/kubernetes/cilium-config.yaml
configmap "cilium-config" created
secret "cilium-etcd-secrets" created
serviceaccount "cilium" created
clusterrolebinding "cilium" created
daemonset "cilium" created
clusterrole "cilium" 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	DESIRED	CURRENT	READY	NODE-SELECTOR	AGE
cilium	1	1	1	<none>	2m
...					

There are two main components to be aware of:

- One `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.
- For production deployments, Cilium should leverage the key-value store cluster (e.g., etcd) used by Kubernetes, which typically runs on the Kubernetes master nodes. The Cilium Kubernetes Installation Guide includes an example DaemonSet which can be customized to point to this key-value store cluster. The simple “all-in-one” DaemonSet for minikube requires no such configuration because it automatically connects to the minikube’s etcd instance.

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.

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

[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.

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

[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 kubeadmin.

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.

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

[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 po -n kube-system -o wide
```

The output is similar to this:

NAME	READY	STATUS	RESTARTS	AGE	IP
weave-net-1t1qg	2/2	Running	0	9d	192.168.2.1
weave-net-231d7	2/2	Running	1	7d	10.2.0.17
weave-net-7nmwt	2/2	Running	3	9d	192.168.2.1
weave-net-pmw8w	2/2	Running	0	9d	192.168.2.1

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.

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

[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 cluster 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 cluster 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/
{
  "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:

```
$ APISERVER=$(kubectl config view | grep server | cut -f 2- -d ":" | tr -d " ")
$ TOKEN=$(kubectl describe secret $(kubectl get secrets | grep default | cut -f1 -d ' ') | grep -E 'token|secret' | awk '{print $2}')
$ 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 and Python.

Go client

- To get the library, run the following command: `go get k8s.io/client-go/<version number>/kubernetes` 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/1.4/pkg/api/v1"` 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/client-go/1.4/kubernetes"
    "k8s.io/client-go/1.4/pkg/api/v1"
    "k8s.io/client-go/1.4/tools/clientcmd"
)
...
// uses the current context in kubeconfig
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, 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 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))

```

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 API server are somewhat different.

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.

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.

From within a Pod, 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/namespaces` in each container.

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

- Use one of the official client libraries as they handle API host discovery and authentication automatically. For Go client, the `rest.InClusterConfig()` function assists with this. See an example here.
- 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.

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

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

[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
```

```
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
kibana-logging is running at https://104.197.5.247/api/v1/namespaces/kube-system/services/kibana-logging
kube-dns is running at https://104.197.5.247/api/v1/namespaces/kube-system/services/kube-dns
grafana is running at https://104.197.5.247/api/v1/namespaces/kube-system/services/grafana
heapster is running at https://104.197.5.247/api/v1/namespaces/kube-system/services/heapster
```

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`.

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/`.
(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/namespace_name/services/service_name[:port]/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`

```
{  
    "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.

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

[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.

FEATURE STATE: Kubernetes v1.10 stable

This feature is *stable*, meaning:

- The version name is vX where X is an integer.
- Stable versions of features will appear in released software for many subsequent versions.
- 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": {
  "alpha.kubernetes.io/nvidia-gpu": "0",
  "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:  
alpha.kubernetes.io/nvidia-gpu: 0  
cpu: 2  
memory: 2049008Ki  
example.com/dongle: 4
```

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

Discussion

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

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

Storage example

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

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

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

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

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

Clean up

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

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

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

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

```
kubectl proxy
```

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

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

Verify that the dongle advertisement has been removed:

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

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

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

[Edit This Page](#)

Autoscale the DNS Service in a Cluster

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

- Before you begin
- Determining whether DNS horizontal autoscaling is already enabled
- Getting the name of your DNS Deployment or ReplicationController
- Determining your scale target
- Enabling DNS horizontal autoscaling
- Tuning autoscaling parameters
- Disable DNS horizontal autoscaling
- Understanding how DNS horizontal autoscaling works
- Future enhancements
- 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`.

Determining whether DNS horizontal autoscaling is already enabled

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

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

The output is similar to this:

NAME	DESIRED	CURRENT	UP-TO-DATE	AVAILABLE	AGE
...					
kube-dns-autoscaler	1	1	1	1	...
...					

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

Getting the name of your DNS Deployment or ReplicationController

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

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

The output is similar to this:

NAME	DESIRED	CURRENT	UP-TO-DATE	AVAILABLE	AGE
...					
kube-dns	1	1	1	1	...
...					

In Kubernetes versions earlier than 1.5 DNS is implemented using a ReplicationController instead of a Deployment. So if you don't see kube-dns, or a similar name, in the preceding output, list the ReplicationControllers in your cluster in the kube-system namespace:

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

The output is similar to this:

NAME	DESIRED	CURRENT	READY	AGE
...				
kube-dns-v20	1	1	1	...
...				

Determining your scale target

If you have a DNS Deployment, your scale target is:

`Deployment/<your-deployment-name>`

where is the name of your DNS Deployment. For example, if your DNS Deployment name is kube-dns, your scale target is `Deployment/kube-dns`.

If you have a DNS ReplicationController, your scale target is:

`ReplicationController/<your-rc-name>`

where is the name of your DNS ReplicationController. For example, if your DNS ReplicationController name is kube-dns-v20, your scale target is `ReplicationController/kube-dns-v20`.

Enabling DNS horizontal autoscaling

In this section, you create a 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:

```
dns-horizontal-autoscaler.yaml docs/tasks/administer-cluster
```

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: kube-dns-autoscaler
  namespace: kube-system
  labels:
    k8s-app: kube-dns-autoscaler
spec:
  selector:
    matchLabels:
      k8s-app: kube-dns-autoscaler
  template:
    metadata:
      labels:
        k8s-app: kube-dns-autoscaler
    spec:
      containers:
        - name: autoscaler
          image: k8s.gcr.io/cluster-proportional-autoscaler-amd64:1.1.1
          resources:
            requests:
              cpu: "20m"
              memory: "10Mi"
          command:
            - /cluster-proportional-autoscaler
            - --namespace=kube-system
            - --configmap=kube-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 create -f dns-horizontal-autoscaler.yaml
```

The output of a successful command is:

```
deployment "kube-dns-autoscaler" created
```

DNS horizontal autoscaling is now enabled.

Tuning autoscaling parameters

Verify that the kube-dns-autoscaler ConfigMap exists:

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

The output is similar to this:

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

Modify the data in the ConfigMap:

```
kubectl edit configmap kube-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 number 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 integers.

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 turning DNS horizontal autoscaling. Which option to use depends on different conditions.

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

This option works for all situations. Enter this command:

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

The output is:

```
deployment "kube-dns-autoscaler" scaled
```

Verify that the replica count is zero:

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

The output displays 0 in the DESIRED and CURRENT columns:

NAME	DESIRED	CURRENT	UP-TO-DATE	AVAILABLE	AGE
...					
kube-dns-autoscaler	0	0	0	0	...
...					

Option 2: Delete the kube-dns-autoscaler deployment

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

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

The output is:

```
deployment "kube-dns-autoscaler" deleted
```

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

This option works if kube-dns-autoscaler is under control of the Addon Manager's control, 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 kube-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 kube-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*.

Future enhancements

Control patterns, in addition to linear and ladder, that consider custom metrics are under consideration as a future development.

Scaling of DNS backends based on DNS-specific metrics is under consideration as a future development. The current implementation, which uses the number of nodes and cores in cluster, is limited.

Support for custom metrics, similar to that provided by Horizontal Pod Autoscaling, is under consideration as a future development.

What's next

Learn more about the implementation of cluster-proportional-autoscaler.

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

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

pvc-bb3ca71d-b7b5-11e6-9d58-0ed433a7dd94	4Gi	RWO	Delete	Bou
--	-----	-----	--------	-----

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

1. Choose one of your PersistentVolumes and change its reclaim policy:
kubectl patch pv -p '{"spec":{"persistentVolumeReclaimPolicy":"Retain"}}'
where <your-pv-name> is the name of your chosen PersistentVolume.
2. Verify that your chosen PersistentVolume has the right policy:
kubectl get pv

The output is similar to this:

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

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.

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

[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	TYPE
standard (default)	<code>kubernetes.io/gce-pd</code>
gold	<code>kubernetes.io/gce-pd</code>

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

1. 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.

1. 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, Kubernetes ignores the annotation, i.e. it behaves as if there is no default StorageClass.

1. Verify that your chosen StorageClass is default:

```
kubectl get storageclass
```

The output is similar to this:

NAME	TYPE
standard	<code>kubernetes.io/gce-pd</code>
gold (default)	<code>kubernetes.io/gce-pd</code>

What's next

- Learn more about StorageClasses.

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

[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 to 1.6

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 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
- ...

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 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-minion-group --size=42 --zone=$ZONE
```

Instance Group will take care of putting appropriate image on new machines and start them, while Kubelet will register its Node with 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 documenta-

tation.

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 KUBE_AUTOSCALER_MAX_NODES=10
```

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  
gcloud container clusters update mytestcluster --enable-autoscaling --min-nodes=1 --max-nodes=10
```

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.

`KUBE_API_VERSIONS` environment variable for the `kube-apiserver` binary which controls the API versions that are supported in the cluster. The first version in the list is used as the cluster’s storage version. Hence, to set a specific version as the storage version, bring it to the front of list of versions in the value of `KUBE_API_VERSIONS`. You need to restart the `kube-apiserver` binary for changes to this variable to take effect.

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.

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

[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/dockerized/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`:

```
my-scheduler.yaml docs/tasks/administer-cluster
apiVersion: v1
kind: ServiceAccount
metadata:
  name: my-scheduler
  namespace: kube-system
---
kind: ClusterRoleBinding
apiVersion: rbac.authorization.k8s.io/v1
metadata:
  name: my-scheduler-as-kube-scheduler
subjects:
- kind: ServiceAccount
  name: my-scheduler
  namespace: kube-system
roleRef:
  kind: ClusterRole
  name: 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
```

```
my-scheduler.yaml docs/tasks/administer-cluster
```

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
-

```

pod1.yaml docs/tasks/administer-cluster
-----
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`
-

```
pod2.yaml docs/tasks/administer-cluster
```

```
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`

```
pod3.yaml docs/tasks/administer-cluster
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
```

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

[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`.

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

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
<code>MemoryPressure</code>	<code>memory.available</code>
<code>DiskPressure</code>	<code>nodefs.available, nodefs.inodesFree, imagefs.available, or imagefs.inodesFree</code>

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 kill 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=200Gi"
```

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
<code>MemoryPressure</code>	No new <code>BestEffort</code> Pods are scheduled to the node.
<code>DiskPressure</code>	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` is 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.

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

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

It is never desired for `kubelet` to evict a `DaemonSet` Pod, since the Pod is immediately recreated and rescheduled back to the same node.

At the moment, the `kubelet` has no ability to distinguish a Pod created from `DaemonSet` versus any other object. If/when that information is available, the `kubelet` could proactively filter those Pods from the candidate set of Pods provided to the eviction strategy.

In general, it is strongly recommended that `DaemonSet` not create `BestEffort` Pods to avoid being identified as a candidate Pod for eviction. Instead `DaemonSet` should ideally launch `Guaranteed` Pods.

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
<code>--image-gc-high-threshold</code>	<code>--eviction-hard</code> or <code>--eviction-soft</code>
<code>--image-gc-low-threshold</code>	<code>--eviction-minimum-reclaim</code>
<code>--maximum-dead-containers</code>	deprecated
<code>--maximum-dead-containers-per-container</code>	deprecated
<code>--minimum-container-ttl-duration</code>	deprecated
<code>--low-diskspace-threshold-mb</code>	<code>--eviction-hard</code> or <code>--eviction-soft</code>
<code>--outofdisk-transition-frequency</code>	<code>--eviction-pressure-transition-period</code>

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.

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

[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:

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

Create the ResourceQuota:

```
kubectl create -f https://k8s.io/docs/tasks/administer-cluster/quota-objects.yaml --namespace
```

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:

```
quota-objects-pvc.yaml docs/tasks/administer-cluster  
kind: PersistentVolumeClaim  
apiVersion: v1  
metadata:  
  name: pvc-quota-demo  
spec:  
  storageClassName: manual  
  accessModes:  
    - ReadWriteOnce  
  resources:  
    requests:  
      storage: 3Gi
```

Create the PersistentVolumeClaim:

```
kubectl create -f https://k8s.io/docs/tasks/administer-cluster/quota-objects-pvc.yaml --name
```

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:

```
quota-objects-pvc-2.yaml docs/tasks/administer-cluster
kind: PersistentVolumeClaim
apiVersion: v1
metadata:
  name: pvc-quota-demo-2
spec:
  storageClassName: manual
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: 4Gi
```

Attempt to create the second PersistentVolumeClaim:

```
kubectl create -f https://k8s.io/docs/tasks/administer-cluster/quota-objects-pvc-2.yaml --n
```

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

String	API Object
”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

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

[Edit This Page](#)

Control CPU Management Policies on the Node

FEATURE STATE: Kubernetes v1.10 beta

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

- The version names contain beta (e.g. v2beta3).
- Code is well tested. Enabling the feature is considered safe. Enabled by default.
- Support for the overall feature will not be dropped, though details may change.
- The schema and/or semantics of objects may change in incompatible ways in a subsequent beta or stable release. When this happens, we will provide instructions for migrating to the next version. This may require deleting, editing, and re-creating API objects. The editing process may require some thought. This may require downtime for applications that rely on the feature.
- Recommended for only non-business-critical uses because of potential for incompatible changes in subsequent releases. If you have multiple clusters that can be upgraded independently, you may be able to relax this restriction.
- **Please do try our beta features and give feedback on them! After they exit beta, it may not be practical for us to make more changes.**

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

- Before you begin
- CPU Management Policies

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`.

CPU Management Policies

By default, the kubelet uses CFS quota to enforce pod CPU limits. When the node runs many CPU-bound pods, the workload can move to different CPU cores depending on whether the pod is throttled and which CPU cores are

available at scheduling time. Many workloads are not sensitive to this migration and thus work fine without any intervention.

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

Configuration

The CPU Manager is an alpha feature in Kubernetes v1.8. It was enabled by default as a beta feature since v1.10.

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

- `none`: the default, which represents the existing scheduling behavior.
- `static`: allows pods with certain resource characteristics to be granted increased CPU affinity and exclusivity on the node.

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

None policy

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

Static policy

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

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

Note: The alpha version of this policy does not guarantee static exclusive allocations across Kubelet restarts.

This policy manages a shared pool of CPUs that initially contains all CPUs in the node. The amount of exclusively allocatable CPUs is equal to the total number of CPUs in the node minus any CPU reservations by the kubelet

--kube-reserved or --system-reserved options. CPUs reserved by these options are taken, in integer quantity, from the initial shared pool in ascending order by physical core ID. This shared pool is the set of CPUs on which any containers in BestEffort and Burstable pods run. Containers in Guaranteed pods with fractional CPU requests also run on CPUs in the shared pool. Only containers that are both part of a Guaranteed pod and have integer CPU requests are assigned exclusive CPUs.

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

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

Consider the containers in the following pod specs:

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

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

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

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

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

```

limits:
  memory: "200Mi"
  cpu: "2"
requests:
  memory: "100Mi"
  cpu: "1"

```

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

```

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

```

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

```

spec:
  containers:
    - name: nginx
      image: nginx
      resources:
        limits:
          memory: "200Mi"
          cpu: "1.5"
        requests:
          memory: "200Mi"
          cpu: "1.5"

```

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

```

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

```

```
cpu: "2"
```

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

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

[Edit This Page](#)

Customizing DNS Service

This page provides hints on configuring DNS Pod and guidance on customizing the DNS resolution process.

- Before you begin
- Introduction
- Inheriting DNS from the node
- Configure stub-domain and upstream DNS servers
- ConfigMap options
- 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 `kube-dns` addon.

Introduction

Starting from Kubernetes v1.3, DNS is a built-in service launched automatically using the addon manager cluster add-on.

The running Kubernetes DNS pod holds 3 containers:

- “**kubedns**”: The **kubedns** process watches the Kubernetes master for changes in Services and Endpoints, and maintains in-memory lookup structures to serve DNS requests.
- “**dnsmasq**”: The **dnsmasq** container adds DNS caching to improve performance.
- “**sidecar**”: The **sidecar** container provides a single health check endpoint while performing dual healthchecks (for **dnsmasq** and **kubedns**).

The DNS pod is exposed as a Kubernetes Service with a static IP. Once assigned the kubelet passes DNS configured using the `--cluster-dns=<dns-service-ip>` flag to each container.

DNS names also need domains. The local domain is configurable in the kubelet using the flag `--cluster-domain=<default-local-domain>`.

The Kubernetes cluster DNS server is based off the SkyDNS library. It supports forward lookups (A records), service lookups (SRV records) and reverse IP address lookups (PTR records).

Inheriting DNS from the node

When running a pod, kubelet will prepend the cluster DNS server and search paths to the node’s own DNS settings. If the node is able to resolve DNS names specific to the larger environment, pods should be able to, also. See Known issues below for a caveat.

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. Setting it to “” means that pods will not inherit DNS. Setting it to a valid file path means that kubelet will use this file instead of `/etc/resolv.conf` for DNS inheritance.

Configure stub-domain and upstream DNS servers

Cluster administrators can specify custom stub domains and upstream name-servers 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"]
```

As specified, 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 would 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)
widget.com	upstream DNS (one of 8.8.8.8, 8.8.4.4)

See ConfigMap options for details about the configuration option format.

Impacts on Pods

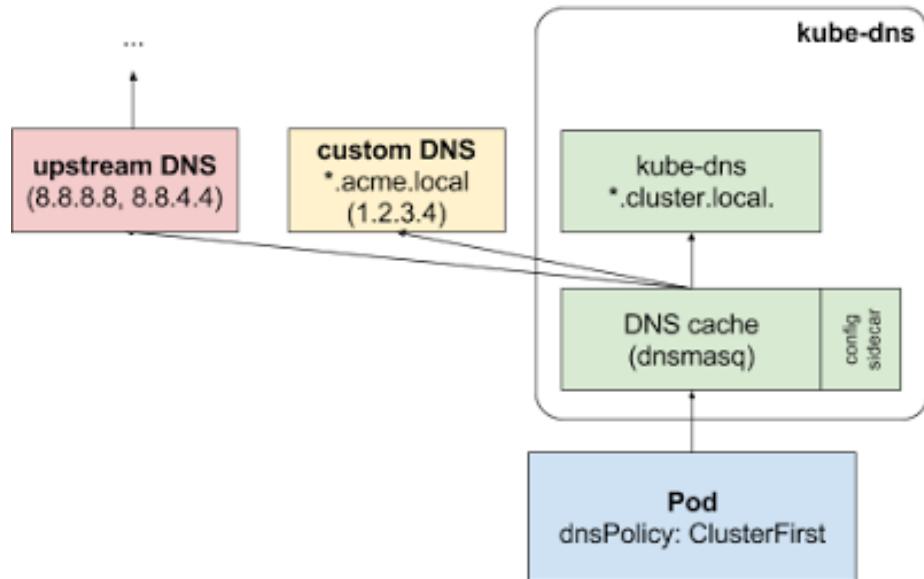
Custom upstream nameservers and stub domains won’t impact Pods that have their `dnsPolicy` set to “Default” or “None”.

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 (as in the previous example), DNS queries will be 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* (e.g.“.cluster.local”): The request is sent to kube-dns.
 - *Names with the stub domain suffix* (e.g. “.acme.local”): The request is sent to the configured custom DNS resolver (e.g. listening at 1.2.3.4).
 - *Names without a matching suffix* (e.g.“widget.com”): The request is forwarded to the upstream DNS (e.g. 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
<code>stubDomains</code> (optional)	A JSON map using a DNS suffix key (e.g. “ <code>acme.local</code> ”) and a value consisting of one or more IP addresses.
<code>upstreamNameservers</code> (optional)	A JSON array of DNS IPs.

Examples

Example: Stub domain

In this example, the user has a Consul DNS service discovery system that they wish 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 simply creates a ConfigMap object as shown below.

```

apiVersion: v1
kind: ConfigMap
metadata:
  name: kube-dns
  namespace: kube-system
data:
  stubDomains: |
    {
      "consul": "10.150.0.1"
    }

```

```
{"consul.local": ["10.150.0.1"]}
```

Note that the cluster administrator did not wish 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. Again, this is easy to accomplish; they just need to 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"]
```

What's next

- Debugging DNS Resolution.

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

[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 `kube-dns` addon.

Create a simple Pod to use as a test environment

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

```
busybox.yaml docs/tasks/administer-cluster
apiVersion: v1
kind: Pod
metadata:
  name: busybox
  namespace: default
spec:
  containers:
    - name: busybox
      image: busybox
      command:
        - sleep
        - "3600"
      imagePullPolicy: IfNotPresent
  restartPolicy: Always
```

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

```
$ kubectl create -f busybox.yaml
pod "busybox" created

$ kubectl get pods busybox
NAME      READY      STATUS      RESTARTS      AGE
busybox   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 busybox -- 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 busybox 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
nameserver 10.0.0.10
options ndots:5
```

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

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

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

or

```
$ kubectl exec -ti busybox -- nslookup kubernetes.default
Server: 10.0.0.10
Address 1: 10.0.0.10 kube-dns.kube-system.svc.cluster.local
```

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

Check if the DNS pod is running

Use the `kubectl get pods` command to verify that the DNS pod is running.

```
$ kubectl get pods --namespace=kube-system -l k8s-app=kube-dns
NAME                  READY     STATUS    RESTARTS   AGE
...
kube-dns-v19-ezo1y    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 daemons.

```
$ kubectl logs --namespace=kube-system $(kubectl get pods --namespace=kube-system -l k8s-app=dns)
$ kubectl logs --namespace=kube-system $(kubectl get pods --namespace=kube-system -l k8s-app=dns)
$ kubectl logs --namespace=kube-system $(kubectl get pods --namespace=kube-system -l k8s-app=dns)
```

See if there is any suspicious log. Letter ‘W’, ‘E’, ‘F’ at the beginning represent Warning, Error and 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      CLUSTER-IP     EXTERNAL-IP   PORT(S)        AGE
...
kube-dns   10.0.0.10    <none>        53/UDP,53/TCP   1h
...
```

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.

Known issues

Kubernetes installs do not configure the nodes' resolv.conf files to use the cluster DNS by default, because that process is inherently distro-specific. This should probably be implemented eventually.

Linux's libc is impossibly stuck (see this bug from 2005) with limits of just 3 DNS `nameserver` records and 6 DNS `search` records. 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`, some of those settings will be lost. As a partial workaround, the node can run `dnsmasq` which will provide more `nameserver` entries, but not more `search` entries. You can also use kubelet's `--resolv-conf` flag.

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.

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

[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

To check the version, enter `kubectl version`.

Create an `nginx` deployment and expose it via a service

To see how Kubernetes network policy works, start off by creating an `nginx` deployment and exposing it via a service.

```
$ kubectl run nginx --image=nginx --replicas=2
deployment "nginx" created
$ kubectl expose deployment nginx --port=80
service "nginx" exposed
```

This runs two `nginx` pods in the default namespace, and exposes them through a service called `nginx`.

```
$ kubectl get svc,pod
NAME           CLUSTER-IP      EXTERNAL-IP    PORT(S)      AGE
svc/kubernetes 10.100.0.1     <none>        443/TCP     46m
svc/nginx       10.100.0.16   <none>        80/TCP      33s

NAME          READY   STATUS    RESTARTS   AGE
po/nginx-701339712-e0qfq  1/1     Running   0          35s
po/nginx-701339712-o00ef  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 test, access the service from another pod in the default namespace. Make sure you haven't enabled isolation on the namespace.

Start a busybox container, and use `wget` on the `nginx` service:

```
$ kubectl run busybox --rm -ti --image=busybox /bin/sh
Waiting for pod default/busybox-472357175-y0m47 to be running, status is Pending, pod ready
```

Hit enter for command prompt

```
/ # wget --spider --timeout=1 nginx
Connecting to nginx (10.100.0.16:80)
/ #
```

Limit access to the `nginx` service

Let's say you want to limit access to the `nginx` service so that only pods with the label `access: true` can query it. To do that, create a `NetworkPolicy` that allows connections only from those pods:

```
kind: NetworkPolicy
apiVersion: networking.k8s.io/v1
metadata:
  name: access-nginx
spec:
  podSelector:
    matchLabels:
      run: nginx
  ingress:
  - from:
    - podSelector:
        matchLabels:
          access: "true"
```

Assign the policy to the service

Use `kubectl` to create a `NetworkPolicy` from the above `nginx-policy.yaml` file:

```
$ kubectl create -f nginx-policy.yaml
networkpolicy "access-nginx" created
```

Test access to the service when access label is not defined

If we attempt to access the nginx Service from a pod without the correct labels, the request will now time out:

```
$ kubectl run busybox --rm -ti --image=busybox /bin/sh
Waiting for pod default/busybox-472357175-y0m47 to be running, status is Pending, pod ready
```

Hit enter for command prompt

```
/ # wget --spider --timeout=1 nginx
Connecting to nginx (10.100.0.16:80)
wget: download timed out
/ #
```

Define access label and test again

Create a pod with the correct labels, and you'll see that the request is allowed:

```
$ kubectl run busybox --rm -ti --labels="access=true" --image=busybox /bin/sh
Waiting for pod default/busybox-472357175-y0m47 to be running, status is Pending, pod ready
```

Hit enter for command prompt

```
/ # wget --spider --timeout=1 nginx
Connecting to nginx (10.100.0.16:80)
/ #
```

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

[Edit This Page](#)

Developing Cloud Controller Manager

FEATURE STATE: Kubernetes 1.8 alpha

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

- The version names contain alpha (e.g. v1alpha1).
- Might be buggy. Enabling the feature may expose bugs. Disabled by default.
- Support for feature may be dropped at any time without notice.
- The API may change in incompatible ways in a later software release without notice.

- Recommended for use only in short-lived testing clusters, due to increased risk of bugs and lack of long-term support.

Before going into how to build your own cloud controller manager, some background on how it works under the hood is helpful. The cloud controller manager is code from `kube-controller-manager` utilizing Go interfaces to allow implementations from any cloud to be plugged in. Most of the scaffolding and generic controller implementations will be in core, but it will always exec out to the cloud interfaces it is provided, so long as the cloud provider interface is satisfied.

To dive a little deeper into implementation details, all cloud controller managers will import packages from Kubernetes core, the only difference being each project will register their own cloud providers by calling `cloud-provider.RegisterCloudProvider` where a global variable of available cloud providers is updated.

- Developing

Developing

Out of Tree

To build an out-of-tree cloud-controller-manager for your cloud, follow these steps:

1. Create a go package with an implementation that satisfies `cloud-provider.Interface`.
2. Use `main.go` in `cloud-controller-manager` from Kubernetes core as a template for your `main.go`. As mentioned above, the only difference should be the cloud package that will be imported.
3. Import your cloud package in `main.go`, ensure your package has an `init` block to run `cloudprovider.RegisterCloudProvider`.

Using existing out-of-tree cloud providers as an example may be helpful. You can find the list here.

In Tree

For in-tree cloud providers, you can run the in-tree cloud controller manager as a Daemonset in your cluster. See the running cloud controller manager docs for more details.

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

[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

To check the version, enter `kubectl version`.

Configuration and determining whether encryption at rest is already enabled

The `kube-apiserver` process accepts an argument `--experimental-encryption-provider-config` that controls how API data is encrypted in etcd. An example configuration is provided below.

Understanding the encryption at rest configuration.

```
kind: EncryptionConfig
apiVersion: v1
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: YWJjZGVmZ2hpamtsbW5vcHFyc3R1dnd4eXoxMjMONTY=

```

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.

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
<code>identity</code>	None
<code>aescbc</code>	AES-CBC with PKCS#7 padding
<code>secretbox</code>	XSalsa20 and Poly1305
<code>aesgcm</code>	AES-GCM with random nonce
<code>kms</code>	Uses envelope encryption scheme: Data is encrypted by data encryption keys (DEKs) using AES

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.

Encrypting your data

Create a new encryption config file:

```
kind: EncryptionConfig
apiVersion: v1
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 Mac OS X, run the following command:
`head -c 32 /dev/urandom | base64`
2. Place that value in the secret field.
3. Set the `--experimental-encryption-provider-config` flag on the `kube-apiserver` to point to the location of the config file.
4. Restart your API server.

IMPORTANT: 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: mydata
```

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:

```
kind: EncryptionConfig
apiVersion: v1
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.

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

[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 Heapster, 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).

- Rescheduler: guaranteed scheduling of critical add-ons
- Config

Rescheduler: guaranteed scheduling of critical add-ons

Rescheduler is deprecated as of Kubernetes 1.10 and will be removed in version 1.12 in accordance with the deprecation policy for beta features.

To avoid eviction of critical pods, you must enable priorities in scheduler before upgrading to Kubernetes 1.10 or higher.

Rescheduler ensures that critical pods created by DaemonSet controller are always scheduled (assuming the cluster has enough resources to run the critical add-on pods in the absence of regular pods). If the scheduler determines that no node has enough free resources to run the critical add-on pod given the pods that are already running in the cluster (indicated by critical add-on pod's pod condition PodScheduled set to false, the reason set to Unschedulable) the rescheduler tries to free up space for the DaemonSet critical pod by evicting some pods; then the scheduler will schedule the add-on pod.

To avoid situation when another pod is scheduled into the space prepared for the critical add-on, the chosen node gets a temporary taint "CriticalAddonsOnly" before the eviction(s) (see more details). Each critical add-on has to tolerate it, while the other pods shouldn't tolerate the taint. The taint is removed once the add-on is successfully scheduled.

Warning: currently there is no guarantee which node is chosen and which pods are being killed in order to schedule critical pods, so if rescheduler is enabled your pods might be occasionally killed for this purpose. Please ensure that rescheduler is not enabled along with priorities & preemptions in default-scheduler as rescheduler is oblivious to priorities and it may evict high priority pods, instead of low priority ones.

Config

Rescheduler doesn't have any user facing configuration (component config) or API.

Marking pod as critical when using Rescheduler.

To be considered critical, the pod has to run in the `kube-system` namespace (configurable via flag) and

- have the `scheduler.alpha.kubernetes.io/critical-pod` annotation set to empty string, and
- have the PodSpec's `tolerations` field set to `[{"key": "CriticalAddonsOnly", "operator": "Exists"}]`.

The first one marks a pod a critical. The second one is required by Rescheduler algorithm.

A pod could also be considered critical, if its priority is greater than or equal to system-critical-priority.

Marking pod as critical when priorities are enabled.

To be considered critical, the pod has to run in the `kube-system` namespace (configurable via flag) and

- Have the `priorityClass` set as “system-cluster-critical” or “system-node-critical”, the latter being the highest for entire cluster and `scheduler.alpha.kubernetes.io/critical-pod` annotation set to empty string (This will be deprecated too).

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

[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 create -f https://raw.githubusercontent.com/kubernetes-incubator/ip-masq-agent/mast
```

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  
RETURN     all   --  anywhere        10.0.0.0/8       /* ip-masq-agent: cluster-local  
MASQUERADE all   --  anywhere        anywhere          /* ip-masq-agent: outbound traffic
```

By default, the link local range (169.254.0.0/16) is also handled by the ip-masq agent, which sets up the appropriate iptables rules. To have the ip-masq-agent ignore link local, you can set `masqLinkLocal` to true in the config map.

```
nonMasqueradeCIDRs:  
  - 10.0.0.0/8  
resyncInterval: 60s  
masqLinkLocal: true
```

IP Masquerade Agent User Guide

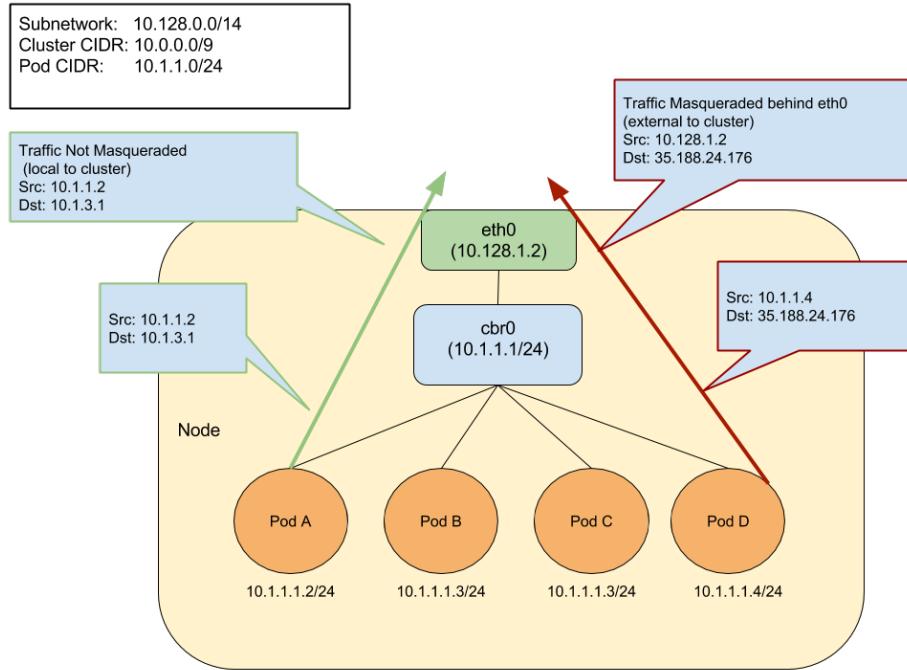
The ip-masq-agent configures iptables rules to hide a pod’s IP address behind the cluster node’s IP address. This is typically done when sending traffic to destinations outside the cluster’s pod CIDR range.

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
RETURN    all  --  anywhere            10.0.0.0/8        /* ip-masq-agent: cluster-local
RETURN    all  --  anywhere            172.16.0.0/12      /* ip-masq-agent: cluster-local
RETURN    all  --  anywhere            192.168.0.0/16     /* ip-masq-agent: cluster-local
MASQUERADE all  --  anywhere          anywhere           /* ip-masq-agent: outbound traffic
```

By default, in GCE/Google Kubernetes Engine starting with Kubernetes ver-

sion 1.7.0, if network policy is enabled or you are using a cluster CIDR not in the 10.0.0.0/8 range, the ip-masq-agent will run in your cluster. If you are running in another environment, you can add the ip-masq-agent DaemonSet to your cluster:

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

[Edit This Page](#)

Kubernetes Cloud Controller Manager

FEATURE STATE: Kubernetes v1.10 alpha

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

- The version names contain alpha (e.g. v1alpha1).
- Might be buggy. Enabling the feature may expose bugs. Disabled by default.
- Support for feature may be dropped at any time without notice.
- The API may change in incompatible ways in a later software release without notice.
- Recommended for use only in short-lived testing clusters, due to increased risk of bugs and lack of long-term support.

Kubernetes v1.6 introduced a new binary called `cloud-controller-manager`. `cloud-controller-manager` is a daemon that embeds cloud-specific control loops. These cloud-specific control loops were originally in the `kube-controller-manager`. Since cloud providers develop and release at a different pace compared to the Kubernetes project, abstracting the provider-specific code to the `cloud-controller-manager` binary allows cloud vendors to evolve independently from the core Kubernetes code.

The `cloud-controller-manager` can be linked to any cloud provider that satisfies `cloudprovider.Interface`. For backwards compatibility, the `cloud-controller-manager` provided in the core Kubernetes project uses the same cloud libraries as `kube-controller-manager`. Cloud providers already supported in Kubernetes core are expected to use the in-tree `cloud-controller-manager` to transition out of Kubernetes core. In future Kubernetes releases, all cloud controller managers will be developed outside of the core Kubernetes project managed by sig leads or cloud vendors.

- Administration
- Examples
- Limitations
- Developing your own Cloud Controller Manager

Administration

Requirements

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

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

Running `cloud-controller-manager`

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

- `kube-apiserver` and `kube-controller-manager` MUST NOT specify the `--cloud-provider` flag. This ensures that it does not run any cloud specific loops that would be run by cloud controller manager. In the future, this flag will be deprecated and removed.
- `kubelet` must run with `--cloud-provider=external`. This is to ensure that the `kubelet` is aware that it must be initialized by the cloud controller manager before it is scheduled any work.
- `kube-apiserver` SHOULD NOT run the `PersistentVolumeLabel` admission controller since the cloud controller manager takes over labeling persistent volumes. To prevent the `PersistentVolumeLabel` admission plugin from running in `kube-apiserver`, include the `PersistentVolumeLabel` as a listed value in the `--disable-admission-plugins` flag.
- For the `cloud-controller-manager` to label persistent volumes, initializers will need to be enabled and an `InitializerConfiguration` needs to be added to the system. Follow these instructions to enable initializers. Use the following YAML to create the `InitializerConfiguration`:

```
persistent-volume-label-initializer-config.yaml
```

```
docs/tasks/administer-cluster
kind:InitializerConfiguration
apiVersion:admissionregistration.k8s.io/v1alpha1
metadata:
  name:pvlabel.kubernetes.io
initializers:
  - name:pvlabel.kubernetes.io
    rules:
      - apiGroups:
          - ""
        apiVersions:
          - "*"
        resources:
          - persistentvolumes
```

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

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

As of v1.8, cloud controller manager can implement:

- node controller - responsible for updating kubernetes nodes using cloud APIs and deleting kubernetes nodes that were deleted on your cloud.
- service controller - responsible for loadbalancers on your cloud against services of type LoadBalancer.
- route controller - responsible for setting up network routes on your cloud

- PersistentVolumeLabel Admission Controller - responsible for labeling persistent volumes on your cloud - ensure that the persistent volume label admission plugin is not enabled on your kube-apiserver.
- any other features you would like to implement if you are running an out-of-tree provider.

Examples

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

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

- DigitalOcean
- keepalived
- Oracle Cloud Infrastructure
- Rancher

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

```
cloud-controller-manager-daemonset-example.yaml
```

```
docs/tasks/administer-cluster
```

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

```
---
```

```
apiVersion: v1
kind: ServiceAccount
metadata:
  name: cloud-controller-manager
  namespace: kube-system
---
```

```
kind: ClusterRoleBinding
apiVersion: rbac.authorization.k8s.io/v1
metadata:
  name: system:cloud-controller-manager
roleRef:
  apiGroup: rbac.authorization.k8s.io
  kind: ClusterRole
  name: cluster-admin
subjects:
- kind: ServiceAccount
  name: cloud-controller-manager
  namespace: kube-system
---
```

```
apiVersion: apps/v1
kind: DaemonSet
metadata:
  labels:
    k8s-app: cloud-controller-manager
  name: cloud-controller-manager
  namespace: kube-system
spec:
```

```
  selector:
    matchLabels:
      k8s-app: cloud-controller-manager
  template:
```

```
    metadata:
      labels:
        k8s-app: cloud-controller-manager
  spec:
```

```
    serviceAccountName: cloud-controller-manager
```

```
    containers:
```

```
      - name: cloud-controller-manager
        # for in-tree providers275 we use k8s.gcr.io/cloud-controller-manager
        # this can be replaced with any other image for out-of-tree providers
        image: k8s.gcr.io/cloud-controller-manager:v1.8.0
        command:
          - /usr/local/bin/cloud-controller-manager
          - --cloud-provider=<YOUR_CLOUD_PROVIDER>  # Add your own cloud provider here!
          - --leader-elect=true
          - --use-service-account-credentials
```

`cloud-controller-manager-daemonset-example.yaml`

`docs/tasks/administer-cluster`

Limitations

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

Support for Volumes

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

Scalability

In the previous architecture for cloud providers, we relied on kubelets using a local metadata service to retrieve node information about itself. With this new architecture, we now fully rely on the cloud controller managers to retrieve information for all nodes. For very larger clusters, you should consider possible bottle necks such as resource requirements and API rate limiting.

Chicken and Egg

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

A good example of this is the TLS bootstrapping feature in the Kubelet. Currently, TLS bootstrapping assumes that the Kubelet has the ability to ask the cloud provider (or a local metadata service) for all its address types (private, public, etc) but cloud controller manager cannot set a node's address types

without being initialized in the first place which requires that the kubelet has TLS certificates to communicate with the apiserver.

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

Developing your own Cloud Controller Manager

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

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

[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.

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

[Edit This Page](#)

Namespaces Walkthrough

Kubernetes *namespaces* help different projects, teams, or customers to share a Kubernetes cluster.

It does this by providing the following:

1. A scope for Names.
2. A mechanism to attach authorization and policy to a subsection of the cluster.

Use of multiple namespaces is optional.

This example demonstrates how to use Kubernetes namespaces to subdivide your cluster.

- Before you begin
- 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:

```
namespace-dev.json docs/tasks/administer-cluster
```

```
{  
    "kind": "Namespace",  
    "apiVersion": "v1",  
    "metadata": {  
        "name": "development",  
        "labels": {  
            "name": "development"  
        }  
    }  
}
```

Create the development namespace using kubectl.

```
$ kubectl create -f https://k8s.io/docs/tasks/administer-cluster/namespace-dev.json
```

Save the following contents into file `namespace-prod.json` which describes a production namespace:

```
namespace-prod.json docs/tasks/administer-cluster
```

```
{  
    "kind": "Namespace",  
    "apiVersion": "v1",  
    "metadata": {  
        "name": "production",  
        "labels": {  
            "name": "production"  
        }  
    }  
}
```

And then let's create the production namespace using kubectl.

```
$ kubectl create -f https://k8s.io/docs/tasks/administer-cluster/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: h5M0FtUUIf1BSdI7
    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=kubernetes/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      DESIRED   CURRENT   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=kubernetes/serve_hostname --replicas=5  
  
$ kubectl get deployment  
NAME      DESIRED   CURRENT   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.

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

[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.

Always have a backup plan for etcd's data for your Kubernetes cluster. For in-depth information on etcd, see etcd documentation.

- Before you begin
- Prerequisites
- Resource requirements
- Starting Kubernetes API server
- 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
- Notes for etcd Version 2.2.1

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.

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 Kubernetes API server

This section covers starting a Kubernetes API server with an etcd cluster in the deployment.

Single-node etcd cluster

Use a single-node etcd cluster only for testing purpose.

1. Run the following:

```
./etcd --listen-client-urls=http://$PRIVATE_IP:2379 --advertise-client-urls=http://$PRI
```

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://10.0.0.1:12380, 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,  
export ETCD_INITIAL_CLUSTER_STATE=existing  
etcd [flags]
```

5. Do either of the following:

- (a) Update its `--etcd-servers` flag to make Kubernetes aware of the configuration changes, then restart the Kubernetes API server.
- (b) 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. `datadir` is located at `$DATA_DIR/member/snap/db`. 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. `datadir` is located at `$DATA_DIR/member/snap/db`. 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

Important assumptions

The upgrade procedure described in this document assumes that either:

1. The etcd cluster has only a single node.
2. The etcd cluster has multiple nodes.

In this case, the upgrade procedure requires shutting down the etcd cluster. During the time the etcd cluster is shut down, the Kubernetes API Server will be read only.

Warning: Deviations from the assumptions are untested by continuous integration, and deviations might create undesirable consequences. Additional information about operating an etcd cluster is available from the etcd maintainers.

Background

As of Kubernetes version 1.5.1, we are still using etcd from the 2.2.1 release with the v2 API. Also, we have no pre-existing process for updating etcd, as we have never updated etcd by either minor or major version.

Note that we need to migrate both the etcd versions that we are using (from 2.2.1 to at least 3.0.x) as well as the version of the etcd API that Kubernetes talks to. The etcd 3.0.x binaries support both the v2 and v3 API.

This document describes how to do this migration. If you want to skip the background and cut right to the procedure, see Upgrade Procedure.

etcd upgrade requirements

There are requirements on how an etcd cluster upgrade can be performed. The primary considerations are:

- Upgrade between one minor release at a time
- Rollback supported through additional tooling

One minor release at a time

Upgrade only one minor release at a time. For example, we cannot upgrade directly from 2.1.x to 2.3.x. Within patch releases it is possible to upgrade and downgrade between arbitrary versions. Starting a cluster for any intermediate minor release, waiting until the cluster is healthy, and then shutting down the cluster will perform the migration. For example, to upgrade from version 2.1.x to 2.3.y, it is enough to start etcd in 2.2.z version, wait until it is healthy, stop it, and then start the 2.3.y version.

Rollback via additional tooling

Versions 3.0+ of etcd do not support general rollback. That is, after migrating from M.N to M.N+1, there is no way to go back to M.N. The etcd team has provided a custom rollback tool but the rollback tool has these limitations:

- This custom rollback tool is not part of the etcd repo and does not receive the same testing as the rest of etcd. We are testing it in a couple of end-to-end tests. There is only community support here.
- The rollback can be done only from the 3.0.x version (that is using the v3 API) to the 2.2.1 version (that is using the v2 API).
- The tool only works if the data is stored in `application/json` format.
- Rollback doesn't preserve resource versions of objects stored in etcd.

Warning: If the data is not kept in `application/json` format (see Upgrade Procedure), you will lose the option to roll back to etcd 2.2.

The last bullet means that any component or user that has some logic depending on resource versions may require restart after etcd rollback. This includes that all clients using the watch API, which depends on resource versions. Since both the kubelet and kube-proxy use the watch API, a rollback might require restarting all Kubernetes components on all nodes.

Note: At the time of writing, both Kubelet and KubeProxy are using “resource version” only for watching (i.e. are not using resource versions for anything else). And both are using reflector and/or informer frameworks for watching (i.e. they don’t send watch requests themselves). Both those frameworks if they can’t renew watch, they will start from “current version” by doing “list + watch from the resource version returned by list”. That means that if the apiserver will be down for the period of rollback, all of node components should basically restart their watches and start from “now” when apiserver is back. And it will be back with new resource version. That would mean that restarting node components is not needed. But the assumptions here may not hold forever.

Design

This section describes how we are going to do the migration, given the etcd upgrade requirements.

Note that because the code changes in Kubernetes code needed to support the etcd v3 API are local and straightforward, we do not focus on them at all. We focus only on the upgrade/rollback here.

New etcd Docker image

We decided to completely change the content of the etcd image and the way it works. So far, the Docker image for etcd in version X has contained only the etcd and etcdctl binaries.

Going forward, the Docker image for etcd in version X will contain multiple versions of etcd. For example, the 3.0.17 image will contain the 2.2.1, 2.3.7, and 3.0.17 binaries of etcd and etcdctl. This will allow running etcd in multiple different versions using the same Docker image.

Additionally, the image will contain a custom script, written by the Kubernetes team, for doing migration between versions. The image will also contain the rollback tool provided by the etcd team.

Migration script

The migration script that will be part of the etcd Docker image is a bash script that works as follows:

1. Detect which version of etcd we were previously running. For that purpose, we have added a dedicated file, `version.txt`, that holds that information and is stored in the etcd-data-specific directory, next to the etcd data. If the file doesn't exist, we default it to version 2.2.1.
2. If we are in version 2.2.1 and are supposed to upgrade, backup data.
3. Based on the detected previous etcd version and the desired one (communicated via environment variable), do the upgrade steps as needed. This means that for every minor etcd release greater than the detected one and less than or equal to the desired one:
 - (a) Start etcd in that version.
 - (b) Wait until it is healthy. Healthy means that you can write some data to it.
 - (c) Stop this etcd. Note that this etcd will not listen on the default etcd port. It is hard coded to listen on ports that the API server is not configured to connect to, which means that API server won't be able to connect to it. Assuming no other client goes out of its way to try to connect and write to this obscure port, no new data will be written during this period.
4. If the desired API version is v3 and the detected version is v2, do the offline migration from the v2 to v3 data format. For that we use two tools:
 - `./etcdctl migrate`: This is the official tool for migration provided by the etcd team.
 - A custom script that is attaching TTLs to events in the etcd. Note that etcdctl migrate doesn't support TTLs.

- After every successful step, update contents of the version file. This will protect us from the situation where something crashes in the meantime ,and the version file gets completely unsynchronized with the real data. Note that it is safe if the script crashes after the step is done and before the file is updated. This will only result in redoing one step in the next try.

All the previous steps are for the case where the detected version is less than or equal to the desired version. In the opposite case, that is for a rollback, the script works as follows:

- Verify that the detected version is 3.0.x with the v3 API, and the desired version is 2.2.1 with the v2 API. We don't support any other rollback.
- If so, we run the custom tool provided by etcd team to do the offline rollback. This tool reads the v3 formatted data and writes it back to disk in v2 format.
- Finally update the contents of the version file.

Upgrade procedure

Simply modify the command line in the etcd manifest to:

- Run the migration script. If the previously run version is already in the desired version, this will be no-op.
- Start etcd in the desired version.

Starting in Kubernetes version 1.6, this has been done in the manifests for new Google Compute Engine clusters. You should also specify these environment variables. In particular, you must keep `STORAGE_MEDIA_TYPE` set to `application/json` if you wish to preserve the option to roll back.

```
TARGET_STORAGE=etcd3
ETCD_IMAGE=3.0.17
TARGET_VERSION=3.0.17
STORAGE_MEDIA_TYPE=application/json
```

To roll back, use these:

```
TARGET_STORAGE=etcd2
ETCD_IMAGE=3.0.17
TARGET_VERSION=2.2.1
STORAGE_MEDIA_TYPE=application/json
```

Notes for etcd Version 2.2.1

Default configuration

The default setup scripts use kubelet's file-based static pods feature to run etcd in a pod. This manifest should only be run on master VMs. The default location that kubelet scans for manifests is `/etc/kubernetes/manifests/`.

Kubernetes's usage of etcd

By default, Kubernetes objects are stored under the `/registry` key in etcd. This path can be prefixed by using the kube-apiserver flag `--etcd-prefix="/foo"`.

etcd is the only place that Kubernetes keeps state.

Troubleshooting

To test whether etcd is running correctly, you can try writing a value to a test key. On your master VM (or somewhere with firewalls configured such that you can talk to your cluster's etcd), try:

```
curl -X PUT "http://${host}:${port}/v2/keys/_test"
```

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

[Edit This Page](#)

Persistent Volume Claim Protection

FEATURE STATE: Kubernetes v1.9 alpha

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

- The version names contain alpha (e.g. v1alpha1).
- Might be buggy. Enabling the feature may expose bugs. Disabled by default.
- Support for feature may be dropped at any time without notice.
- The API may change in incompatible ways in a later software release without notice.
- Recommended for use only in short-lived testing clusters, due to increased risk of bugs and lack of long-term support.

As of Kubernetes 1.9, persistent volume claims (PVCs) that are in active use by a pod can be protected from pre-mature removal.

- Before you begin
- PVC Protection Verification

Before you begin

- A v1.9 or higher Kubernetes must be installed.
- As PVC Protection is a Kubernetes v1.9 alpha feature it must be enabled:
- Admission controller must be started with the PVC Protection plugin.
- All Kubernetes components must be started with the PVCProtection alpha features enabled.

PVC Protection Verification

The example below uses a GCE PD `StorageClass`, however, similar steps can be performed for any volume type.

Create a `StorageClass` for convenient storage provisioning:

```
apiVersion: storage.k8s.io/v1
kind: StorageClass
metadata:
  name: slow
provisioner: kubernetes.io/gce-pd
parameters:
  type: pd-standard
```

There are two scenarios: a PVC deleted by a user is either in active use or not in active use by a pod.

Scenario 1: The PVC is not in active use by a pod

- Create a PVC:

```
kind: PersistentVolumeClaim
apiVersion: v1
metadata:
  name: slzc
spec:
  accessModes:
    - ReadWriteOnce
  storageClassName: slow
  resources:
    requests:
      storage: 3.7Gi
```

- Check that the PVC has the finalizer `kubernetes.io/pvc-protection` set:

```
$ kubectl describe pvc slzc
Name:           slzc
Namespace:      default
StorageClass:   slow
Status:         Bound
Volume:         pvc-bee8c30a-d6a3-11e7-9af0-42010a800002
Labels:         <none>
Annotations:    pv.kubernetes.io/bind-completed=yes
                  pv.kubernetes.io/bound-by-controller=yes
                  volume.beta.kubernetes.io/storage-provisioner=kubernetes.io/gce-pd
Finalizers:     [kubernetes.io/pvc-protection]
Capacity:       4Gi
Access Modes:   RWO
Events:
Type  Reason          Age   From            Message
----  -----          ----  --              -----
Normal ProvisioningSucceeded  2m    persistentvolume-controller  Successfully provisioned

```

- Delete the PVC and check that the PVC (not in active use by a pod) was removed successfully.

Scenario 2: The PVC is in active use by a pod

- Again, create the same PVC.
- Create a pod that uses the PVC:

```
kind: Pod
apiVersion: v1
metadata:
  name: app1
spec:
  containers:
  - name: test-pod
    image: k8s.gcr.io/busybox:1.24
    command:
    - "/bin/sh"
    args:
    - "-c"
    - "date > /mnt/app1.txt; sleep 60 && exit 0 || exit 1"
  volumeMounts:
  - name: path-pvc
    mountPath: "/mnt"
restartPolicy: "Never"
```

```

volumes:
  - name: path-pvc
    persistentVolumeClaim:
      claimName: slzc

• Wait until the pod status is Running, i.e. the PVC becomes in active use.
• Delete the PVC that is now in active use by a pod and verify that the
  PVC is not removed but its status is Terminating:

Name:          slzc
Namespace:     default
StorageClass:  slow
Status:        Terminating (since Fri, 01 Dec 2017 14:47:55 +0000)
Volume:        pvc-803a1f4d-d6a6-11e7-9af0-42010a800002
Labels:        <none>
Annotations:   pv.kubernetes.io/bind-completed=yes
                pv.kubernetes.io/bound-by-controller=yes
                volume.beta.kubernetes.io/storage-provisioner=kubernetes.io/gce-pd
Finalizers:    [kubernetes.io/pvc-protection]
Capacity:      4Gi
Access Modes:  RWO
Events:
  Type  Reason           Age   From           Message
  ----  -----           ----  ----
  Normal ProvisioningSucceeded  52s   persistentvolume-controller  Successfully provisioned

• Wait until the pod status is Terminated (either delete the pod or wait
  until it finishes). Afterwards, check that the PVC is removed.

```

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

[Edit This Page](#)

Reconfigure a Node's Kubelet in a Live Cluster

FEATURE STATE: Kubernetes v1.10 alpha

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

- The version names contain alpha (e.g. v1alpha1).
- Might be buggy. Enabling the feature may expose bugs. Disabled by default.
- Support for feature may be dropped at any time without notice.
- The API may change in incompatible ways in a later software release without notice.

- Recommended for use only in short-lived testing clusters, due to increased risk of bugs and lack of long-term support.

As of Kubernetes 1.8, the new Dynamic Kubelet Configuration feature is available in alpha. This allows you to change the configuration of Kubelets in a live Kubernetes cluster via first-class Kubernetes concepts. Specifically, this feature allows you to configure individual Nodes' Kubelets via ConfigMaps.

Warning: All Kubelet configuration parameters may be changed dynamically, but not all parameters are safe to change dynamically. This feature is intended for system experts who have a strong understanding of how configuration changes will affect behavior. No documentation currently exists which plainly lists “safe to change” fields, but we plan to add it before this feature graduates from alpha.

- Before you begin
- Reconfiguring the Kubelet on a Live Node in your Cluster
- Kubectl Patch Example
- Understanding KubeletConfigOK Conditions

Before you begin

- A live Kubernetes cluster with both Master and Node at v1.8 or higher must be running, with the `DynamicKubeletConfig` feature gate enabled and the Kubelet's `--dynamic-config-dir` flag set to a writable directory on the Node. This flag must be set to enable Dynamic Kubelet Configuration.
- The `kubectl` command-line tool must be also v1.8 or higher, and must be configured to communicate with the cluster.

Reconfiguring the Kubelet on a Live Node in your Cluster

Basic Workflow Overview

The basic workflow for configuring a Kubelet in a live cluster is as follows:

1. Write a YAML or JSON configuration file containing the Kubelet's configuration.
2. Wrap this file in a ConfigMap and save it to the Kubernetes control plane.
3. Update the Kubelet's corresponding Node object to use this ConfigMap.

Each Kubelet watches a configuration reference on its respective Node object. When this reference changes, the Kubelet downloads the new configuration, updates a local reference to refer to the file, and exits. For the feature to work correctly, you must be running a process manager (like `systemd`) which will

restart the Kubelet when it exits. When the Kubelet is restarted, it will begin using the new configuration.

The new configuration completely overrides configuration provided by `--config`, and is overridden by command-line flags. Unspecified values in the new configuration will receive default values appropriate to the configuration version (e.g. `kubelet.config.k8s.io/v1beta1`), unless overridden by flags.

The status of the Node's Kubelet configuration is reported via the `KubeletConfigOK` condition in the Node status. Once you have updated a Node to use the new ConfigMap, you can observe this condition to confirm that the Node is using the intended configuration. A table describing the possible conditions can be found at the end of this article.

This document describes editing Nodes using `kubectl edit`. There are other ways to modify a Node's spec, including `kubectl patch`, for example, which facilitate scripted workflows.

This document only describes a single Node consuming each ConfigMap. Keep in mind that it is also valid for multiple Nodes to consume the same ConfigMap.

Node Authorizer Workarounds

The Node Authorizer does not yet pay attention to which ConfigMaps are assigned to which Nodes. If you currently use the Node authorizer, your Kubelets will not be automatically granted permission to download their respective ConfigMaps.

The temporary workaround used in this document is to manually create the RBAC Roles and RoleBindings for each ConfigMap. The Node Authorizer will be extended before the Dynamic Kubelet Configuration feature graduates from alpha, so doing this in production should never be necessary.

Generating a file that contains the current configuration

The Dynamic Kubelet Configuration feature allows you to provide an override for the entire configuration object, rather than a per-field overlay. This is a simpler model that makes it easier to trace the source of configuration values and debug issues. The compromise, however, is that you must start with knowledge of the existing configuration to ensure that you only change the fields you intend to change.

In the future, the Kubelet will be bootstrapped from a file on disk (see Set Kubelet parameters via a config file), and you will simply edit a copy of this file (which, as a best practice, should live in version control) while creating the first Kubelet ConfigMap. Today, however, the Kubelet is still bootstrapped with command-line flags. Fortunately, there is a dirty trick you can use to generate a

config file containing a Node's current configuration. The trick involves accessing the Kubelet server's `configz` endpoint via the `kubectl` proxy. This endpoint, in its current implementation, is intended to be used only as a debugging aid, which is part of why this is a dirty trick. The endpoint may be improved in the future, but until then it should not be relied on for production scenarios. This trick also requires the `jq` command to be installed on your machine, for unpacking and editing the JSON response from the endpoint.

Do the following to generate the file:

1. Pick a Node to reconfigure. We will refer to this Node's name as `NODE_NAME`.
2. Start the `kubectl` proxy in the background with `kubectl proxy --port=8001 &`
3. Run the following command to download and unpack the configuration from the `configz` endpoint:

```
$ export NODE_NAME=the-name-of-the-node-you-are-reconfiguring
$ curl -sSL http://localhost:8001/api/v1/nodes/${NODE_NAME}/proxy/configz | jq '.kubeletconfi
```

Note that we have to manually add the `kind` and `apiVersion` to the downloaded object, as these are not reported by the `configz` endpoint. This is one of the limitations of the endpoint.

Edit the configuration file

Using your editor of choice, change one of the parameters in the `kubelet_configz_${NODE_NAME}` file from the previous step. A QPS parameter, `eventRecordQPS` for example, is a good candidate.

Push the configuration file to the control plane

Push the edited configuration file to the control plane with the following command:

```
$ kubectl -n kube-system create configmap my-node-config --from-file=kubelet=kubelet_configz
```

You should see a response similar to:

```
apiVersion: v1
data:
  kubelet: |
    {...}
kind: ConfigMap
metadata:
  creationTimestamp: 2017-09-14T20:23:33Z
  name: my-node-config-gkt4c2m4b2
  namespace: kube-system
```

```
resourceVersion: "119980"
selfLink: /api/v1/namespaces/kube-system/configmaps/my-node-config-gkt4c2m4b2
uid: 946d785e-998a-11e7-a8dd-42010a800006
```

Note that the configuration data must appear under the ConfigMap's `kubelet` key.

We create the ConfigMap in the `kube-system` namespace, which is appropriate because this ConfigMap configures a Kubernetes system component - the Kubelet.

The `--append-hash` option appends a short checksum of the ConfigMap contents to the name. This is convenient for an edit->push workflow, as it will automatically, yet deterministically, generate new names for new ConfigMaps.

We use the `-o yaml` output format so that the name, namespace, and uid are all reported following creation. We will need these in the next step. We will refer to the name as `CONFIG_MAP_NAME` and the uid as `CONFIG_MAP_UID`.

Authorize your Node to read the new ConfigMap

Now that you've created a new ConfigMap, you need to authorize your node to read it. First, create a Role for your new ConfigMap with the following commands:

```
$ export CONFIG_MAP_NAME=name-from-previous-output
$ kubectl -n kube-system create role ${CONFIG_MAP_NAME}-reader --verb=get --resource=configmaps
```

Next, create a RoleBinding to associate your Node with the new Role:

```
$ kubectl -n kube-system create rolebinding ${CONFIG_MAP_NAME}-reader --role=${CONFIG_MAP_NAME}-reader --serviceaccount=kubelet
```

Once the Node Authorizer is updated to do this automatically, you will be able to skip this step.

Set the Node to use the new configuration

Edit the Node's reference to point to the new ConfigMap with the following command:

```
kubectl edit node ${NODE_NAME}
```

Once in your editor, add the following YAML under `spec`:

```
configSource:
  configMapRef:
    name: CONFIG_MAP_NAME
    namespace: kube-system
    uid: CONFIG_MAP_UID
```

Be sure to specify all three of `name`, `namespace`, and `uid`.

Observe that the Node begins using the new configuration

Retrieve the Node with `kubectl get node ${NODE_NAME} -o yaml`, and look for the `KubeletConfigOK` condition in `status.conditions`. You should see the message `Using current (UID: CONFIG_MAP_UID)` when the Kubelet starts using the new configuration.

For convenience, you can use the following command (using `jq`) to filter down to the `KubeletConfigOK` condition:

```
$ kubectl get no ${NODE_NAME} -o json | jq '.status.conditions|map(select(.type=="KubeletConfigOK"))' | jq -r '.[0]'
```

If something goes wrong, you may see one of several different error conditions, detailed in the table of `KubeletConfigOK` conditions, below. When this happens, you should check the Kubelet's log for more details.

Edit the configuration file again

To change the configuration again, we simply repeat the above workflow. Try editing the `kubelet` file, changing the previously changed parameter to a new value.

Push the newly edited configuration to the control plane

Push the new configuration to the control plane in a new ConfigMap with the following command:

```
$ kubectl create configmap my-node-config --namespace=kube-system --from-file=kubelet=kubelet
```

This new ConfigMap will get a new name, as we have changed the contents. We will refer to the new name as `NEW_CONFIG_MAP_NAME` and the new uid as `NEW_CONFIG_MAP_UID`.

Authorize your Node to read the new ConfigMap

Now that you've created a new ConfigMap, you need to authorize your node to read it. First, create a Role for your new ConfigMap with the following commands:

```
$ export NEW_CONFIG_MAP_NAME=name-from-previous-output
$ kubectl -n kube-system create role ${NEW_CONFIG_MAP_NAME}-reader --verb=get --resource=con
```

Next, create a RoleBinding to associate your Node with the new Role:

```
$ kubectl -n kube-system create rolebinding ${NEW_CONFIG_MAP_NAME}-reader --role=${NEW_CONFIG
```

Once the Node Authorizer is updated to do this automatically, you will be able to skip this step.

Configure the Node to use the new configuration

Once more, edit the Node's `spec.configSource` with `kubectl edit node ${NODE_NAME}`. Your new `spec.configSource` should look like the following, with name and uid substituted as necessary:

```
configSource:
  configMapRef:
    name: ${NEW_CONFIG_MAP_NAME}
    namespace: kube-system
    uid: ${NEW_CONFIG_MAP_UID}
```

Observe that the Kubelet is using the new configuration

Once more, retrieve the Node with `kubectl get node ${NODE_NAME} -o yaml`, and look for the `KubeletConfigOK` condition in `status.conditions`. You should see the message `using current: /api/v1/namespaces/kube-system/configmaps/${NEW_CONFIG_MAP_NAME}` when the Kubelet starts using the new configuration.

Deauthorize your Node from reading the old ConfigMap

Once you know your Node is using the new configuration and are confident that the new configuration has not caused any problems, it is a good idea to deauthorize the node from reading the old ConfigMap. Run the following commands to remove the RoleBinding and Role:

```
$ kubectl -n kube-system delete rolebinding ${CONFIG_MAP_NAME}-reader
$ kubectl -n kube-system delete role ${CONFIG_MAP_NAME}-reader
```

Note that this does not necessarily prevent the Node from reverting to the old configuration, as it may locally cache the old ConfigMap for an indefinite period of time.

You may optionally also choose to remove the old ConfigMap:

```
$ kubectl -n kube-system delete configmap ${CONFIG_MAP_NAME}
```

Once the Node Authorizer is updated to do this automatically, you will be able to skip this step.

Reset the Node to use its local default configuration

Finally, if you wish to reset the Node to use the configuration it was provisioned with, simply edit the Node with `kubectl edit node ${NODE_NAME}` and remove the `spec.configSource` subfield.

Observe that the Node is using its local default configuration

After removing this subfield, you should eventually observe that the Kubelet-ConfigOK condition's message reverts to `using current: local`.

Deauthorize your Node from reading the old ConfigMap

Once you know your Node is using the default configuration again, it is a good idea to deauthorize the node from reading the old ConfigMap. Run the following commands to remove the RoleBinding and Role:

```
$ kubectl -n kube-system delete rolebinding ${NEW_CONFIG_MAP_NAME}-reader  
$ kubectl -n kube-system delete role ${NEW_CONFIG_MAP_NAME}-reader
```

Note that this does not necessarily prevent the Node from reverting to the old ConfigMap, as it may locally cache the old ConfigMap for an indefinite period of time.

You may optionally also choose to remove the old ConfigMap:

```
$ kubectl -n kube-system delete configmap ${NEW_CONFIG_MAP_NAME}
```

Once the Node Authorizer is updated to do this automatically, you will be able to skip this step.

Kubectl Patch Example

As mentioned above, there are many ways to change a Node's configSource. Here is an example command that uses `kubectl patch`:

```
kubectl patch node ${NODE_NAME} -p "{\"spec\":{\"configSource\":[\"configMapRef\":{\"name\":"
```

Understanding KubeletConfigOK Conditions

The following table describes several of the `KubeletConfigOK` Node conditions you might encounter in a cluster that has Dynamic Kubelet Config enabled. If you observe a condition with `status=False`, you should check the Kubelet log for more error details by searching for the message or reason text.

Possible Messages	Possible Reasons	Status
using current: local	when the config source is nil, the Kubelet uses its local config	True
using current: /api/v1/namespaces/\${CURRENT_CONFIG_MAP_NAMESPACE}/configmaps/\${CURRENT_CONFIG_MAP_NAME}	passing all checks	True
using last-known-good: local	<ul style="list-style-type: none">failed to load current: /api/v1/namespaces/\${CURRENT_CONFIG_MAP_NAMESPACE}/configmaps/\${CURRENT_CONFIG_MAP_NAME}failed to parse current: /api/v1/namespaces/\${CURRENT_CONFIG_MAP_NAMESPACE}/configmaps/\${CURRENT_CONFIG_MAP_NAME}failed to validate current: /api/v1/namespaces/\${CURRENT_CONFIG_MAP_NAMESPACE}/configmaps/\${CURRENT_CONFIG_MAP_NAME}	False
using last-known-good: /api/v1/namespaces/\${LAST_KNOWN_GOOD_CONFIG_MAP_NAMESPACE}/configmaps/\${LAST_KNOWN_GOOD_CONFIG_MAP_NAME}	<ul style="list-style-type: none">failed to load current: /api/v1/namespaces/\${CURRENT_CONFIG_MAP_NAMESPACE}/configmaps/\${CURRENT_CONFIG_MAP_NAME}failed to parse current: /api/v1/namespaces/\${CURRENT_CONFIG_MAP_NAMESPACE}/configmaps/\${CURRENT_CONFIG_MAP_NAME}failed to validate current: /api/v1/namespaces/\${CURRENT_CONFIG_MAP_NAMESPACE}/configmaps/\${CURRENT_CONFIG_MAP_NAME}	False

Possible Messages	Possible Reasons	Status
<p>The reasons in the next column could potentially appear for any of the above messages.</p> <p>This condition indicates that the Kubelet is having trouble reconciling ‘spec.configSource’, and thus no change to the in-use configuration has occurred.</p> <p>The ”failed to sync” reasons are specific to the failure that occurred, and the next column does not necessarily contain all possible failure reasons.</p>	<p>failed to sync, reason:</p> <ul style="list-style-type: none"> • failed to read Node from informer object cache • failed to reset to local config • invalid NodeConfigSource, exactly one subfield must be non-nil, but all were nil • invalid ObjectReference, all of UID, Name, and Namespace must be specified • invalid Config-Source.ConfigMapRef.UID: \${UID} does not match \${API_PATH}.UID: \${UID_OF_CONFIG_MAP_AT_API_PATH} • failed to determine whether object \${API_PATH} with UID \${UID} was already checkpointed • failed to download ConfigMap with name \${NAME} from namespace \${NAMESPACE} • failed to save config checkpoint for object \${API_PATH} with UID \${UID} • failed to set current config checkpoint to local config • failed to set current config checkpoint to object \${API_PATH} with UID \${UID} 	False

Possible Messages	Possible Reasons	Status
-------------------	------------------	--------

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

[Documentation](#) [Blog](#) [Partners](#) [Community](#) [Case Studies](#)

[twitter](#) [Github](#) [Slack](#)

[Stack Overflow](#) [Forum](#) [Events](#) [Calendar](#)

[Get Kubernetes](#) [Contribute](#)

© 2018 The Kubernetes Authors | Documentation Distributed under CC BY 4.0

Copyright © 2018 The Linux Foundation®. All rights reserved. The Linux Foundation has registered trademarks and uses trademarks. For a list of trademarks of The Linux Foundation, please see our Trademark Usage page

[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

Node Capacity

kube-reserved

system-reserved

eviction-threshold

allocatable
(available for pods)

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]`
- **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.

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=100mi] [,] [memory=100Mi] [,] [ephemeral-storage=1Gi]`
- **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).

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.

Eviction Thresholds

- **Kubelet Flag:** `--eviction-hard=[memory.available<500Mi]`

Memory pressure at the node level leads to System OOMs which affects the entire node and all pods running on it. Nodes can go offline temporarily until memory has been reclaimed. To avoid (or reduce the probability of) system OOMs kubelet provides Out of Resource management. Evictions are supported for `memory` and `ephemeral-storage` only. By reserving some memory via `--eviction-hard` flag, the kubelet attempts to evict pods whenever memory availability on the node drops below the reserved value. Hypothetically, if system daemons did not exist on a node, pods cannot use more than `capacity - eviction-hard`. For this reason, resources reserved for evictions are not available for pods.

Enforcing Node Allocatable

- **Kubelet Flag:** `--enforce-node-allocatable=pods[,,][system-reserved][,,][kube-reserved]`

The scheduler treats `Allocatable` as the available `capacity` for pods.

`kubelet` enforce `Allocatable` across pods by default. Enforcement is performed by evicting pods whenever the overall usage across all pods exceeds `Allocatable`. More details on eviction policy can be found here. This enforcement is controlled by specifying `pods` value to the kubelet flag `--enforce-node-allocatable`.

Optionally, `kubelet` can be made to enforce `kube-reserved` and `system-reserved` by specifying `kube-reserved` & `system-reserved` values in the same flag. Note that to enforce `kube-reserved` or `system-reserved`, `--kube-reserved-cgroup` or `--system-reserved-cgroup` needs to be specified respectively.

General Guidelines

System daemons are expected to be treated similar to `Guaranteed` pods. System daemons can burst within their bounding control groups and this behavior

needs to be managed as part of kubernetes deployments. For example, `kubelet` should have its own control group and share `Kube-reserved` resources with the container runtime. However, Kubelet cannot burst and use up all available Node resources if `kube-reserved` is enforced.

Be extra careful while enforcing `system-reserved` reservation since it can lead to critical system services being CPU starved or OOM killed on the node. The recommendation is to enforce `system-reserved` only if a user has profiled their nodes exhaustively to come up with precise estimates and is confident in their ability to recover if any process in that group is `oom_killed`.

- To begin with enforce `Allocatable` on pods.
- Once adequate monitoring and alerting is in place to track kube system daemons, attempt to enforce `kube-reserved` based on usage heuristics.
- If absolutely necessary, enforce `system-reserved` over time.

The resource requirements of kube system daemons may grow over time as more and more features are added. Over time, kubernetes project will attempt to bring down utilization of node system daemons, but that is not a priority as of now. So expect a drop in `Allocatable` capacity in future releases.

Example Scenario

Here is an example to illustrate Node Allocatable computation:

- Node has 32Gi of `memory`, 16 CPUs and 100Gi of `Storage`
- `--kube-reserved` is set to `cpu=1,memory=2Gi,ephemeral-storage=1Gi`
- `--system-reserved` is set to `cpu=500m,memory=1Gi,ephemeral-storage=1Gi`
- `--eviction-hard` is set to `memory.available<500Mi,nodefs.available<10%`

Under this scenario, `Allocatable` will be 14.5 CPUs, 28.5Gi of memory and 98Gi of local storage. Scheduler ensures that the total memory `requests` across all pods on this node does not exceed 28.5Gi and storage doesn't exceed 88Gi. Kubelet evicts pods whenever the overall memory usage across pods exceeds 28.5Gi, or if overall disk usage exceeds 88Gi. If all processes on the node consume as much CPU as they can, pods together cannot consume more than 14.5 CPUs.

If `kube-reserved` and/or `system-reserved` is not enforced and system daemons exceed their reservation, `kubelet` evicts pods whenever the overall node memory usage is higher than 31.5Gi or `storage` is greater than 90Gi

Feature Availability

As of Kubernetes version 1.2, it has been possible to **optionally** specify `kube-reserved` and `system-reserved` reservations. The scheduler switched to using `Allocatable` instead of `Capacity` when available in the same release.

As of Kubernetes version 1.6, `eviction-thresholds` are being considered by computing `Allocatable`. To revert to the old behavior set `--experimental-allocatable-ignore-eviction` kubelet flag to `true`.

As of Kubernetes version 1.6, kubelet enforces `Allocatable` on pods using control groups. To revert to the old behavior unset `--enforce-node-allocatable` kubelet flag. Note that unless `--kube-reserved`, or `--system-reserved` or `--eviction-hard` flags have non-default values, `Allocatable` enforcement does not affect existing deployments.

As of Kubernetes version 1.6, kubelet launches pods in their own cgroup sandbox in a dedicated part of the cgroup hierarchy it manages. Operators are required to drain their nodes prior to upgrade of the kubelet from prior versions in order to ensure pods and their associated containers are launched in the proper part of the cgroup hierarchy.

As of Kubernetes version 1.7, kubelet supports specifying `storage` as a resource for `kube-reserved` and `system-reserved`.

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

[Edit This Page](#)

Safely Drain a Node while Respecting Application SLOs

This page shows how to safely drain a machine, respecting the application-level disruption SLOs you have specified using PodDisruptionBudget.

- 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 ≥ 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 without violating any application-level disruption SLOs). It is then safe to bring down the node by powering down its physical machine or, if running on a cloud platform, deleting its virtual machine.

First, identify the name of the node you wish to drain. You can list all of the nodes in your cluster with

```
kubectl get nodes
```

Next, tell Kubernetes to drain the node:

```
kubectl drain <node name>
```

Once it returns (without giving an error), you can power down the node (or equivalently, if on a cloud platform, delete the virtual machine backing the node). If you leave the node in the cluster during the maintenance operation, you need to run

```
kubectl uncordon <node name>
```

afterwards to tell Kubernetes that it can resume scheduling new pods onto the node.

Draining multiple nodes in parallel

The `kubectl drain` command should only be issued to a single node at a time. However, you can run multiple `kubectl drain` commands for different nodes in parallel, in different terminals or in the background. Multiple drain commands running concurrently will still respect the `PodDisruptionBudget` you specify.

For example, if you have a StatefulSet with three replicas and have set a `PodDisruptionBudget` for that set specifying `minAvailable: 2`. `kubectl drain` 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
```

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.

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

[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 Level 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.

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 1.7 contains encryption at rest, an alpha feature that 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 experimental, it may offer 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.

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

[Edit This Page](#)

Set Kubelet parameters via a config file

FEATURE STATE: Kubernetes v1.10 beta

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

- The version names contain beta (e.g. v2beta3).
- Code is well tested. Enabling the feature is considered safe. Enabled by default.
- Support for the overall feature will not be dropped, though details may change.
- The schema and/or semantics of objects may change in incompatible ways in a subsequent beta or stable release. When this happens, we will provide instructions for migrating to the next version. This may require deleting, editing, and re-creating API objects. The editing process may require some thought. This may require downtime for applications that rely on the feature.
- Recommended for only non-business-critical uses because of potential for incompatible changes in subsequent releases. If you have multiple clusters that can be upgraded independently, you may be able to relax this restriction.
- **Please do try our beta features and give feedback on them! After they exit beta, it may not be practical for us to make more changes.**

A subset of the Kubelet's configuration parameters may be set via an on-disk config file, as a substitute for command-line flags. This functionality is considered beta in v1.10.

Providing parameters via a config file is the recommended approach because it simplifies node deployment and configuration management.

- Before you begin
- Create the config file
- Start a Kubelet process configured via the config file
- Relationship to Dynamic Kubelet Config

Before you begin

- A v1.10 or higher Kubelet binary must be installed for beta functionality.

Create the config file

The subset of the Kubelet's configuration that can be configured via a file is defined by the `KubeletConfiguration` struct here (v1beta1).

The configuration file must be a JSON or YAML representation of the parameters in this struct. Make sure the Kubelet has read permissions on the file.

Here is an example of what this file might look like:

```
kind: KubeletConfiguration
apiVersion: kubelet.config.k8s.io/v1beta1
evictionHard:
  memory.available: "200Mi"
```

In the example, the Kubelet is configured to evict Pods when available memory drops below 200Mi. All other Kubelet configuration values are left at their built-in defaults, unless overridden by flags. Command line flags which target the same value as a config file will override that value.

For a trick to generate a configuration file from a live node, see Reconfigure a Node's Kubelet in a Live Cluster.

Start a Kubelet process configured via the config file

Start the Kubelet with the `--config` flag set to the path of the Kubelet's config file. The Kubelet will then load its config from this file.

Note that command line flags which target the same value as a config file will override that value. This helps ensure backwards compatibility with the command-line API.

Note that relative file paths in the Kubelet config file are resolved relative to the location of the Kubelet config file, whereas relative paths in command line flags are resolved relative to the Kubelet's current working directory.

Note that some default values differ between command-line flags and the Kubelet config file. If `--config` is provided and the values are not specified via the command line, the defaults for the `KubeletConfiguration` version apply. In the above example, this version is `kubelet.config.k8s.io/v1beta1`.

Relationship to Dynamic Kubelet Config

If you are using the Dynamic Kubelet Configuration feature, the combination of configuration provided via `--config` and any flags which override these values is considered the default “last known good” configuration by the automatic rollback mechanism.

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

[Edit This Page](#)

Set up High-Availability Kubernetes Masters

FEATURE STATE: Kubernetes 1.5 alpha

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

- The version names contain alpha (e.g. v1alpha1).
- Might be buggy. Enabling the feature may expose bugs. Disabled by default.
- Support for feature may be dropped at any time without notice.
- The API may change in incompatible ways in a later software release without notice.
- Recommended for use only in short-lived testing clusters, due to increased risk of bugs and lack of long-term support.

You can replicate Kubernetes masters in `kube-up` or `kube-down` scripts for Google Compute Engine. This document describes how to use `kube-up/down` scripts to manage highly available (HA) masters and how HA masters are implemented for use with GCE.

- Before you begin
- Starting an HA-compatible cluster
- Adding a new master replica
- Removing a master replica
- Handling master replica failures
- Best practices for replicating masters for HA clusters
- Implementation notes
- Additional reading

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`.

Starting an HA-compatible cluster

To create a new HA-compatible cluster, you must set the following flags in your `kube-up` script:

- `MULTIZONE=true` - to prevent removal of master replicas kubelets from zones different than server's default zone. Required if you want to run master replicas in different zones, which is recommended.
- `ENABLE_ETCD_QUORUM_READS=true` - to ensure that reads from all API servers will return most up-to-date data. If true, reads will be directed to leader etcd replica. Setting this value to true is optional: reads will be more reliable but will also be slower.

Optionally, you can specify a GCE zone where the first master replica is to be created. Set the following flag:

- `KUBE_GCE_ZONE=zone` - zone where the first master replica will run.

The following sample command sets up a HA-compatible cluster in the GCE zone `europe-west1-b`:

```
$ MULTIZONE=true KUBE_GCE_ZONE=europe-west1-b ENABLE_ETCD_QUORUM_READS=true ./cluster/kube-
```

Note that the commands above create a cluster with one master; however, you can add new master replicas to the cluster with subsequent commands.

Adding a new master replica

After you have created an HA-compatible cluster, you can add master replicas to it. You add master replicas by using a `kube-up` script with the following flags:

- `KUBE_REPLICATE_EXISTING_MASTER=true` - to create a replica of an existing master.
- `KUBE_GCE_ZONE=zone` - zone where the master replica will run. Must be in the same region as other replicas' zones.

You don't need to set the `MULTIZONE` or `ENABLE_ETCD_QUORUM_READS` flags, as those are inherited from when you started your HA-compatible cluster.

The following sample command replicates the master on an existing HA-compatible cluster:

```
$ KUBE_GCE_ZONE=europe-west1-c KUBE_REPLICATE_EXISTING_MASTER=true ./cluster/kube-up.sh
```

Removing a master replica

You can remove a master replica from an HA cluster by using a `kube-down` script with the following flags:

- `KUBE_DELETE_NODES=false` - to restrain deletion of kubelets.
- `KUBE_GCE_ZONE=zone` - the zone from where master replica will be removed.
- `KUBE_REPLICA_NAME=replica_name` - (optional) the name of master replica to remove. If empty: any replica from the given zone will be removed.

The following sample command removes a master replica from an existing HA cluster:

```
$ KUBE_DELETE_NODES=false KUBE_GCE_ZONE=europe-west1-c ./cluster/kube-down.sh
```

Handling master replica failures

If one of the master replicas in your HA cluster fails, the best practice is to remove the replica from your cluster and add a new replica in the same zone. The following sample commands demonstrate this process:

1. Remove the broken replica:

```
$ KUBE_DELETE_NODES=false KUBE_GCE_ZONE=replica_zone KUBE_REPLICA_NAME=replica_name ./cluster/kube-down.sh
```

2. Add a new replica in place of the old one:

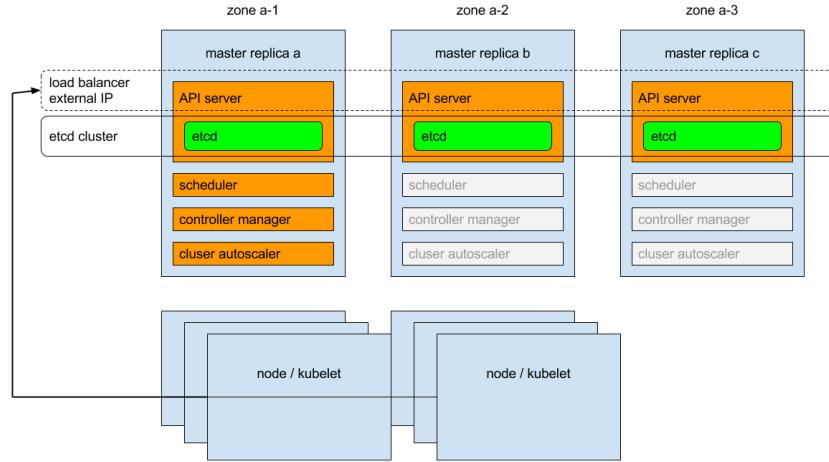
```
$ KUBE_GCE_ZONE=replica-zone KUBE_REPLICATE_EXISTING_MASTER=true ./cluster/kube-up.sh
```

Best practices for replicating masters for HA clusters

- Try to place master replicas in different zones. During a zone failure, all masters placed inside the zone will fail. To survive zone failure, also place nodes in multiple zones (see [multiple-zones](#) for details).
- Do not use a cluster with two master replicas. Consensus on a two-replica cluster requires both replicas running when changing persistent state. As a result, both replicas are needed and a failure of any replica turns cluster into majority failure state. A two-replica cluster is thus inferior, in terms of HA, to a single replica cluster.
- When you add a master replica, cluster state (`etcd`) is copied to a new instance. If the cluster is large, it may take a long time to duplicate its state. This operation may be sped up by migrating `etcd` data directory,

as described here (we are considering adding support for etcd data dir migration in future).

Implementation notes



Overview

Each of master replicas will run the following components in the following mode:

- etcd instance: all instances will be clustered together using consensus;
- API server: each server will talk to local etcd - all API servers in the cluster will be available;
- controllers, scheduler, and cluster auto-scaler: will use lease mechanism - only one instance of each of them will be active in the cluster;
- add-on manager: each manager will work independently trying to keep add-ons in sync.

In addition, there will be a load balancer in front of API servers that will route external and internal traffic to them.

Load balancing

When starting the second master replica, a load balancer containing the two replicas will be created and the IP address of the first replica will be promoted to IP address of load balancer. Similarly, after removal of the penultimate master replica, the load balancer will be removed and its IP address will be assigned to the last remaining replica. Please note that creation and removal of load balancer are complex operations and it may take some time (~20 minutes) for them to propagate.

Master service & kubelets

Instead of trying to keep an up-to-date list of Kubernetes apiserver in the Kubernetes service, the system directs all traffic to the external IP:

- in one master cluster the IP points to the single master,
- in multi-master cluster the IP points to the load balancer in-front of the masters.

Similarly, the external IP will be used by kubelets to communicate with master.

Master certificates

Kubernetes generates Master TLS certificates for the external public IP and local IP for each replica. There are no certificates for the ephemeral public IP for replicas; to access a replica via its ephemeral public IP, you must skip TLS verification.

Clustering etcd

To allow etcd clustering, ports needed to communicate between etcd instances will be opened (for inside cluster communication). To make such deployment secure, communication between etcd instances is authorized using SSL.

Additional reading

[Automated HA master deployment - design doc](#)

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

[Edit This Page](#)

Set up a Highly Available etcd Cluster With kubeadm

Kubeadm defaults to running a single member etcd cluster in a static pod managed by the kubelet on the control plane node. This is not a highly available setup as the the etcd cluster contains only one member and cannot sustain any members becoming unavailable. This task walks through the process of creating a highly available etcd cluster of three members that can be used as an external etcd when using kubeadm to set up a kubernetes cluster.

- Before you begin
- Setting up the cluster
- What's next

Before you begin

- Three hosts that can talk to each other over ports 2379 and 2380. This document assumes these default ports. However, they are configurable through the kubeadm config file.
- Each host must have docker, kubelet, and kubeadm installed.
- Some infrastructure to copy files between hosts (e.g., ssh).

Setting up the cluster

The general approach is to generate all certs on one node and only distribute the *necessary* files to the other nodes. Note that kubeadm contains all the necessary cryptographic machinery to generate the certificates described below; no other cryptographic tooling is required for this example.

1. Create configuration files for kubeadm.

Generate one kubeadm configuration file for each host that will have an etcd member running on it using the following script.

```
# Update HOST0, HOST1, and HOST2 with the IPs or resolvable names of your hosts
export HOST0=10.0.0.6
export HOST1=10.0.0.7
export HOST2=10.0.0.8

# Create temp directories to store files that will end up on other hosts.
mkdir -p /tmp/${HOST0}/ /tmp/${HOST1}/ /tmp/${HOST2}/

ETCDHOSTS=(${HOST0} ${HOST1} ${HOST2})
NAMES=("infra0" "infra1" "infra2")
```

```

for i in "${!ETCDHOSTS[@]}"; do
HOST=${ETCDHOSTS[$i]}
NAME=${NAMES[$i]}
cat << EOF > /tmp/${HOST}/kubeadmcf.yaml
apiVersion: "kubeadm.k8s.io/v1alpha2"
kind: MasterConfiguration
etcd:
  localEtcd:
    serverCertSANs:
      - "${HOST}"
    peerCertSANs:
      - "${HOST}"
    extraArgs:
      initial-cluster: infra0=https://$HOST:2380,infra1=https://$HOST
      initial-cluster-state: new
      name: ${NAME}
      listen-peer-urls: https://${HOST}:2380
      listen-client-urls: https://${HOST}:2379
      advertise-client-urls: https://${HOST}:2379
      initial-advertise-peer-urls: https://${HOST}:2380
EOF
done

```

2. Generate the certificate authority

If you already have a CA then the only action that is copying the CA's crt and key file to `/etc/kubernetes/pki/etcd/ca.crt` and `/etc/kubernetes/pki/etcd/ca.key`. After those files have been copied, please skip this step.

If you do not already have a CA then run this command on `$HOST0` (where you generated the configuration files for kubeadm).

```
kubeadm alpha phase certs etcd-ca
```

This creates two files

- `/etc/kubernetes/pki/etcd/ca.crt`
- `/etc/kubernetes/pki/etcd/ca.key`

3. Create certificates for each member

```

kubeadm alpha phase certs etcd-server --config=/tmp/${HOST2}/kubeadmcf.yaml
kubeadm alpha phase certs etcd-peer --config=/tmp/${HOST2}/kubeadmcf.yaml
kubeadm alpha phase certs etcd-healthcheck-client --config=/tmp/${HOST2}/kubeadmcf.yaml
kubeadm alpha phase certs apiserver-etcd-client --config=/tmp/${HOST2}/kubeadmcf.yaml
cp -R /etc/kubernetes/pki /tmp/${HOST2}/
# cleanup non-reusable certificates
find /etc/kubernetes/pki -not -name ca.crt -not -name ca.key -type f -delete

```

```

kubeadm alpha phase certs etcd-server --config=/tmp/${HOST1}/kubeadmcf.cfg.yaml
kubeadm alpha phase certs etcd-peer --config=/tmp/${HOST1}/kubeadmcf.cfg.yaml
kubeadm alpha phase certs etcd-healthcheck-client --config=/tmp/${HOST1}/kubeadmcf.cfg.yaml
kubeadm alpha phase certs apiserver-etcd-client --config=/tmp/${HOST1}/kubeadmcf.cfg.yaml
cp -R /etc/kubernetes/pki /tmp/${HOST2}/
find /etc/kubernetes/pki -not -name ca.crt -not -name ca.key -type f -delete

kubeadm alpha phase certs etcd-server --config=/tmp/${HOST0}/kubeadmcf.cfg.yaml
kubeadm alpha phase certs etcd-peer --config=/tmp/${HOST0}/kubeadmcf.cfg.yaml
kubeadm alpha phase certs etcd-healthcheck-client --config=/tmp/${HOST0}/kubeadmcf.cfg.yaml
kubeadm alpha phase certs apiserver-etcd-client --config=/tmp/${HOST0}/kubeadmcf.cfg.yaml
# No need to move the certs because they are for HOST0

# clean up certs that should not be copied off this host
find /tmp/${HOST2} -name ca.key -type f -delete
find /tmp/${HOST1} -name ca.key -type f -delete

```

4. Copy certificates and kubeadm configs

The certificates have been generated and now they must be moved to their respective hosts.

```

USER=ubuntu
HOST=${HOST1}
scp -r /tmp/${HOST}/* ${USER}@${HOST}:
ssh ${USER}@${HOST}
USER@HOST $ sudo -Es
root@HOST $ chown -R root:root pki
root@HOST $ mv pki /etc/kubernetes/

```

5. Ensure all expected files exist

The complete list of required files on \$HOST0 is:

```

/tmp/${HOST0}
  kubeadmcf.cfg.yaml
---
/etc/kubernetes/pki
  apiserver-etcd-client.crt
  apiserver-etcd-client.key
  etcd
    ca.crt
    ca.key
    healthcheck-client.crt
    healthcheck-client.key
    peer.crt
    peer.key
    server.crt
    server.key

```

On \$HOST1:

```
$HOME
  kubeadmcfg.yaml
---
/etc/kubernetes/pki
  apiserver-etcd-client.crt
  apiserver-etcd-client.key
  etcd
    ca.crt
    healthcheck-client.crt
    healthcheck-client.key
    peer.crt
    peer.key
    server.crt
    server.key
```

On \$HOST2

```
$HOME
  kubeadmcfg.yaml
---
/etc/kubernetes/pki
  apiserver-etcd-client.crt
  apiserver-etcd-client.key
  etcd
    ca.crt
    healthcheck-client.crt
    healthcheck-client.key
    peer.crt
    peer.key
    server.crt
    server.key
```

6. Create the static pod manifests

Now that the certificates and configs are in place it's time to create the manifests. On each host run the `kubeadm` command to generate a static manifest for etcd.

```
root@HOST0 $ kubeadm alpha phase etcd local --config=/tmp/${HOST0}/kubeadmcfg.yaml
root@HOST1 $ kubeadm alpha phase etcd local --config=/home/ubuntu/kubeadmcfg.yaml
root@HOST2 $ kubeadm alpha phase etcd local --config=/home/ubuntu/kubeadmcfg.yaml
```

7. Optional: Check the cluster health

```
docker run --rm -it \
--net host \
-v /etc/kubernetes:/etc/kubernetes quay.io/coreos/etcd:v3.2.18 etcdctl \
--cert-file /etc/kubernetes/pki/etcd/peer.crt \
```

```
--key-file /etc/kubernetes/pki/etcd/peer.key \
--ca-file /etc/kubernetes/pki/etcd/ca.crt \
--endpoints https://${HOST0}:2379 cluster-health
...
cluster is healthy
```

What's next

Once you have a working 3 member etcd cluster, you can continue setting up a highly available control plane using the external etcd method with kubeadm.

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

[Edit This Page](#)

Share a Cluster with Namespaces

This page shows how to view, work in, and delete namespaces. The page also shows how to use Kubernetes namespaces to subdivide your cluster.

- Before you begin
- 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
```

Kubernetes starts with two initial namespaces:

- **default** The default namespace for objects with no other namespace
- **kube-system** The namespace for objects created by the Kubernetes system

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
```

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

1. 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 namespace's by doing the following:

```
$ kubectl get namespaces
NAME      STATUS    AGE
default   Active   13m
```

1. 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.

Use the file `namespace-dev.json` which describes a development namespace:

```
namespace-dev.json docs/tasks/administer-cluster
{
    "kind": "Namespace",
    "apiVersion": "v1",
    "metadata": {
        "name": "development",
        "labels": {
            "name": "development"
        }
    }
}
```

Create the development namespace using kubectl.

```
$ kubectl create -f docs/tasks/administer-cluster/namespace-dev.json
```

And then let's create the production namespace using kubectl.

```
$ kubectl create -f docs/tasks/administer-cluster/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
```

1. 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
$ kubectl config set-context prod --namespace=production --cluster=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=kubernetes/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      DESIRED   CURRENT   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=kubernetes/serve_hostname --replicas=5  
  
$ kubectl get deployment  
NAME      DESIRED   CURRENT   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.

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

[Edit This Page](#)

Static Pods

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!

Static pods are managed directly by kubelet daemon on a specific node, without the API server observing it. It does not have an associated replication controller, and kubelet daemon itself watches it and restarts it when it crashes. There is no health check. Static pods are always bound to one kubelet daemon and always run on the same node with it.

Kubelet automatically tries to create a *mirror pod* on the Kubernetes API server for each static pod. This means that the pods are visible on the API server but cannot be controlled from there.

- Static pod creation
- Behavior of static pods
- Dynamic addition and removal of static pods

Static pod creation

Static pod can be created in two ways: either by using configuration file(s) or by HTTP.

Configuration files

The configuration files are just standard pod definitions in json or yaml format in a specific directory. Use `kubelet --pod-manifest-path=<the directory>` to start kubelet daemon, which periodically scans the directory and creates/deletes static pods as yaml/json files appear/disappear there. Note that 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 we want to run the static pod. In this example, it's `my-node1`.

```
[joe@host ~] $ 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`:

```
[root@my-node1 ~] $ mkdir /etc/kubelet.d/
[root@my-node1 ~] $ 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
```

- Configure your kubelet daemon 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/kubernetes/pods"
```

Instructions for other distributions or Kubernetes installations may vary.

- Restart kubelet. On Fedora, this is:

```
[root@my-node1 ~] $ systemctl restart kubelet
```

Pods created via HTTP

Kubelet periodically downloads a file specified by `--manifest-url=<URL>` argument and interprets it as a json/yaml file with a pod definition. It works the same as `--pod-manifest-path=<directory>`, i.e. it's reloaded every now and then and changes are applied to running static pods (see below).

Behavior of static pods

When kubelet starts, it automatically starts all pods defined in directory specified in `--pod-manifest-path=` or `--manifest-url=` arguments, i.e. our static-web. (It may take some time to pull nginx image, be patient...):

```
[joe@my-node1 ~] $ docker ps
CONTAINER ID IMAGE COMMAND CREATED STATUS PORTS NAMES
f6d05272b57e nginx:latest "nginx" 8 minutes ago Up 8 minutes k8s_web.6f802af4
```

If we look at our Kubernetes API server (running on host `my-master`), we see that a new mirror-pod was created there too:

```
[joe@host ~] $ ssh my-master
[joe@my-master ~] $ kubectl get pods
NAME READY STATUS RESTARTS AGE
static-web-my-node1 1/1 Running 0 2m
```

Labels from the static pod are propagated into the mirror-pod and can be used as usual for filtering.

Notice we cannot delete the pod with the API server (e.g. via `kubectl` command), kubelet simply won't remove it.

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`(/docs/concepts/policy/pod-security-policy/).

```
[joe@my-master ~] $ kubectl delete pod static-web-my-node1
pod "static-web-my-node1" deleted
[joe@my-master ~] $ kubectl get pods
NAME           READY   STATUS    RESTARTS   AGE
static-web-my-node1   1/1     Running   0          12s
```

Back to our `my-node1` host, we can try to stop the container manually and see, that kubelet automatically restarts it in a while:

```
[joe@host ~] $ ssh my-node1
[joe@my-node1 ~] $ docker stop f6d05272b57e
[joe@my-node1 ~] $ sleep 20
[joe@my-node1 ~] $ docker ps
CONTAINER ID        IMAGE       COMMAND       CREATED      ...
5b920cbaf8b1        nginx:latest "nginx -g 'daemon of ...

```

Dynamic addition and removal of static pods

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.

```
[joe@my-node1 ~] $ mv /etc/kubelet.d/static-web.yaml /tmp
[joe@my-node1 ~] $ sleep 20
[joe@my-node1 ~] $ docker ps
// no nginx container is running
[joe@my-node1 ~] $ mv /tmp/static-web.yaml /etc/kubelet.d/
[joe@my-node1 ~] $ sleep 20
[joe@my-node1 ~] $ docker ps
CONTAINER ID        IMAGE       COMMAND       CREATED      ...
e7a62e3427f1        nginx:latest "nginx -g 'daemon of ...

```

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

[Edit This Page](#)

Storage Object in Use Protection

FEATURE STATE: Kubernetes v1.10 beta

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

- The version names contain beta (e.g. v2beta3).
- Code is well tested. Enabling the feature is considered safe. Enabled by default.

- Support for the overall feature will not be dropped, though details may change.
- The schema and/or semantics of objects may change in incompatible ways in a subsequent beta or stable release. When this happens, we will provide instructions for migrating to the next version. This may require deleting, editing, and re-creating API objects. The editing process may require some thought. This may require downtime for applications that rely on the feature.
- Recommended for only non-business-critical uses because of potential for incompatible changes in subsequent releases. If you have multiple clusters that can be upgraded independently, you may be able to relax this restriction.
- **Please do try our beta features and give feedback on them! After they exit beta, it may not be practical for us to make more changes.**

Persistent volume claims (PVCs) that are in active use by a pod and persistent volumes (PVs) that are bound to PVCs can be protected from pre-mature removal.

- Before you begin
- Storage Object in Use Protection feature used for PVC Protection
- Storage Object in Use Protection feature used for PV Protection

Before you begin

- The Storage Object in Use Protection feature is enabled in a version of Kubernetes in which it is supported.

Storage Object in Use Protection feature used for PVC Protection

The example below uses a GCE PD `StorageClass`, however, similar steps can be performed for any volume type.

Create a `StorageClass` for convenient storage provisioning:

```
apiVersion: storage.k8s.io/v1
kind: StorageClass
metadata:
  name: slow
provisioner: kubernetes.io/gce-pd
parameters:
  type: pd-standard
```

Verification scenarios follow below.

Scenario 1: The PVC is not in active use by a pod

- Create a PVC:

```
kind: PersistentVolumeClaim
apiVersion: v1
metadata:
  name: slzc
spec:
  accessModes:
    - ReadWriteOnce
  storageClassName: slow
  resources:
    requests:
      storage: 3.7Gi
```

- Check that the PVC has the finalizer `kubernetes.io/pvc-protection` set:

```
kubectl describe pvc slzc
Name:           slzc
Namespace:      default
StorageClass:   slow
Status:         Bound
Volume:         pvc-bee8c30a-d6a3-11e7-9af0-42010a800002
Labels:         <none>
Annotations:    pv.kubernetes.io/bind-completed=yes
                  pv.kubernetes.io/bound-by-controller=yes
                  volume.beta.kubernetes.io/storage-provisioner=kubernetes.io/gce-pd
Finalizers:     [kubernetes.io/pvc-protection]
Capacity:       4Gi
Access Modes:   RWO
Events:
  Type      Reason          Age      From            Message
  ----      ----          ----      ----            -----
  Normal    ProvisioningSucceeded  2m      persistentvolume-controller  Successfully provisioned
```

- Delete the PVC and check that the PVC (not in active use by a pod) was removed successfully.

Scenario 2: The PVC is in active use by a pod

- Again, create the same PVC.
- Create a pod that uses the PVC:

```
kind: Pod
apiVersion: v1
```

```

metadata:
  name: app1
spec:
  containers:
  - name: test-pod
    image: k8s.gcr.io/busybox:1.24
    command:
      - "/bin/sh"
    args:
      - "-c"
      - "date > /mnt/app1.txt; sleep 60 && exit 0 || exit 1"
  volumeMounts:
    - name: path-pvc
      mountPath: "/mnt"
  restartPolicy: "Never"
  volumes:
    - name: path-pvc
      persistentVolumeClaim:
        claimName: slzc

```

- Wait until the pod status is **Running**, i.e. the PVC becomes in active use.
- Delete the PVC that is now in active use by a pod and verify that the PVC is not removed but its status is **Terminating**:

Name:	slzc			
Namespace:	default			
StorageClass:	slow			
Status:	Terminating (since Fri, 01 Dec 2017 14:47:55 +0000)			
Volume:	pvc-803a1f4d-d6a6-11e7-9af0-42010a800002			
Labels:	<none>			
Annotations:	pv.kubernetes.io/bind-completed=yes pv.kubernetes.io/bound-by-controller=yes volume.beta.kubernetes.io/storage-provisioner=kubernetes.io/gce-pd			
Finalizers:	[kubernetes.io/pvc-protection]			
Capacity:	4Gi			
Access Modes:	RWO			
Events:				
Type	Reason	Age	From	Message
----	-----	----	----	-----
Normal	ProvisioningSucceeded	52s	persistentvolume-controller	Successfully provisioned

- Wait until the pod status is **Terminated** (either delete the pod or wait until it finishes). Afterwards, check that the PVC is removed.

Scenario 3: A pod starts using a PVC that is in Terminating state

- Again, create the same PVC.
- Create a first pod that uses the PVC:

```
kind: Pod
apiVersion: v1
metadata:
  name: app1
spec:
  containers:
    - name: test-pod
      image: k8s.gcr.io/busybox:1.24
      command:
        - "/bin/sh"
      args:
        - "-c"
        - "date > /mnt/app1.txt; sleep 600 && exit 0 || exit 1"
    volumeMounts:
      - name: path-pvc
        mountPath: "/mnt"
  restartPolicy: "Never"
  volumes:
    - name: path-pvc
      persistentVolumeClaim:
        claimName: slzc
```

- Wait until the pod status is `Running`, i.e. the PVC becomes in active use.
- Delete the PVC that is now in active use by a pod and verify that the PVC is not removed but its status is `Terminating`:

```
Name:           slzc
Namespace:      default
StorageClass:   slow
Status:         Terminating (since Fri, 01 Dec 2017 14:47:55 +0000)
Volume:         pvc-803a1f4d-d6a6-11e7-9af0-42010a800002
Labels:         <none>
Annotations:    pv.kubernetes.io/bind-completed=yes
                  pv.kubernetes.io/bound-by-controller=yes
                  volume.beta.kubernetes.io/storage-provisioner=kubernetes.io/gce-pd
Finalizers:     [kubernetes.io/pvc-protection]
Capacity:       4Gi
Access Modes:   RWO
Events:
  Type  Reason          Age    From            Message
  ----  -----          ----   ----
  Normal ProvisioningSucceeded  52s   persistentvolume-controller  Successfully provisioned
```

- Create a second pod that uses the same PVC:

```

kind: Pod
apiVersion: v1
metadata:
  name: app2
spec:
  containers:
  - name: test-pod
    image: gcr.io/google_containers/busybox:1.24
    command:
      - "/bin/sh"
    args:
      - "-c"
      - "date > /mnt/app1.txt; sleep 600 && exit 0 || exit 1"
  volumeMounts:
  - name: path-pvc
    mountPath: "/mnt"
  restartPolicy: "Never"
  volumes:
  - name: path-pvc
    persistentVolumeClaim:
      claimName: slzc

```

- Verify that the scheduling of the second pod fails with the below warning:

Warning FailedScheduling 18s (x4 over 21s) default-scheduler persistentvolumeclaim "slzc"

- Wait until the pod status of both pods is Terminated or Completed (either delete the pods or wait until they finish). Afterwards, check that the PVC is removed.

Storage Object in Use Protection feature used for PV Protection

The example below uses a HostPath PV.

Verification scenarios follow below.

Scenario 1: The PV is not bound to a PVC

- Create a PV:

```

kind: PersistentVolume
apiVersion: v1
metadata:
  name: task-pv-volume

```

```

labels:
  type: local
spec:
  capacity:
    storage: 1Gi
  accessModes:
    - ReadWriteOnce
  persistentVolumeReclaimPolicy: Delete
  storageClassName: standard
  hostPath:
    path: "/tmp/data"

```

- Check that the PV has the finalizer `kubernetes.io/pv-protection` set:

```

Name:           task-pv-volume
Labels:         type=local
Annotations:   pv.kubernetes.io/bound-by-controller=yes
Finalizers:    [kubernetes.io/pv-protection]
StorageClass:  standard
Status:        Terminating (lasts 1m)
Claim:         default/task-pv-claim
Reclaim Policy: Delete
Access Modes:  RWO
Capacity:     1Gi
Message:
Source:
  Type:      HostPath (bare host directory volume)
  Path:      /tmp/data
  HostPathType:
Events:        <none>

```

- Delete the PV and check that the PV (not bound to a PVC) is removed successfully.

Scenario 2: The PV is bound to a PVC

- Again, create the same PV.
- Create a PVC

```

kind: PersistentVolumeClaim
apiVersion: v1
metadata:
  name: task-pv-claim
spec:
  accessModes:
    - ReadWriteOnce

```

```
resources:  
  requests:  
    storage: 1Gi
```

- Wait until the PV and PVC are bound to each other.
- Delete the PV and verify that the PV is not removed but its status is **Terminating**:

NAME	CAPACITY	ACCESS MODES	RECLAIM POLICY	STATUS	CLAIM
task-pv-volume	1Gi	RWO	Delete	Terminating	default/task-pv-

- Delete the PVC and verify that the PV is removed too.

```
kubectl delete pvc task-pv-claim  
persistentvolumeclaim "task-pv-claim" deleted  
$ kubectl get pvc  
No resources found.  
$ kubectl get pv  
No resources found.
```

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

[Edit This Page](#)

Using CoreDNS for Service Discovery

This page describes how to enable CoreDNS instead of kube-dns for service discovery.

- Before you begin
- Installing CoreDNS with kubeadm
- Using a custom CoreDNS image repository with kubeadm
- Upgrading an Existing Cluster with kubeadm
- 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 version v1.9 or later. To check the version, enter `kubectl version`.

Installing CoreDNS with kubeadm

In Kubernetes 1.9, CoreDNS is available as an alpha feature, and in Kubernetes 1.10 it is available as a beta feature. In either case, you may install it during cluster creation by setting the `CoreDNS` feature gate to `true` during `kubeadm init`:

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

This installs CoreDNS instead of kube-dns.

Using a custom CoreDNS image repository with kubeadm

To use a custom image repository for the CoreDNS image, e.g. one located in your own Docker registry, you can execute the following command after kubeadm has deployed the CoreDNS manifest:

```
kubectl set image -n kube-system deploy/coredns coredns=prefix.example.com/coredns:coredns:1.10.0
```

Upgrading an Existing Cluster with kubeadm

In Kubernetes 1.10, 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.

Note that if you are running CoreDNS in your cluster already, prior to upgrade, your existing 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.

This process will be modified for the GA release of this feature, such that an existing Corefile will not be overwritten.

FEATURE STATE: Kubernetes v1.10 beta

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

- The version names contain beta (e.g. v2beta3).
- Code is well tested. Enabling the feature is considered safe. Enabled by default.
- Support for the overall feature will not be dropped, though details may change.

- The schema and/or semantics of objects may change in incompatible ways in a subsequent beta or stable release. When this happens, we will provide instructions for migrating to the next version. This may require deleting, editing, and re-creating API objects. The editing process may require some thought. This may require downtime for applications that rely on the feature.
- Recommended for only non-business-critical uses because of potential for incompatible changes in subsequent releases. If you have multiple clusters that can be upgraded independently, you may be able to relax this restriction.
- **Please do try our beta features and give feedback on them! After they exit beta, it may not be practical for us to make more changes.**

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.

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

[Edit This Page](#)

Using Sysctls in a Kubernetes Cluster

This document describes how sysctls are used within a Kubernetes cluster.

- Before you begin
- Listing all Sysctl Parameters
- Enabling Unsafe Sysctls
- Setting Sysctls for a Pod
- PodSecurityPolicy Annotations

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`.

Listing all Sysctl Parameters

In Linux, the sysctl interface allows an administrator to modify kernel parameters at runtime. Parameters are available via the `/proc/sys/` virtual process file system. The parameters cover various subsystems such as:

- kernel (common prefix: `kernel.`)
- networking (common prefix: `net.`)
- virtual memory (common prefix: `vm.`)
- MDADM (common prefix: `dev.`)
- More subsystems are described in Kernel docs.

To get a list of all parameters, you can run

```
$ sudo sysctl -a
```

Enabling Unsafe Sysctls

Sysctls are grouped into *safe* and *unsafe* sysctls. In addition to proper namespacing a *safe* sysctl must be properly *isolated* between pods on the same node. This means that setting a *safe* sysctl for one pod

- must not have any influence on any other pod on the node
- must not allow to harm the node's health
- must not allow to gain CPU or memory resources outside of the resource limits of a pod.

By far, most of the *namespaced* sysctls are not necessarily considered *safe*. The following sysctls are supported in the *safe* set:

- `kernel.shm_rmid_forced`,
- `net.ipv4.ip_local_port_range`,
- `net.ipv4.tcp_syncookies`.

Note: The example `net.ipv4.tcp_syncookies` is not namespaced on Linux kernel version 4.4 or lower.

This list will be extended in future Kubernetes versions when the kubelet supports better isolation mechanisms.

All *safe* sysctls are enabled by default.

All *unsafe* sysctls are disabled by default and must be allowed manually by the cluster admin on a per-node basis. Pods with disabled unsafe sysctls will be scheduled, but will fail to launch.

With the warning above in mind, the cluster admin can allow certain *unsafe* sysctls for very special situations like e.g. high-performance or real-time application tuning. *Unsafe* sysctls are enabled on a node-by-node basis with a flag of the kubelet, e.g.:

```
$ kubelet --experimental-allowed-unsafe-sysctls \
  'kernel.msg*,net.ipv4.route.min_pmtu' ...
```

For minikube, this can be done via the `extra-config` flag:

```
$ minikube start --extra-config="kubelet.AllowedExceptions=kernel.msg*,net.ipv4.route.min_pmtu"
```

Only *namespaced* sysctls can be enabled this way.

Setting Sysctls for a Pod

A number of sysctls are *namespaced* in today's Linux kernels. This means that they can be set independently for each pod on a node. Being namespaced is a requirement for sysctls to be accessible in a pod context within Kubernetes.

The following sysctls are known to be *namespaced*:

- `kernel.shm*`,
- `kernel.msg*`,
- `kernel.sem`,
- `fs.mqueue.*`,
- `net.*`.

Sysctls which are not namespaced are called *node-level* and must be set manually by the cluster admin, either by means of the underlying Linux distribution of the nodes (e.g. via `/etc/sysctls.conf`) or using a DaemonSet with privileged containers.

The sysctl feature is an alpha API. Therefore, sysctls are set using annotations on pods. They apply to all containers in the same pod.

Here is an example, with different annotations for *safe* and *unsafe* sysctls:

```
apiVersion: v1
kind: Pod
metadata:
  name: sysctl-example
  annotations:
    security.alpha.kubernetes.io/sysctls: kernel.shm_rmid_forced=1
    security.alpha.kubernetes.io/unsafe-sysctls: net.ipv4.route.min_pmtu=1000,kernel.msgmax=1
spec:
  ...
  
```

Warning: Due to their nature of being *unsafe*, the use of *unsafe* sysctls is at-your-own-risk and can lead to severe problems like wrong

behavior of containers, resource shortage or complete breakage of a node.

It is good practice to consider nodes with special sysctl settings as *tainted* within a cluster, and only schedule pods onto them which need those sysctl settings. It is suggested to use the Kubernetes *taints and toleration* feature to implement this.

A pod with the *unsafe* sysctls will fail to launch on any node which has not enabled those two *unsafe* sysctls explicitly. As with *node-level* sysctls it is recommended to use *taints and toleration* feature or taints on nodes to schedule those pods onto the right nodes.

PodSecurityPolicy Annotations

The use of sysctl in pods can be controlled via annotation on the PodSecurityPolicy.

Sysctl annotation represents a whitelist of allowed safe and unsafe sysctls in a pod spec. It's a comma-separated list of plain sysctl names or sysctl patterns (which end in *). The string * matches all sysctls.

Here is an example, it authorizes binding user creating pod with corresponding sysctls.

```
apiVersion: policy/v1beta1
kind: PodSecurityPolicy
metadata:
  name: sysctl-psp
  annotations:
    security.alpha.kubernetes.io/sysctls: 'net.ipv4.route.*,kernel.msg*'
spec:
  ...

```

Create an Issue Edit this Page

[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 v1.10 alpha

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

- The version names contain alpha (e.g. v1alpha1).
- Might be buggy. Enabling the feature may expose bugs. Disabled by default.
- Support for feature may be dropped at any time without notice.
- The API may change in incompatible ways in a later software release without notice.
- Recommended for use only in short-lived testing clusters, due to increased risk of bugs and lack of long-term support.

The KMS encryption provider uses an envelope encryption scheme to encrypt data in etcd. The data is encrypted using a data encryption key (DEK); a new DEK is generated for each encryption. The DEKs are encrypted with a key encryption key (KEK) that is stored and managed in a remote KMS. The KMS provider uses gRPC to communicate with a specific KMS plugin. The KMS plugin, which is implemented as a gRPC server and deployed on the same host(s) as the Kubernetes master(s), is responsible for all communication with the remote KMS.

Configuring the KMS provider

To configure a KMS provider on the API server, include a provider of type `kms` in the providers array in the encryption configuration file and set the following

properties:

- **name**: Display name of the KMS plugin.
- **endpoint**: Listen address of the gRPC server (KMS plugin). The endpoint is a UNIX domain socket.
- **cachesize**: Number of data encryption keys (DEKs) to be cached in the clear. When cached, DEKs can be used without another call to the KMS; whereas DEKs that are not cached require a call to the KMS to unwrap..

See Understanding the encryption at rest configuration.

Implementing a KMS plugin

To implement a KMS plugin, you can develop a new plugin gRPC server or enable a KMS plugin already provided by your cloud provider. You then integrate the plugin with the remote KMS and deploy it on the Kubernetes master.

Enabling the KMS supported by your cloud provider

Refer to your cloud provider for instructions on enabling the cloud provider-specific KMS plugin.

Developing a KMS plugin gRPC server

You can develop a KMS plugin gRPC server using a stub file available for Go. For other languages, you use a proto file to create a stub file that you can use to develop the gRPC server code.

- Using Go: Use the functions and data structures in the stub file: service.pb.go to develop the gRPC server code
- Using languages other than Go: Use the protoc compiler with the proto file: service.proto to generate a stub file for the specific language

Then use the functions and data structures in the stub file to develop the server code.

Notes:

- kms plugin version: **v1beta1**

In response to procedure call Version, a compatible KMS plugin should return v1beta1 as VersionResponse.version

- message version: **v1beta1**

All messages from KMS provider have the version field set to current version v1beta1

- protocol: UNIX domain socket (`unix`)

The gRPC server should listen at UNIX domain socket

Integrating a KMS plugin with the remote KMS

The KMS plugin can communicate with the remote KMS using any protocol supported by the KMS. All configuration data, including authentication credentials the KMS plugin uses to communicate with the remote KMS, are stored and managed by the KMS plugin independently. The KMS plugin can encode the ciphertext with additional metadata that may be required before sending it to the KMS for decryption.

Deploying the KMS plugin

Ensure that the KMS plugin runs on the same host(s) as the Kubernetes master(s).

Encrypting your data with the KMS provider

To encrypt the data:

1. Create a new encryption configuration file using the appropriate properties for the `kms` provider:

```
kind: EncryptionConfig
apiVersion: v1
resources:
  - resources:
    - secrets
    providers:
      - kms:
          name: myKmsPlugin
          endpoint: unix:///tmp/socketfile.sock
          cachesize: 100
  - identity: {}
```

1. Set the `--experimental-encryption-provider-config` flag on the kube-apiserver to point to the location of the configuration file.
2. Restart your API server.

Verifying that the data is encrypted

Data is encrypted when written to etcd. After restarting your kube-apiserver, any newly created or updated secret should be encrypted when stored. To verify,

you can use the etcdctl command line program to retrieve the contents of your secret.

1. Create a new secret called secret1 in the default namespace:
`kubectl
create secret generic secret1 -n default --from-literal=mykey=mydata`
2. Using the etcdctl command line, read that secret out of etcd:
`ETCDCTL_API=3 etcdctl get /kubernetes.io/secrets/default/secret1 [...] | hexdump -C`
where [...] must be the additional arguments for connecting to the etcd server.
3. Verify the stored secret is prefixed with `k8s:enc:kms:v1:`, which indicates that the `kms` provider has encrypted the resulting data.
4. Verify that the secret is correctly decrypted when retrieved via the API:
`kubectl describe secret secret1 -n default`
should match `mykey: mydata`

Ensuring all secrets are encrypted

Because secrets are encrypted on write, performing an update on a secret encrypts that content.

The following command reads all secrets and then updates them to apply server side encryption. If an error occurs due to a conflicting write, retry the command. For larger clusters, you may wish to subdivide the secrets by namespace or script an update.

```
kubectl get secrets --all-namespaces -o json | kubectl replace -f -
```

Switching from a local encryption provider to the KMS provider

To switch from a local encryption provider to the `kms` provider and re-encrypt all of the secrets:

1. Add the `kms` provider as the first entry in the configuration file as shown in the following example.

```
kind: EncryptionConfig
apiVersion: v1
resources:
  - resources:
    - secrets
  providers:
  - kms:
```

```
    name : myKmsPlugin
    endpoint: unix:///tmp/socketfile.sock
    cachesize: 100
- aescbc:
    keys:
      - name: key1
        secret: <BASE 64 ENCODED SECRET>
```

1. Restart all kube-apiserver processes.
2. Run the following command to force all secrets to be re-encrypted using the `kms` provider.

```
kubectl get secrets --all-namespaces -o json | kubectl replace -f -
```

Disabling encryption at rest

To disable encryption at rest:

1. Place the `identity` provider as the first entry in the configuration file:

```
kind: EncryptionConfig
apiVersion: v1
resources:
- resources:
  - secrets
providers:
- identity: {}
- kms:
    name : myKmsPlugin
    endpoint: unix:///tmp/socketfile.sock
    cachesize: 100
```

1. Restart all kube-apiserver processes.
2. Run the following command to force all secrets to be decrypted. `kubectl get secrets --all-namespaces -o json | kubectl replace -f -`

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

[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
- Create a Pod that has access to the secret data through 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`.

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 Base64 encoding 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`.

Create a Secret

Here is a configuration file you can use to create a Secret that holds your user-name and password:

```
secret.yaml docs/tasks/inject-data-application
apiVersion: v1
kind: Secret
metadata:
  name: test-secret
data:
  username: bXktYXBw
  password: Mzk1MjgkdmRnN0pi
```

1. Create the Secret

```
kubectl create -f https://k8s.io/docs/tasks/inject-data-application/secret.yaml
```

****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'
```

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
```

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:

```
secret-pod.yaml docs/tasks/inject-data-application
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 create -f https://k8s.io/docs/tasks/inject-data-application/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
```

Create a Pod that has access to the secret data through environment variables

Here is a configuration file you can use to create a Pod:

```
secret-envars-pod.yaml docs/tasks/inject-data-application
```

```
apiVersion: v1
kind: Pod
metadata:
  name: secret-envars-test-pod
spec:
  containers:
    - name: envars-test-container
      image: nginx
      env:
        - name: SECRET_USERNAME
          valueFrom:
            secretKeyRef:
              name: test-secret
              key: username
        - name: SECRET_PASSWORD
          valueFrom:
            secretKeyRef:
              name: test-secret
              key: password
```

1. Create the Pod:

```
kubectl create -f https://k8s.io/docs/tasks/inject-data-application/secret-envvars-pod.yaml
```

2. Verify that your Pod is running:

```
kubectl get pod secret-envvars-test-pod
```

Output:

NAME	READY	STATUS	RESTARTS	AGE
secret-envvars-test-pod	1/1	Running	0	4m

3. Get a shell into the Container that is running in your Pod:

```
kubectl exec -it secret-envvars-test-pod - /bin/bash
```

4. In your shell, display the environment variables:

```
root@secret-envvars-test-pod:/# printenv
```

The output includes your username and password:

```
...
SECRET_USERNAME=my-app
...
SECRET_PASSWORD=39528$vdg7Jb
```

What's next

- Learn more about Secrets.
- Learn about Volumes.

Reference

- Secret
- Volume
- Pod

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

[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 PodThe 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.

In this exercise, you create a Pod that runs one container. The configuration file for the Pod defines a command and two arguments:

```
commands.yaml docs/tasks/inject-data-application
```

```
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 create -f https://k8s.io/docs/tasks/inject-data-application/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]
[/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 containers and commands.
- Learn more about configuring pods and containers.
- Learn more about running commands in a container.
- See Container.

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

[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
- 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:

```
envars.yaml docs/tasks/inject-data-application
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 create -f https://k8s.io/docs/tasks/inject-data-application/envvars.yaml
```

2. List the running Pods:

```
kubectl get pods -l purpose=demonstrate-envvars
```

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.

What's next

- Learn more about environment variables.
- Learn about using secrets as environment variables.
- See EnvVarSource.

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

[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
- DownwardAPIVolumeFiles

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:

```
dapi-envars-pod.yaml docs/tasks/inject-data-application
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 create -f https://k8s.io/docs/tasks/inject-data-application/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:

```
dapi-envars-container.yaml docs/tasks/inject-data-application
```

```
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
```

```
dapi-envars-container.yaml docs/tasks/inject-data-application
```

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 create -f https://k8s.io/docs/tasks/inject-data-application/dapi-envars-container.yaml
```

Verify that the Container in the Pod is running:

```
kubectl get pods
```

View the Container's logs:

```
kubectl logs dapi-envars-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

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

[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
- DownwardAPIVolumeFiles

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:

```
dapi-volume.yaml docs/tasks/inject-data-application
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
          readOnly: false
  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 create -f https://k8s.io/docs/tasks/inject-data-application/dapi-volume.yaml
```

Verify that 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
lwxrwxrwx  ... Feb 6 21:47 ..data -> ..2982_06_02_21_47_53.299460680
lwxrwxrwx  ... Feb 6 21:47 annotations -> ..data/annotations
lwxrwxrwx  ... 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:

```
dapi-volume-resources.yaml docs/tasks/inject-data-application
```

```
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
          readOnly: false
    volumes:
      - name: podinfo
    downwardAPI:
      items:
        - path: "cpu_limit"
          resourceFieldRef:
            containerName: client-container
            resource: limits.cpu
        - path: "cpu_request"
          resourceFieldRef:
            containerName: client-container
            resource: requests383cpu
        - path: "mem_limit"
          resourceFieldRef:
            containerName: client-container
            resource: limits.memory
        - path: "mem_request"
          resourceFieldRef:
            containerName: client-container
```

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 should be stored in a file named `cpu_limit`.

Create the Pod:

```
kubectl create -f https://k8s.io/docs/tasks/inject-data-application/dapi-volume-resources.ya
```

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`:
 - `spec.nodeName` - the node's name
 - `status.hostIP` - the node's IP
 - `metadata.name` - the pod's name
 - `metadata.namespace` - the pod's namespace
 - `status.podIP` - the pod's IP address
 - `spec.serviceAccountName` - the pod's service account name
 - `metadata.uid` - the pod's UID
 - `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

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

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

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

[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
- Create a Pod that has access to the secret data through 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`.

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 Base64 encoding 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`.

Create a Secret

Here is a configuration file you can use to create a Secret that holds your user-name and password:

```
secret.yaml docs/tasks/inject-data-application
apiVersion: v1
kind: Secret
metadata:
  name: test-secret
data:
  username: bXktYXBw
  password: Mzk1MjgkdmRnN0pi
```

1. Create the Secret

```
kubectl create -f https://k8s.io/docs/tasks/inject-data-application/secret.yaml
```

****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'
```

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
```

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:

```
secret-pod.yaml docs/tasks/inject-data-application
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 create -f https://k8s.io/docs/tasks/inject-data-application/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
```

Create a Pod that has access to the secret data through environment variables

Here is a configuration file you can use to create a Pod:

```
secret-envars-pod.yaml docs/tasks/inject-data-application
```

```
apiVersion: v1
kind: Pod
metadata:
  name: secret-envars-test-pod
spec:
  containers:
    - name: envars-test-container
      image: nginx
      env:
        - name: SECRET_USERNAME
          valueFrom:
            secretKeyRef:
              name: test-secret
              key: username
        - name: SECRET_PASSWORD
          valueFrom:
            secretKeyRef:
              name: test-secret
              key: password
```

1. Create the Pod:

```
kubectl create -f https://k8s.io/docs/tasks/inject-data-application/secret-envars-pod.yaml
```

2. Verify that your Pod is running:

```
kubectl get pod secret-envars-test-pod
```

Output:

NAME	READY	STATUS	RESTARTS	AGE
secret-envars-test-pod	1/1	Running	0	4m

3. Get a shell into the Container that is running in your Pod:

```
kubectl exec -it secret-envars-test-pod - /bin/bash
```

4. In your shell, display the environment variables:

```
root@secret-envars-test-pod:/# printenv
```

The output includes your username and password:

```
...
SECRET_USERNAME=my-app
...
```

```
SECRET_PASSWORD=39528$vdg7Jb
```

What's next

- Learn more about Secrets.
- Learn about Volumes.

Reference

- Secret
- Volume
- Pod

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

[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
- Create a Pod Preset
- 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`.

Create a Pod Preset

Simple Pod Spec Example

This is a simple example to show how a Pod spec is modified by the Pod Preset.

```
podpreset-preset.yaml docs/tasks/inject-data-application
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 create -f https://k8s.io/docs/tasks/inject-data-application/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 docs/tasks/inject-data-application
```

```
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/docs/tasks/inject-data-application/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 docs/tasks/inject-data-application
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 docs/tasks/inject-data-application
```

```
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 docs/tasks/inject-data-application
```

```
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 docs/tasks/inject-data-application
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
```

```
docs/tasks/inject-data-application
```

```
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 docs/tasks/inject-data-application
```

```
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 docs/tasks/inject-data-application
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
```

```
docs/tasks/inject-data-application
```

```
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 docs/tasks/inject-data-application
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 docs/tasks/inject-data-application
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 docs/tasks/inject-data-application
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 docs/tasks/inject-data-application
```

```
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 docs/tasks/inject-data-application
```

```
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
```

```
docs/tasks/inject-data-application
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 docs/tasks/inject-data-application
```

```
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
Tue, 07 Feb 2017 16:56:12 -0700   Tue, 07 Feb 2017 16:56:12 -0700 1   {podpreset.admission}
```

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
```

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

[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.

```
mysql-deployment.yaml docs/tasks/run-application
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
```

```
mysql-deployment.yaml docs/tasks/run-application
```

```
mysql-pv.yaml docs/tasks/run-application
```

```
kind: PersistentVolume
apiVersion: v1
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 create -f https://k8s.io/docs/tasks/run-application/mysql-pv.yaml
```

2. Deploy the contents of the YAML file:

```
kubectl create -f https://k8s.io/docs/tasks/run-application/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
    -----  -----  -----  -----  -----  -----  -----
    33s       33s       1      {deployment-controller}  Normal

```

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:      mysql-pv-claim
```

```

Namespace:    default
StorageClass:
Status:       Bound
Volume:       mysql-pv
Labels:        <none>
Annotations:   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 -ppass
```

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 r
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

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

[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 version v1.9 or later. 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:

```
deployment.yaml docs/tasks/run-application
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: # create pods using pod definition in this template
    metadata:
      # unlike pod-nginx.yaml, the name is not included in the meta data as a unique name
      # generated from the deployment name
    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/docs/tasks/run-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	1/1	Running	0	16h
nginx-deployment-1771418926-r18az	1/1	Running	0	16h

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.

```
deployment-update.yaml docs/tasks/run-application
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/docs/tasks/run-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:

```
deployment-scale.yaml docs/tasks/run-application
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/docs/tasks/run-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-148880595-6zgi1	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 by using a ReplicationController.

What's next

- Learn more about Deployment objects.

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

[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.

```
mysql-deployment.yaml docs/tasks/run-application
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
```

```
mysql-deployment.yaml docs/tasks/run-application
```

```
mysql-pv.yaml docs/tasks/run-application
```

```
kind: PersistentVolume
apiVersion: v1
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 create -f https://k8s.io/docs/tasks/run-application/mysql-pv.yaml
```

2. Deploy the contents of the YAML file:

```
kubectl create -f https://k8s.io/docs/tasks/run-application/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
    -----  -----  -----  -----  -----  -----  -----
    33s       33s       1      {deployment-controller}  Normal

```

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:      mysql-pv-claim
```

```

Namespace:    default
StorageClass:
Status:       Bound
Volume:       mysql-pv
Labels:        <none>
Annotations:   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 -ppass
```

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 r
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

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

[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 that **this is not a production configuration**. In particular, 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:

```
kubectl create -f https://k8s.io/docs/tasks/run-application/mysql-configmap.yaml
```

```
mysql-configmap.yaml docs/tasks/run-application
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
```

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:

```
kubectl create -f https://k8s.io/docs/tasks/run-application/mysql-services.yaml
```

```
mysql-services.yaml docs/tasks/run-application
# 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-0.mysql.
apiVersion: v1
kind: Service
metadata:
  name: mysql-read
  labels:
    app: mysql
spec:
  ports:
    - name: mysql
      port: 3306
  selector:
    app: mysql
```

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:

```
kubectl create -f https://k8s.io/docs/tasks/run-application/mysql-statefulset.yaml
```

```
mysql-statefulset.yaml docs/tasks/run-application
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
```

`mysql-statefulset.yaml` `docs/tasks/run-application`

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>`. In this case, that 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-minion-group-912

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-minion-group
[...]						
mysql-2	0/2	Pending	0	0s	<none>	kubernetes-minion-group
mysql-2	0/2	Init:0/2	0	0s	<none>	kubernetes-minion-group
mysql-2	0/2	Init:1/2	0	20s	10.244.5.32	kubernetes-minion-group
mysql-2	0/2	PodInitializing	0	21s	10.244.5.32	kubernetes-minion-group
mysql-2	1/2	Running	0	22s	10.244.5.32	kubernetes-minion-group

```
mysql-2  2/2    Running      0       30s    10.244.5.32  kubernetes-minion-group
```

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
data-mysql-0	Bound	pvc-8acbf5dc-b103-11e6-93fa-42010a800002	10Gi	RWO
data-mysql-1	Bound	pvc-8ad39820-b103-11e6-93fa-42010a800002	10Gi	RWO
data-mysql-2	Bound	pvc-8ad69a6d-b103-11e6-93fa-42010a800002	10Gi	RWO
data-mysql-3	Bound	pvc-50043c45-b1c5-11e6-93fa-42010a800002	10Gi	RWO
data-mysql-4	Bound	pvc-500a9957-b1c5-11e6-93fa-42010a800002	10Gi	RWO

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
```

1. Delete the StatefulSet. This also begins terminating the Pods.

```
kubectl delete statefulset mysql
```

1. 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
```

1. 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

- Look in the Helm Charts repository for other stateful application examples.

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

[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:

```
deployment-patch-demo.yaml docs/tasks/run-application
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 create -f https://k8s.io/docs/tasks/run-application/deployment-patch-demo.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-containers.yaml` that has this content:

```
spec:  
  template:  
    spec:  
      containers:  
        - name: patch-demo-ctr-2  
          image: redis
```

Patch your Deployment:

```
kubectl patch deployment patch-demo --patch "$(cat patch-file-containers.yaml)"
```

View the patched Deployment:

```
kubectl get deployment patch-demo --output yaml
```

The output shows that the PodSpec in the Deployment has two Containers:

```
containers:  
  - image: redis  
    imagePullPolicy: Always  
    name: patch-demo-ctr-2  
    ...  
  - image: nginx  
    imagePullPolicy: Always  
    name: patch-demo-ctr  
    ...
```

View the Pods associated with your patched Deployment:

```
kubectl get pods
```

The output shows that the running Pods have different names from the Pods that were running previously. The Deployment terminated the old Pods and created two new Pods that comply with the updated Deployment spec. The 2/2 indicates that each Pod has two Containers:

NAME	READY	STATUS	RESTARTS	AGE
patch-demo-1081991389-2wrn5	2/2	Running	0	1m
patch-demo-1081991389-jmg7b	2/2	Running	0	1m

Take a closer look at one of the patch-demo Pods:

```
kubectl get pod <your-pod-name> --output yaml
```

The output shows that the Pod has two Containers: one running nginx and one running redis:

```
containers:  
  - image: redis  
    ...  
  - image: nginx  
    ...
```

Notes on the strategic merge patch

The patch you did in the preceding exercise is called a *strategic merge patch*. Notice that the patch did not replace the `containers` list. Instead it added a new Container to the list. In other words, the list in the patch was merged with the existing list. This is not always what happens when you use a strategic merge patch on a list. In some cases, the list is replaced, not merged.

With a strategic merge patch, a list is either replaced or merged depending on its patch strategy. The patch strategy is specified by the value of the `patchStrategy` key in a field tag in the Kubernetes source code. For example, the `Containers` field of `PodSpec` struct has a `patchStrategy` of `merge`:

```
type PodSpec struct {
    ...
    Containers []Container `json:"containers" patchStrategy:"merge" patchMergeKey:"name" ...`
```

You can also see the patch strategy in the OpenAPI spec:

```
"io.k8s.api.core.v1.PodSpec": {
    ...
    "containers": {
        "description": "List of containers belonging to the pod. ...",
        "x-kubernetes-patch-merge-key": "name",
        "x-kubernetes-patch-strategy": "merge"
    },
}
```

And you can see the patch strategy in the Kubernetes API documentation.

Create a file named `patch-file-tolerations.yaml` that has this content:

```
spec:
  template:
    spec:
      tolerations:
        - effect: NoSchedule
          key: disktype
          value: ssd
```

Patch your Deployment:

```
kubectl patch deployment patch-demo --patch "$(cat patch-file-tolerations.yaml)"
```

View the patched Deployment:

```
kubectl get deployment patch-demo --output yaml
```

The output shows that the PodSpec in the Deployment has only one Toleration:

```
tolerations:
  - effect: NoSchedule
```

```
key: disktype
value: ssd
```

Notice that the `tolerations` list in the PodSpec was replaced, not merged. This is because the `Tolerations` field of PodSpec does not have a `patchStrategy` key in its field tag. So the strategic merge patch uses the default patch strategy, which is `replace`.

```
type PodSpec struct {
    ...
    Tolerations []Toleration `json:"tolerations,omitempty" protobuf:"bytes,22,opt,name=tolerat
```

Use a JSON merge patch to update a Deployment

A strategic merge patch is different from a JSON merge patch. With a JSON merge patch, if you want to update a list, you have to specify the entire new list. And the new list completely replaces the existing list.

The `kubectl patch` command has a `type` parameter that you can set to one of these values:

Parameter value	Merge type
json	JSON Patch, RFC 6902
merge	JSON Merge Patch, RFC 7386
strategic	Strategic merge patch

For a comparison of JSON patch and JSON merge patch, see [JSON Patch](#) and [JSON Merge Patch](#).

The default value for the `type` parameter is `strategic`. So in the preceding exercise, you did a strategic merge patch.

Next, do a JSON merge patch on your same Deployment. Create a file named `patch-file-2.yaml` that has this content:

```
spec:
  template:
    spec:
      containers:
        - name: patch-demo-ctr-3
          image: gcr.io/google-samples/node-hello:1.0
```

In your patch command, set `type` to `merge`:

```
kubectl patch deployment patch-demo --type merge --patch "$(cat patch-file-2.yaml)"
```

View the patched Deployment:

```
kubectl get deployment patch-demo --output yaml
```

The `containers` list that you specified in the patch has only one Container. The output shows that your list of one Container replaced the existing `containers` list.

```
spec:  
  containers:  
    - image: gcr.io/google-samples/node-hello:1.0  
      ...  
      name: patch-demo-ctr-3
```

List the running Pods:

```
kubectl get pods
```

In the output, you can see that the existing Pods were terminated, and new Pods were created. The 1/1 indicates that each new Pod is running only one Container.

NAME	READY	STATUS	RESTARTS	AGE
patch-demo-1307768864-69308	1/1	Running	0	1m
patch-demo-1307768864-c86dc	1/1	Running	0	1m

Alternate forms of the kubectl patch command

The `kubectl patch` command takes YAML or JSON. It can take the patch as a file or directly on the command line.

Create a file named `patch-file.json` that has this content:

```
{  
  "spec": {  
    "template": {  
      "spec": {  
        "containers": [  
          {  
            "name": "patch-demo-ctr-2",  
            "image": "redis"  
          }  
        ]  
      }  
    }  
  }  
}
```

The following commands are equivalent:

```
kubectl patch deployment patch-demo --patch "$(cat patch-file.yaml)"  
kubectl patch deployment patch-demo --patch 'spec:\n  template:\n    spec:\n      containers:\n        -
```

```
kubectl patch deployment patch-demo --patch "$(cat patch-file.json)"  
kubectl patch deployment patch-demo --patch '{"spec": {"template": {"spec": {"containers": [
```

Summary

In this exercise, you used `kubectl patch` to change the live configuration of a Deployment object. You did not change the configuration file that you originally used to create the Deployment object. Other commands for updating API objects include `kubectl annotate`, `kubectl edit`, `kubectl replace`, `kubectl scale`, and `kubectl apply`.

What's next

- Kubernetes Object Management
- Managing Kubernetes Objects Using Imperative Commands
- Imperative Management of Kubernetes Objects Using Configuration Files
- Declarative Management of Kubernetes Objects Using Configuration Files

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

[Edit This Page](#)

Scale a StatefulSet

This page shows how to scale a StatefulSet.

- Before you begin
- Use `kubectl` to scale StatefulSets
- Troubleshooting
- What's next

Before you begin

- StatefulSets are only available in Kubernetes version 1.5 or later.
- **Not all stateful applications scale nicely.** You need to understand your StatefulSets well before continuing. If you're unsure, remember that it might not be safe to scale your StatefulSets.
- You should perform scaling only when you're sure that your stateful application cluster is completely healthy.

Use `kubectl` to scale StatefulSets

Make sure you have `kubectl` upgraded to Kubernetes version 1.5 or later before continuing. If you're unsure, run `kubectl version` and check `Client Version` for which `kubectl` you're using.

`kubectl scale`

First, find the StatefulSet you want to scale. Remember, you need to first understand if you can scale it or not.

```
kubectl get statefulsets <stateful-set-name>
```

Change the number of replicas of your StatefulSet:

```
kubectl scale statefulsets <stateful-set-name> --replicas=<new-replicas>
```

Alternative: `kubectl apply` / `kubectl edit` / `kubectl patch`

Alternatively, you can do in-place updates on your StatefulSets.

If your StatefulSet was initially created with `kubectl apply` or `kubectl create --save-config`, 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 doesn't 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.

With a StatefulSet of size > 1 , if there is an unhealthy Pod, there is no way for Kubernetes to know (yet) if it is due to a permanent fault or a transient one (upgrade/maintenance/node reboot). 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/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're sure that your stateful application cluster is completely healthy.

What's next

Learn more about deleting a StatefulSet.

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

[Edit This Page](#)

Delete a StatefulSet

This task shows you how to delete a StatefulSet.

- 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 Deleting StatefulSet Pods for details.

What's next

Learn more about force deleting StatefulSet Pods.

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

[Edit This Page](#)

Force Delete StatefulSet Pods

This page shows how to delete Pods which are part of a stateful set, 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 ≥ 1.5 , do the following:

```
kubectl delete pods <pod> --grace-period=0 --force
```

If you're using any version of kubectl ≤ 1.4 , you should omit the `--force` option and use:

```
kubectl delete pods <pod> --grace-period=0
```

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.

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

[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 | -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 no `NEW_NAME` is specified, a new replication controller is created 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 `µs` (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:

```
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:

NAME	READY	STATUS	RESTARTS	AGE

my-nginx-ccba8fdb8cc8160970f63f9a2696fc46-k156z	1/1	Running	0	1m
my-nginx-ccba8fdb8cc8160970f63f9a2696fc46-v95yh	1/1	Running	0	35s
my-nginx-divi2	1/1	Running	0	2h
my-nginx-o0ef1	1/1	Running	0	2h
my-nginx-q6all	1/1	Running	0	8m

kubectl rolling-update reports progress as it progresses:

```
Scaling up my-nginx-ccba8fdb8cc8160970f63f9a2696fc46 from 0 to 3, scaling down my-nginx from 0 to 2
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
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:

```
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:

```

$ kubectl rolling-update my-nginx -f ./nginx-rc.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)
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.

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

[Edit This Page](#)

Horizontal Pod Autoscaler

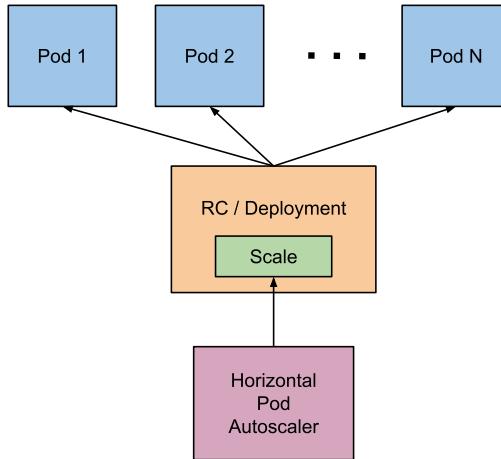
The Horizontal Pod Autoscaler automatically scales the number of pods in a replication controller, deployment or replica set based on observed CPU util-

lization (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
- 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 30 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 autoscaling algorithm design document for further details 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, a single metric is fetched (which describes the object in question), and compared to the target value, to produce a ratio as above.

The HorizontalPodAutoscaler controller can fetch metrics in two different ways: direct Heapster access, and REST client access.

When using direct Heapster access, the HorizontalPodAutoscaler queries Heapster directly through the API server's service proxy subresource. Heapster needs to be deployed on the cluster and running in the kube-system namespace.

See Support for custom metrics for more details on REST client access.

The autoscaler accesses corresponding replication controller, deployment or replica set by 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.

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/v2beta1`. The new fields introduced in `autoscaling/v2beta1` 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 rc foo --min=2 --max=5 --cpu-percent=80` will create an autoscaler for replication controller `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:

- `--horizontal-pod-autoscaler-downscale-delay`: 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`).
- `--horizontal-pod-autoscaler-upscale-delay`: The value for this option is a duration that specifies how long the autoscaler has to wait before another upscale operation can be performed after the current one has completed. The default value is 3 minutes (`3m0s`).

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/v2beta1` 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/v2beta1` API. Kubernetes then queries the new custom metrics API to fetch the values of the appropriate custom metrics.

Requirements

To use custom metrics with your Horizontal Pod Autoscaler, you must set the necessary configurations when deploying your cluster:

- Enable the API aggregation layer if you have not already done so.
- Register your resource metrics API, your custom metrics API and, optionally, external metrics API with the API aggregation layer. All of these API servers must be running *on* your cluster.
 - *Resource Metrics API*: You can use Heapster’s implementation of the resource metrics API, by running Heapster with its `--api-server` flag set to true.
 - *Custom Metrics API*: This must be provided by a separate component. To get started with boilerplate code, see the kubernetes-incubator/custom-metrics-apiserver and the k8s.io/metrics repositories.
 - *External Metrics API*: Starting from Kubernetes 1.10 you can use this API if you need to autoscale on metrics not related to any Kubernetes object. Similarly to *Custom Metrics API* this must be provided by a separate component.
- Set the appropriate flags for kube-controller-manager:
 - `--horizontal-pod-autoscaler-use-rest-clients` should be true.
 - `--kubeconfig <path-to-kubeconfig>` OR `--master <ip-address-of-apiserver>`

Note that either the `--master` or `--kubeconfig` flag can be used; `--master` will override `--kubeconfig` if both are specified. These flags specify the location of the API aggregation layer, allowing the controller manager to communicate to the API server.

In Kubernetes 1.7, the standard aggregation layer that Kubernetes provides runs in-process with the kube-apiserver, so the target IP address can be found with `kubectl get pods --selector k8s-app=kube-apiserver --namespace kube-system -o jsonpath='{{.items[0].status.podIP}}'`.

What's next

- Design documentation: Horizontal Pod Autoscaling.
- `kubectl autoscale` command: `kubectl autoscale`.
- Usage example of Horizontal Pod Autoscaler.

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

[Edit This Page](#)

Horizontal Pod Autoscaler Walkthrough

Horizontal Pod Autoscaler automatically scales the number of pods in a replication controller, deployment or replica 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: Other possible scenarios

Before you begin

This example requires a running Kubernetes cluster and kubectl, version 1.2 or later. Heapster monitoring needs to be deployed in the cluster as Horizontal Pod Autoscaler uses it to collect metrics (if you followed getting started on GCE guide, heapster 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:

```
$ kubectl run php-apache --image=k8s.gcr.io/hpa-example --requests(cpu=200m) --expose --port=80  
service "php-apache" created  
deployment "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  
deployment "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 CURRENT 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 -i --tty 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	CURRENT	MINPODS	MAXPODS	REPLICAS
php-apache	Deployment/php-apache/scale	305% / 50%	305%	1	10	1

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	DESIRED	CURRENT	UP-TO-DATE	AVAILABLE	AGE
php-apache	7	7	7	7	19m

Note Sometimes 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	DESIRED	CURRENT	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/v2beta1` API version.

First, get the YAML of your HorizontalPodAutoscaler in the `autoscaling/v2beta1` form:

```
$ kubectl get hpa.v2beta1.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/v2beta1
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
      targetAverageUtilization: 50
status:
  observedGeneration: 1
  lastScaleTime: <some-time>
  currentReplicas: 1
  desiredReplicas: 1
  currentMetrics:
  - type: Resource
    resource:
      name: cpu
      currentAverageUtilization: 0
      currentAverageValue: 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 Heapster is deployed.

You can also specify resource metrics in terms of direct values, instead of as percentages of the requested value. To do so, use the `targetAverageValue` field instead of the `targetAverageUtilization` field.

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* have the `targetAverageValue` field.

Pod metrics are specified using a metric block like this:

```
type: Pods
pods:
  metricName: packets-per-second
  targetAverageValue: 1k
```

The second alternative metric type is *object metrics*. These metrics describe a different object in the same namespace, instead of describing pods. Note that the metrics are not fetched from the object – they simply describe it. Object metrics do not involve averaging, and look like this:

```
type: Object
object:
  metricName: requests-per-second
  target:
    apiVersion: extensions/v1beta1
    kind: Ingress
    name: main-route
  targetValue: 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/v2beta1
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
    targetAverageUtilization: 50
- type: Pods
  pods:
    metricName: packets-per-second
    targetAverageValue: 1k
- type: Object
  object:
    metricName: requests-per-second
    target:
      apiVersion: extensions/v1beta1
      kind: Ingress
      name: main-route
      targetValue: 10k
status:
  observedGeneration: 1
  lastScaleTime: <some-time>
  currentReplicas: 1
  desiredReplicas: 1
  currentMetrics:
    - type: Resource
      resource:
        name: cpu
        currentAverageUtilization: 0
        currentAverageValue: 0

```

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 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 a certain level of knowledge of your monitoring system, and it requires a cluster monitoring setup similar to one required for using custom metrics. With external metrics, you can autoscale based on

any metric available in your monitoring system by providing a `metricName` field in your HorizontalPodAutoscaler manifest. Additionally you can use a `metricSelector` field to limit which metrics' time series you want to use for autoscaling. If multiple time series are matched by `metricSelector`, the sum of their values is used by the HorizontalPodAutoscaler.

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:
    metricName: queue_messages_ready
    metricSelector:
      matchLabels:
        queue: worker_tasks
    targetAverageValue: 30
```

If your metric describes work or resources that can be divided between autoscaled pods the `targetAverageValue` field describes how much of that work each pod can handle. Instead of using the `targetAverageValue` field, you could use the `targetValue` to define a desired value of your external metric.

Appendix: Horizontal Pod Autoscaler Status Conditions

When using the `autoscaling/v2beta1` 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:
  "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
ScalingActive	True	ValidMetricFound	the HPA was able to successfully cal
ScalingLimited	False	DesiredWithinRange	the desired replica count is within

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: 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:

```
hpa-php-apache.yaml docs/tasks/run-application
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/docs/tasks/run-application/hpa-php-apache.yaml
horizontalpodautoscaler "php-apache" created
```

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

[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

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`.

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.

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.

A `maxUnavailable` of 0% (or 0) or a `minAvailable` of 100% (or equal to the number of replicas) may block node drains entirely. 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`:

```
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):

```
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 the PDB object with a command like `kubectl create -f mypdb.yaml`.

You cannot update PDB objects. They must be deleted and re-created.

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 update 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.

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

[Edit This Page](#)

Running automated tasks with cron jobs

You can use CronJobs to run jobs 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.

Note: CronJob resource in `batch/v2alpha1` API group has been deprecated starting from cluster version 1.8. You should switch to using `batch/v1beta1`, instead, which is enabled by default in the API server. Examples in this document use `batch/v1beta1` in all examples.

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

To check the version, enter `kubectl version`.

- You need a working Kubernetes cluster at version ≥ 1.8 (for CronJob). For previous versions of cluster (< 1.8) you need to explicitly enable `batch/v2alpha1` API by passing `--runtime-config=batch/v2alpha1=true`

to the API server (see Turn on or off an API version for your cluster for more), and then restart both the API server and the controller manager component.

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:

```
cronjob.yaml docs/tasks/job
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 cron job by downloading the example file and then running this command:

```
$ kubectl create -f ./cronjob.yaml
cronjob "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
cronjob "hello" created
```

After creating the cron job, get its status using this command:

```
$ kubectl get cronjob hello
```

```
NAME      SCHEDULE      SUSPEND      ACTIVE      LAST-SCHEDULE
hello    */1 * * * *    False        0           <none>
```

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
NAME      DESIRED      SUCCESSFUL      AGE
hello-4111706356   1            1            2s
```

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
NAME      SCHEDULE      SUSPEND      ACTIVE      LAST-SCHEDULE
hello    */1 * * * *    False        0           Mon, 29 Aug 2016 14:34:00 -0700
```

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 that 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..me

$ echo $pods
hello-4111706356-o9qcm

$ kubectl logs $pods
Mon Aug 29 21:34:09 UTC 2016
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`:

```
$ kubectl delete cronjob hello
cronjob "hello" deleted
```

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.

Note: The 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.

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.

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.

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

[Edit This Page](#)

Running automated tasks with cron jobs

You can use CronJobs to run jobs 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.

Note: CronJob resource in `batch/v2alpha1` API group has been deprecated starting from cluster version 1.8. You should switch to using `batch/v1beta1`, instead, which is enabled by default in the API server. Examples in this document use `batch/v1beta1` in all examples.

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

To check the version, enter `kubectl version`.

- You need a working Kubernetes cluster at version ≥ 1.8 (for Cron-Job). For previous versions of cluster (< 1.8) you need to explicitly enable `batch/v2alpha1` API by passing `--runtime-config=batch/v2alpha1=true` to the API server (see Turn on or off an API version for your cluster for more), and then restart both the API server and the controller manager component.

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:

```
cronjob.yaml docs/tasks/job
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 cron job by downloading the example file and then running this command:

```
$ kubectl create -f ./cronjob.yaml
cronjob "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
cronjob "hello" created
```

After creating the cron job, get its status using this command:

```
$ kubectl get cronjob hello
NAME      SCHEDULE      SUSPEND      ACTIVE      LAST-SCHEDULE
hello    */1 * * * *    False        0           <none>
```

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
NAME            DESIRED   SUCCESSFUL   AGE
hello-4111706356   1         1           2s
```

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
NAME      SCHEDULE      SUSPEND      ACTIVE      LAST-SCHEDULE
hello    */1 * * * *    False        0           Mon, 29 Aug 2016 14:34:00 -0700
```

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 that 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..name}
```



```
$ echo $pods
hello-4111706356-o9qcm

$ kubectl logs $pods
Mon Aug 29 21:34:09 UTC 2016
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`:

```
$ kubectl delete cronjob hello
cronjob "hello" deleted
```

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.

Note: The 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.

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.

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.

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

[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.yaml`

```
job.yaml docs/tasks/job
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.

```
# Expand files into a temporary directory
$ mkdir ./jobs
$ for i in apple banana cherry
```

```

do
  cat job.yaml | sed "s/\$ITEM/$i/" > ./jobs/job-$i.yaml
done

```

Check if it worked:

```

$ ls jobs/
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
job "process-item-apple" created
job "process-item-banana" created
job "process-item-cherry" created

```

Now, check on the jobs:

```

$ kubectl get jobs -l jobgroup=jobexample
NAME      DESIRED   SUCCESSFUL   AGE
process-item-apple   1          1           31s
process-item-banana   1          1           31s
process-item-cherry   1          1           31s

```

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
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

```

There is not a single command to check on the output of all jobs at once, but looping over all the pods is pretty easy:

```

$ for p in $(kubectl get pods -l jobgroup=jobexample -o name)
do
  kubectl logs $p
done
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": "http://www.orangepippin.com/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(sy
```

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 create -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.

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

[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

To check the version, enter `kubectl version`.

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 examples/celery-rabbitmq/rabbitmq-service.yaml
service "rabbitmq-service" created
$ kubectl create -f 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:14.04
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 lets 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:

```
coarse-parallel-processing-work-queue/worker.py  
docs/tasks/job/coarse-parallel-processing-work-queue  
#!/usr/bin/env python  
  
# Just prints standard out and sleeps for 10 seconds.  
import sys  
import time  
print("Processing " + sys.stdin.readlines())  
time.sleep(10)
```

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`.

```
coarse-parallel-processing-work-queue/job.yaml
```

```
docs/tasks/job/coarse-parallel-processing-work-queue
```

```
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 create -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
                job-name=job-wq-1
Annotations:   <none>
```

```

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
  27s        27s       1      {job }           Normal  SuccessfulCreate  Create
  26s        26s       1      {job }           Normal  SuccessfulCreate  Create
  15s        15s       1      {job }           Normal  SuccessfulCreate  Create
  14s        14s       1      {job }           Normal  SuccessfulCreate  Create
  14s        14s       1      {job }           Normal  SuccessfulCreate  Create

```

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 used 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.

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

[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

To check the version, enter `kubectl version`.

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.

Start a temporary Pod running Redis and a service so we can find it.

```
$ kubectl create -f docs/tasks/job/fine-parallel-processing-work-queue/redis-pod.yaml
pod "redis-master" created
$ kubectl create -f docs/tasks/job/fine-parallel-processing-work-queue/redis-service.yaml
service "redis" created
```

If you’re not working from the source tree, you could also download `redis-pod.yaml` and `redis-service.yaml` directly.

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:

```
fine-parallel-processing-work-queue/worker.py

docs/tasks/job/fine-parallel-processing-work-queue
#!/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")
```

If you are working from the source tree, change directory to the `docs/tasks/job/fine-parallel-processing`-directory. Otherwise, download `worker.py`, `rediswq.py`, and `Dockerfile` using above links. 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:

```
fine-parallel-processing-work-queue/job.yaml

docs/tasks/job/fine-parallel-processing-work-queue
-----
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 create -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:       -
      Environment: <none>
      Mounts:     -
      Volumes:    -
  Events:
    FirstSeen  LastSeen  Count  From            SubobjectPath  Type  Reason
    -----  -----  -----  ----  -----
    33s        33s       1      {job-controller }          Normal  SuccessfulCreate
```

```
$ 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 `replicationController`

instead, and consider running a background processing library such as <https://github.com/resque/resque>.

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

[Edit This Page](#)

Create an External Load Balancer

This page shows how to create an External Load Balancer.

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
- 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:

```
{
  "kind": "Service",
  "apiVersion": "v1",
  "metadata": {
    "name": "example-service"
  },
  "spec": {
    "ports": [{
      "port": 8765,
      "targetPort": 9376
    }],
    "selector": {
      "app": "example"
    },
    "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:	123.45.678.9
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 will *not be 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 healthcheck node-Port (numeric port number) for the service. If not specified, `healthCheckNodePort` is created by the service API backend with the allocated node-Port. It will use the user-specified `nodePort` value if specified by the client. It only has an effect when `type` is set to “LoadBalancer” and `externalTrafficPolicy` is set to “Local”.

This feature can be activated by setting `externalTrafficPolicy` to “Local” in the Service Configuration file.

```
{
  "kind": "Service",
  "apiVersion": "v1",
  "metadata": {
    "name": "example-service"
  },
  "spec": {
    "ports": [
      {
        "port": 8765,
        "targetPort": 9376
      }
    ],
    "selector": {
      "app": "example"
    },
    "type": "LoadBalancer",
    "externalTrafficPolicy": "Local"
  }
}
```

Feature availability

k8s version	Feature support
1.7+	Supports the full API fields
1.5 - 1.6	Supports Beta Annotations
<1.5	Unsupported

Below you could find the deprecated Beta annotations used to enable this feature prior to its stable version. Newer Kubernetes versions may stop supporting these after v1.7. Please update existing applications to use the fields directly.

- `service.beta.kubernetes.io/external-traffic` annotation <-> `service.spec.externalTrafficPolicy` field
- `service.beta.kubernetes.io/healthcheck-nodeport` annotation <-> `service.spec.healthCheckNodePort` field

`service.beta.kubernetes.io/external-traffic` annotation has a different set of values compared to the `service.spec.externalTrafficPolicy` field. The values match as follows:

- “OnlyLocal” for annotation <-> “Local” for field
- “Global” for annotation <-> “Cluster” for field

Note that this feature is not currently implemented for all cloud-providers/environments.

Known issues:

- AWS: kubernetes/kubernetes#35758
- Weave-Net: weaveworks/weave/#2924

External Load Balancer Providers

It is important to note that the datapath for this functionality is provided by a load balancer external to the Kubernetes cluster.

When the service type is set to `LoadBalancer`, Kubernetes provides functionality equivalent to `type=<ClusterIP>` to pods within the cluster and extends it by programming the (external to Kubernetes) load balancer with entries for the Kubernetes pods. The Kubernetes service controller automates the creation of the external load balancer, health checks (if needed), firewall rules (if needed) and retrieves the external IP allocated by the cloud provider and populates it in the service object.

Caveats and Limitations when preserving source IPs

GCE/AWS load balancers do not provide weights for their target pools. This was not an issue with the old LB kube-proxy rules which would correctly balance across all endpoints.

With the new functionality, the external traffic will not be equally load balanced across pods, but rather equally balanced at the node level (because GCE/AWS and other external LB implementations do not have the ability for specifying the weight per node, they balance equally across all target nodes, disregarding the number of pods on each node).

We can, however, state that for `NumServicePods << NumNodes` or `NumServicePods >> NumNodes`, a fairly close-to-equal distribution will be seen, even without weights.

Once the external load balancers provide weights, this functionality can be added to the LB programming path. *Future Work: No support for weights is provided for the 1.4 release, but may be added at a future date*

Internal pod to pod traffic should behave similar to ClusterIP services, with equal probability across all pods.

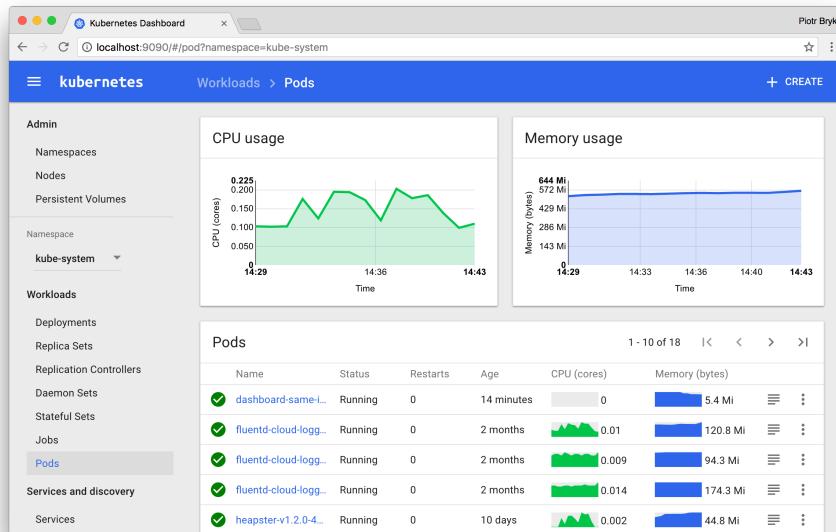
[Create an Issue](#) [Edit this Page](#)

[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 itself along with its attendant 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 create -f https://raw.githubusercontent.com/kubernetes/dashboard/master/src/deploy/recommended.yaml
```

Accessing the Dashboard UI

There are multiple ways you can access the Dashboard UI; either by using the `kubectl` command-line interface, or by accessing the Kubernetes master apiserver using your web browser.

Command line proxy

You can access Dashboard using the `kubectl` command-line tool by running the following command:

```
kubectl proxy
```

Kubectl will handle authentication with apiserver and make Dashboard available at `http://localhost:8001/api/v1/namespaces/kube-system/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.

Master server

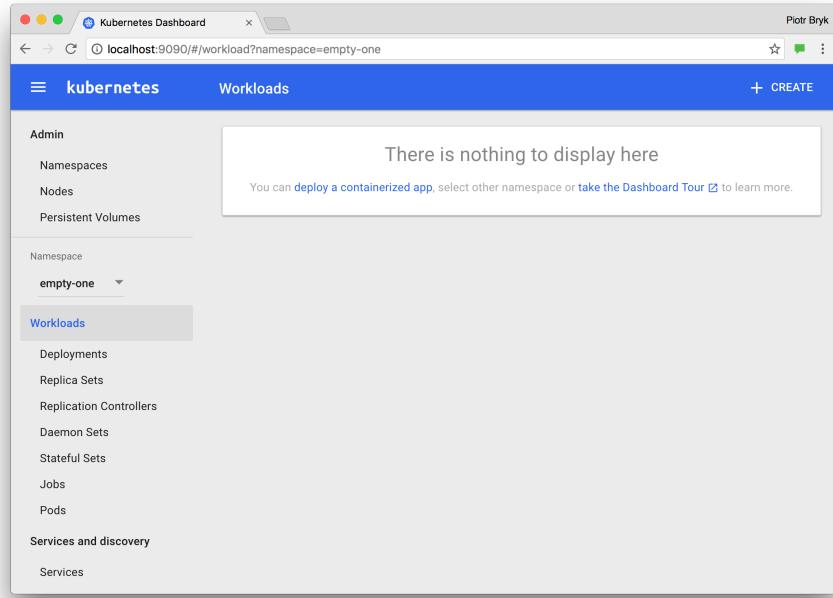
You may access the UI directly via the Kubernetes master apiserver. Open a browser and navigate to `https://<master-ip>:<apiserver-port>/api/v1/namespaces/kube-system/services/https:kubernetes-dashboard:/proxy/` where `<kubernetes-master>` is IP address or domain name of the Kubernetes master.

Please note, this works only if the apiserver is set up to allow authentication with username and password. This is not currently the case with some setup tools (e.g., `kubeadm`). Refer to the authentication admin documentation for information on how to configure authentication manually.

If the username and password are configured but unknown to you, then use `kubectl config view` to find it.

Welcome view

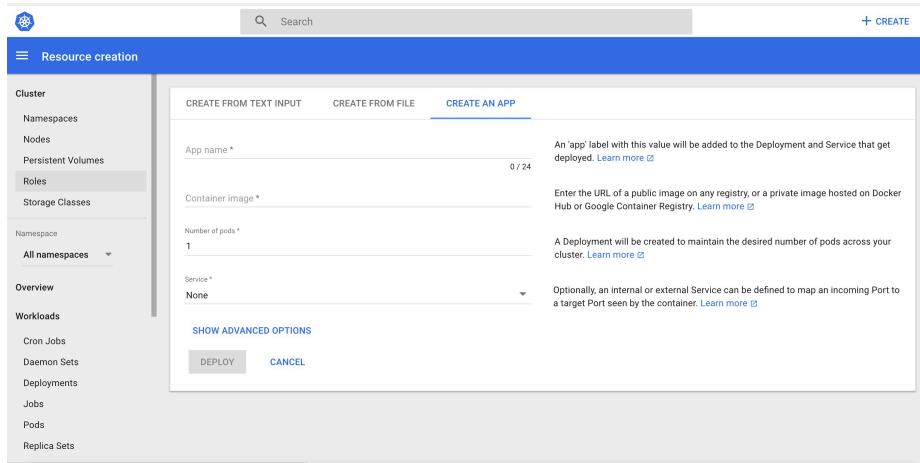
When you access Dashboard on an empty cluster, you'll see the welcome page. This page contains a link to this document as well as a button to deploy your first application. In addition, you can view which system applications are running by default in the `kube-system` namespace of your cluster, for example 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.

To access the deploy wizard from the Welcome page, click the respective button. To access the wizard at a later point in time, click the **CREATE** button in the upper right corner of any page.



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=pod
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,

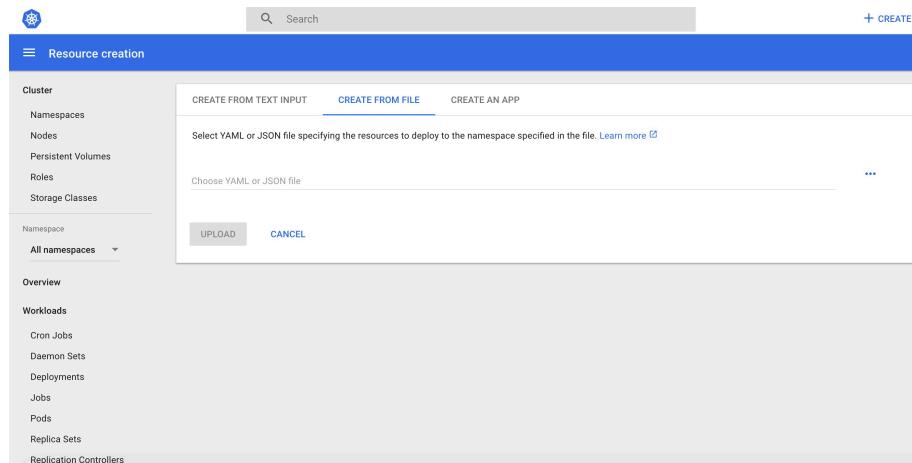
Pods run with unbounded CPU and memory limits.

- **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.

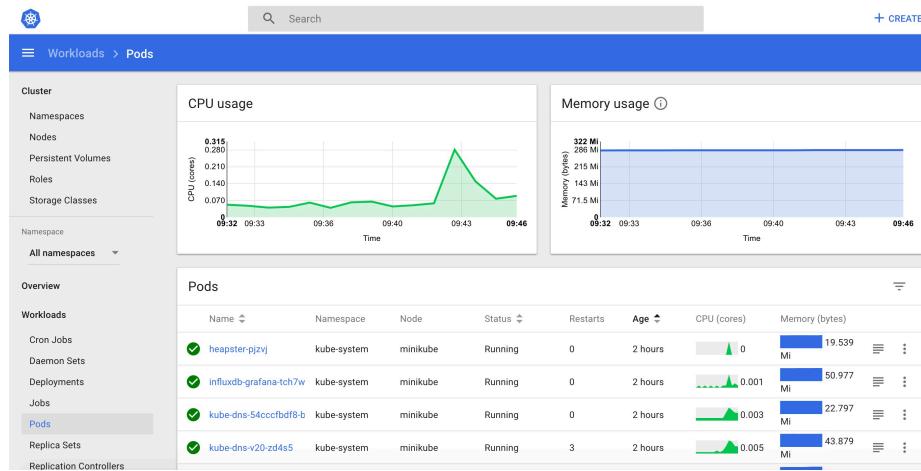
Navigation

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.

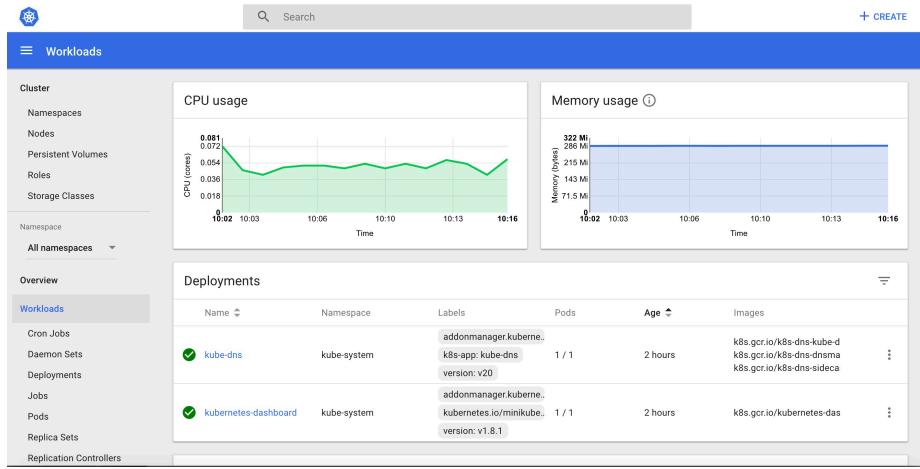
Admin

View for cluster and namespace administrators. It 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

Entry point view that 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.

This screenshot shows a detailed view of a deployment named 'review-app'. The top navigation bar includes 'Workloads > Deployments > review-app' and standard dashboard buttons for EDIT, DELETE, and CREATE.

The left sidebar lists Admin, Namespaces, Nodes, Persistent Volumes, and Workloads (Replica Sets, Deployments, Daemon Sets, Stateful Sets, Jobs, Pods). The Services and discovery section lists Services and Ingresses.

The main content area contains several sections:

- Revision history limit:** Not set. Rolling update strategy: Max surge: 1, Max unavailable: 1. Status: 2 updated, 3 total, 1 available, 2 unavailable.
- New Replica Set:** A table showing a single entry for 'review-app-57159...' with 1 / 2 pods, 28 days age, and the image 'kubernetesdashbo...'. There are three dots at the end of the row.
- Old Replica Sets:** A table showing a single entry for 'review-app-43809...' with 1 / 1 pods, 28 days age, and the image 'kubernetesdashbo...'. There are three dots at the end of the row.
- Horizontal Pod Autoscalers:** A table showing one entry for 'review-app' with target CPU utilization at 0%, current utilization at 0%, min replicas at 2, max replicas at 8, and an age of 10 days. There are three dots at the end of the row.

Services and discovery

Services and discovery view 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.

The screenshot shows the Kubernetes Services page. The left sidebar has a 'Services' section selected. The main area displays a table of services:

Name	Namespace	Labels	Cluster IP	Internal endpoints	External endpoints	Age
kubernetes	default	component: apiser, provider: kubernetes	10.96.0.1	kubernetes:443 TCP kubernetes:0 TCP	-	2 hours
heapster	kube-system	addonmanager.kubernetes.io/mode=Always	10.99.185.73	heapster:kube-system heapster:kube-system	-	2 hours
kube-dns	kube-system	addonmanager.kubernetes.io/mode=Always k8s-app: kube-dns	10.96.0.10	kube-dns:kube-system kube-dns:kube-system kube-dns:kube-system kube-dns:kube-system	-	2 hours
kubernetes-dashboard	kube-system	addonmanager.kubernetes.io/mode=Always app: kubernetes-dashboard	10.111.43.173	kubernetes-dashboard:kubernetes-dashboard kubernetes-dashboard:kubernetes-dashboard	-	2 hours
monitoring-grafana	kube-system	addonmanager.kubernetes.io/mode=Always kubernetes.io/minion-role=master	10.98.122.255	monitoring-grafana:monitoring-grafana monitoring-grafana:monitoring-grafana	-	2 hours

Storage

Storage view shows Persistent Volume Claim resources which are used by applications for storing data.

Config

Config view shows all Kubernetes resources that are used for live configuration of applications running in clusters. This is now Config Maps and Secrets. The view allows for editing and managing config objects and displays secrets hidden by default.

The screenshot shows the Kubernetes Secrets page. The left sidebar has a 'Secrets' section selected. The main area displays a detailed view of a secret named 'default-token-l8xm6':

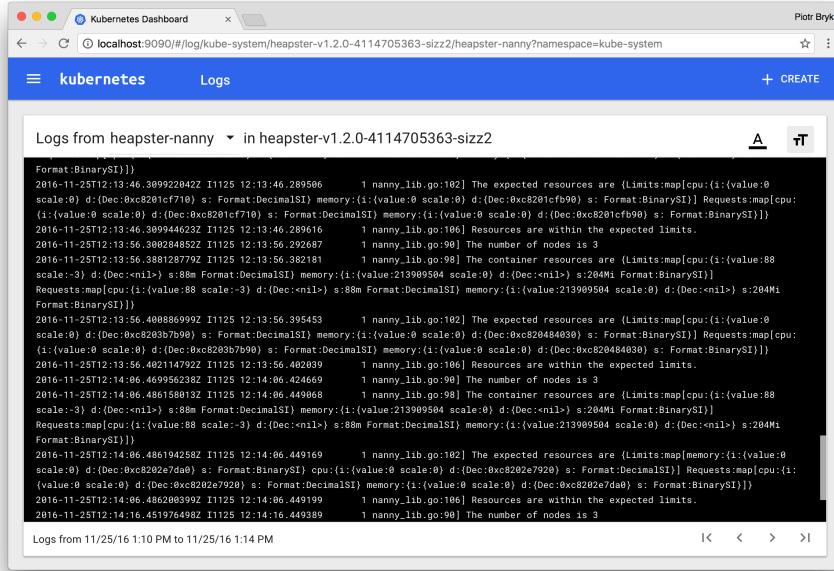
Details	
Name:	default-token-l8xm6
Namespace:	kube-public
Annotations:	kubernetes.io/service-account.name: default kubernetes.io/service-account.uid: 9e2ee2dc-31e7-11e8-8106-de40fc5cd9dc
Creation Time:	2018-03-27T17:52 UTC
Type:	kubernetes.io/service-account-token

Data

- ca.crt: 1066 bytes
- namespace: 11 bytes
- token: 856 bytes

Logs viewer

Pod lists and detail pages link to logs viewer that is built into Dashboard. The viewer allows for drilling down logs from containers belonging to a single Pod.



The screenshot shows the Kubernetes Dashboard with the URL `localhost:9090/#/log/kube-system/heapster-v1.2.0-4114705363-sizz2/heapster-nanny?namespace=kube-system`. The page title is "Logs". The logs displayed are from the "heapster-nanny" container in the "heapster-v1.2.0-4114705363-sizz2" pod. The logs show various system messages and resource usage statistics, such as CPU and memory limits and requests. The logs are timestamped and include file paths like `/var/log/containers/heapster-nanny`.

```
Format:Binary{SI})}
2016-11-25T12:13:46.39994646232 I|I125 12:13:46.289506      1 nanny.lib.go:102] The expected resources are {Limits:map[cpu:{i:(value:0 scale:0) d:(Dec:0xc820fc7f10) s: Format:DecimalSI} memory:{i:(value:0 scale:0) d:(Dec:0xc820fcfb90) s: Format:BinarySI}] Requests:map[cpu:{i:(value:0 scale:0) d:(Dec:0xc820fc7f10) s: Format:DecimalSI} memory:{i:(value:0 scale:0) d:(Dec:0xc820fcfb90) s: Format:BinarySI}]}
2016-11-25T12:13:46.39994646232 I|I125 12:13:46.289616      1 nanny.lib.go:106] Resources are within the expected limits.
2016-11-25T12:13:56.3802848522 I|I125 12:13:56.292687      1 nanny.lib.go:98] The number of nodes is 3
2016-11-25T12:13:56.3881287792 I|I125 12:13:56.382181      1 nanny.lib.go:98] The container resources are {Limits:map[cpu:{i:(value:88 scale:-3) d:(Dec:<n1>) s:88m Format:DecimalSI} memory:{i:(value:213990504 scale:0) d:(Dec:<n1>) s:204Mi Format:BinarySI}]
Requests:map[cpu:{i:(value:88 scale:-3) d:(Dec:<n1>) s:88m Format:DecimalSI} memory:{i:(value:213990504 scale:0) d:(Dec:<n1>) s:204Mi Format:BinarySI}]}
2016-11-25T12:13:56.4008869992 I|I125 12:13:56.395453      1 nanny.lib.go:102] The expected resources are {Limits:map[cpu:{i:(value:0 scale:0) d:(Dec:0xc82057d90) s: Format:DecimalSI} memory:{i:(value:0 scale:0) d:(Dec:0xc82044030) s: Format:BinarySI}] Requests:map[cpu:{i:(value:0 scale:0) d:(Dec:0xc82057d90) s: Format:DecimalSI} memory:{i:(value:0 scale:0) d:(Dec:0xc82044030) s: Format:BinarySI}]}
2016-11-25T12:13:56.4021147922 I|I125 12:13:56.402099      1 nanny.lib.go:106] Resources are within the expected limits.
2016-11-25T12:14:06.4609562382 I|I125 12:14:06.424669      1 nanny.lib.go:98] The number of nodes is 3
2016-11-25T12:14:06.4861580132 I|I125 12:14:06.449068      1 nanny.lib.go:98] The container resources are {Limits:map[cpu:{i:(value:88 scale:-3) d:(Dec:<n1>) s:88m Format:DecimalSI} memory:{i:(value:213990504 scale:0) d:(Dec:<n1>) s:204Mi Format:BinarySI}]
Requests:map[cpu:{i:(value:88 scale:-3) d:(Dec:<n1>) s:88m Format:DecimalSI} memory:{i:(value:213990504 scale:0) d:(Dec:<n1>) s:204Mi Format:BinarySI}]}
2016-11-25T12:14:06.4861580132 I|I125 12:14:06.449169      1 nanny.lib.go:102] The expected resources are {Limits:map[memory:{i:(value:0 scale:0) d:(Dec:0xc8202e7dd0) s: Format:BinarySI} cpu:{i:(value:0 scale:0) d:(Dec:0xc8202e7920) s: Format:DecimalSI}] Requests:map[cpu:{i:(value:0 scale:0) d:(Dec:0xc8202e7920) s: Format:DecimalSI} memory:{i:(value:0 scale:0) d:(Dec:0xc8202e7dd0) s: Format:BinarySI}]}
2016-11-25T12:14:06.4862083992 I|I125 12:14:06.449199      1 nanny.lib.go:106] Resources are within the expected limits.
2016-11-25T12:14:16.4519764982 I|I125 12:14:16.449389      1 nanny.lib.go:98] The number of nodes is 3
```

What's next

For more information, see the Kubernetes Dashboard project page.

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

[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/
{
  "versions": [
    "v1"
  ]
}
```

Without `kubectl proxy`

In Kubernetes version 1.3 or later, `kubectl config view` no longer displays the token. Use `kubectl describe secret...` to get the token for the default service account, like this:

```
$ APISERVER=$(kubectl config view | grep server | cut -f 2- -d ":" | tr -d " ")
$ TOKEN=$(kubectl describe secret $(kubectl get secrets | grep default | cut -f1 -d ' ') | grep -w token | awk '{print $2}')
$ 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 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/<version number>/kubernetes`. 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/1.4/pkg/api/v1"` 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
```

```
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/elasticse...
kibana-logging is running at https://104.197.5.247/api/v1/namespaces/kube-system/services/kibana-logging...
kube-dns is running at https://104.197.5.247/api/v1/namespaces/kube-system/services/kube-dns...
grafana is running at https://104.197.5.247/api/v1/namespaces/kube-system/services/grafana...
heapster is running at https://104.197.5.247/api/v1/namespaces/kube-system/services/monit...
heapster is running at https://104.197.5.247/api/v1/namespaces/kube-system/services/monit...
```

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/` 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/elasticsearc...` (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]`

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/`

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/_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/_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.

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

[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 --insecure-skip-tls-verify=true  
kubectl config --kubeconfig=config-demo set-cluster scratch --server=https://5.6.7.8 --insecure-skip-tls-verify=true
```

Add user details to your configuration file:

```
kubectl config --kubeconfig=config-demo set-credentials developer --client-certificate=fake-ca-file.pem  
kubectl config --kubeconfig=config-demo set-credentials experimenter --username=exp --password=exp
```

Add context details to your configuration file:

```
kubectl config --kubeconfig=config-demo set-context dev-frontend --cluster=development --name=dev-frontend  
kubectl config --kubeconfig=config-demo set-context dev-storage --cluster=development --name=dev-storage  
kubectl config --kubeconfig=config-demo set-context exp-scratch --cluster=scratch --name=exp-scratch
```

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.pem  
    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

```

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, on Linux:

```
export KUBECONFIG_SAVED=$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, on Linux:

```
export KUBECONFIG=$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, on Linux:

```
export KUBECONFIG=$KUBECONFIG:$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, on Linux:

```
export KUBECONFIG=$KUBECONFIG_SAVED
```

What's next

- Organizing Cluster Access Using kubeconfig Files
- `kubectl config`

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

[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 create -f https://k8s.io/docs/tutorials/stateless-application/guestbook/redis-master-deployment.yaml
```

The output of a successful command verifies that the deployment was created:

```
deployment "redis-master" created
```

When the pod is ready, you can get:

```
kubectl get pods
```

NAME	READY	STATUS	RESTARTS	AGE
redis-master-765d459796-258hz	1/1	Running	0	50s

```
kubectl get deployment
```

```
NAME      DESIRED  CURRENT  UP-TO-DATE  AVAILABLE  AGE
redis-master  1        1         1          1         55s
```

```
kubectl get rs NAME DESIRED CURRENT READY AGE redis-master-765d459796 1 1 1 1m
```

2. Create a Redis service:

```
kubectl create -f https://k8s.io/docs/tutorials/stateless-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
```

```
NAME      TYPE      CLUSTER-IP      EXTERNAL-IP      PORT(S)      AGE
redis-master  ClusterIP  10.0.0.213  <none>           6379/TCP   27s
```

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 (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 service name, to select a matching pod to port forward to since Kubernetes v1.10.

```
kubectl port-forward redis-master-765d459796-258hz 6379:6379
```

which is the same as

```
kubectl port-forward pods/redis-master-765d459796-258hz 6379:6379
```

or

```
kubectl port-forward deployment/redis-master 6379:6379
```

or

```
kubectl port-forward rs/redis-master 6379:6379
```

or

```
kubectl port-forward svc/redis-master 6379:6379
```

Any of the above commands works. The output is similar to this:

```
I0710 14:43:38.274550    3655 portforward.go:225] Forwarding from 127.0.0.1:6379 -> 6379
I0710 14:43:38.274797    3655 portforward.go:225] Forwarding from [::1]:6379 -> 6379
```

2. Start the Redis command line interface:

```
redis-cli
```

3. At the Redis command line prompt, enter the ping command:

```
127.0.0.1:6379>ping
```

A successful ping request returns PONG.

Discussion

Connections made to local port 6379 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.

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

[Edit This Page](#)

Provide Load-Balanced Access to an Application in a Cluster

This page shows how to create a Kubernetes Service object that provides load-balanced access to an application running in a cluster.

- Objectives
- Before you begin
- Creating a Service for an application running in two pods
- Using a service configuration file
- What's next

Objectives

- Run two instances of a Hello World application
- Create a Service object
- 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

1. Run a Hello World application in your cluster:

```
kubectl run hello-world --replicas=2 --labels="run=load-balancer-example" --image=gcr.io/google-samples/hello-app:1.0
```

2. List the pods that are running the Hello World application:

```
kubectl get pods --selector="run=load-balancer-example"
```

The output is similar to this:

NAME	READY	STATUS	RESTARTS	AGE
hello-world-2189936611-8fyp0	1/1	Running	0	6m
hello-world-2189936611-9isq8	1/1	Running	0	6m

3. Create a Service object that exposes the deployment:

```
kubectl expose deployment <your-deployment-name> --type=NodePort --name=example-service
```

where `<your-deployment-name>` is the name of your deployment.

4. Display the IP addresses for your service:

```
kubectl get services example-service
```

The output shows the internal IP address and the external IP address of your service. If the external IP address shows as `<pending>`, repeat the command.

Note: If you are using Minikube, you don't get an external IP address. The external IP address remains in the pending state.

NAME	CLUSTER-IP	EXTERNAL-IP	PORT(S)	AGE
example-service	10.0.0.160	<pending>	8080/TCP	40s

1. Use your Service object to access the Hello World application:

```
curl :8080
```

where <your-external-ip-address> is the external IP address of your service.

The output is a hello message from the application:

```
Hello Kubernetes!
```

Note: If you are using Minikube, enter these commands:

```
kubectl cluster-info  
kubectl describe services example-service
```

The output displays the IP address of your Minikube node and the NodePort value for your service. Then enter this command to access the Hello World application:

```
curl <minikube-node-ip-address>:<service-node-port>
```

where <minikube-node-ip-address> us the IP address of your Minikube node, and <service-node-port> is the NodePort value for your service.

Using a service configuration file

As an alternative to using `kubectl expose`, you can use a service configuration file to create a Service.

What's next

Learn more about connecting applications with services.

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

[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

1. Run a Hello World application in your cluster:

```
kubectl run hello-world --replicas=2 --labels="run=load-balancer-example" --image=gcr.io/k8s-staging-ingress-controller/hello-world:1.10.0
```

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 replicaset
kubectl describe replicaset
```

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.

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

[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 Service object.

- Objectives
- Before you begin
- What's next

Objectives

- Create and run a microservice using a Deployment 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:

```
hello.yaml docs/tasks/access-application-cluster
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 create -f https://k8s.io/docs/tasks/access-application-cluster/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 selector labels to find the Pods that it routes traffic to.

First, explore the Service configuration file:

```
hello-service.yaml docs/tasks/access-application-cluster
```

```
kind: Service
apiVersion: v1
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 create -f https://k8s.io/docs/tasks/access-application-cluster/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:

`frontend/frontend.conf`

`docs/tasks/access-application-cluster/frontend`

```
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.

```
frontend.yaml docs/tasks/access-application-cluster
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 create -f https://k8s.io/docs/tasks/access-application-cluster/frontend.yaml
```

The output verifies that both resources were created:

```
deployment "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
```

The external IP field may take some time to populate. If this is the case, the external IP is listed as <pending>.

NAME	CLUSTER-IP	EXTERNAL-IP	PORT(S)	AGE
frontend	10.51.252.116	<pending>	80/TCP	10s

Repeat the same command again until it shows an external IP address:

NAME	CLUSTER-IP	EXTERNAL-IP	PORT(S)	AGE
frontend	10.51.252.116	XXX.XXX.XXX.XXX	80/TCP	1m

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>
```

The output shows the message generated by the backend:

```
{"message": "Hello"}
```

What's next

- Learn more about Services
- Learn more about ConfigMaps

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

[Edit This Page](#)

Create an External Load Balancer

This page shows how to create an External Load Balancer.

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
- 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:

```
{
  "kind": "Service",
  "apiVersion": "v1",
  "metadata": {
```

```

        "name": "example-service"
    },
    "spec": {
        "ports": [{
            "port": 8765,
            "targetPort": 9376
        }],
        "selector": {
            "app": "example"
        },
        "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:	123.45.678.9
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 will *not be 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 healthcheck nodePort (numeric port number) for the service. If not specified, `healthCheckNodePort` is created by the service API backend with the allocated nodePort. It will use the user-specified nodePort value if specified by the client. It only has an effect when type is set to “LoadBalancer” and `externalTrafficPolicy` is set to “Local”.

This feature can be activated by setting `externalTrafficPolicy` to “Local” in the Service Configuration file.

```
{
  "kind": "Service",
  "apiVersion": "v1",
  "metadata": {
    "name": "example-service"
  },
  "spec": {
    "ports": [
      {
        "port": 8765,
        "targetPort": 9376
      }
    ],
    "selector": {
      "app": "nginx"
    }
  }
}
```

```

        "app": "example"
    },
    "type": "LoadBalancer",
    "externalTrafficPolicy": "Local"
}
}

```

Feature availability

k8s version	Feature support
1.7+	Supports the full API fields
1.5 - 1.6	Supports Beta Annotations
<1.5	Unsupported

Below you could find the deprecated Beta annotations used to enable this feature prior to its stable version. Newer Kubernetes versions may stop supporting these after v1.7. Please update existing applications to use the fields directly.

- `service.beta.kubernetes.io/external-traffic` annotation <-> `service.spec.externalTrafficPolicy` field
- `service.beta.kubernetes.io/healthcheck-nodeport` annotation <-> `service.spec.healthCheckNodePort` field

`service.beta.kubernetes.io/external-traffic` annotation has a different set of values compared to the `service.spec.externalTrafficPolicy` field. The values match as follows:

- “OnlyLocal” for annotation <-> “Local” for field
- “Global” for annotation <-> “Cluster” for field

Note that this feature is not currently implemented for all cloud-providers/environments.

Known issues:

- AWS: kubernetes/kubernetes#35758
- Weave-Net: weaveworks/weave/#2924

External Load Balancer Providers

It is important to note that the datapath for this functionality is provided by a load balancer external to the Kubernetes cluster.

When the service type is set to `LoadBalancer`, Kubernetes provides functionality equivalent to `type=<ClusterIP>` to pods within the cluster and extends it by

programming the (external to Kubernetes) load balancer with entries for the Kubernetes pods. The Kubernetes service controller automates the creation of the external load balancer, health checks (if needed), firewall rules (if needed) and retrieves the external IP allocated by the cloud provider and populates it in the service object.

Caveats and Limitations when preserving source IPs

GCE/AWS load balancers do not provide weights for their target pools. This was not an issue with the old LB kube-proxy rules which would correctly balance across all endpoints.

With the new functionality, the external traffic will not be equally load balanced across pods, but rather equally balanced at the node level (because GCE/AWS and other external LB implementations do not have the ability for specifying the weight per node, they balance equally across all target nodes, disregarding the number of pods on each node).

We can, however, state that for NumServicePods << NumNodes or NumServicePods >> NumNodes, a fairly close-to-equal distribution will be seen, even without weights.

Once the external load balancers provide weights, this functionality can be added to the LB programming path. *Future Work: No support for weights is provided for the 1.4 release, but may be added at a future date*

Internal pod to pod traffic should behave similar to ClusterIP services, with equal probability across all pods.

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

[Edit This Page](#)

Configure Your Cloud Provider's Firewalls

ERROR

You must define a overview

This template requires that you provide text that states, in one or two sentences, the purpose of this document. The text in this block will be displayed under the heading . To get rid of this message and take advantage of this template, capture the *overview* variable and populate it with content.

- 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 and AWS. 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.

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

[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.containers[*].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'}\nsort
```

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.c
```

What's next

Reference

- Jsonpath reference guide
- Go template reference guide

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

[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.

- 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:

```
two-container-pod.yaml docs/tasks/access-application-cluster
```

```
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 create -f https://k8s.io/docs/tasks/access-application-cluster/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 Volume.
- See Pod.

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

[Edit This Page](#)

Configure DNS for a Cluster

Kubernetes offers a DNS cluster addon, which most of the supported environments enable by default.

For more information on how to configure DNS for a Kubernetes cluster, see the Kubernetes DNS sample plugin.

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

[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 `kubectl 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 that `-c ${CONTAINER_NAME}` is optional and can be omitted 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 create --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 false alarm, see https://github.com/kubernetes/kubernetes/issues/12345
```

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 `containerPort`

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.

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

[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.

```
nginx-dep.yaml docs/tasks/debug-application-cluster
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 create -f https://k8s.io/docs/tasks/debug-application-cluster/nginx-dep.yaml
deployment "nginx-deployment" created

$ kubectl get pods
NAME                      READY   STATUS    RESTARTS   AGE
nginx-deployment-1006230814-6winp   1/1     Running   0          11s
nginx-deployment-1006230814-fmgu3   1/1     Running   0          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","re
Status:        Running
IP:            10.244.0.6
Controllers:   ReplicaSet/nginx-deployment-1006230814
Containers:
  nginx:
    Container ID:  docker://90315cc9f513c724e9957a4788d3e625a078de84750f244a40f97ae355eb114
    Image:         nginx
    Image ID:     docker://6f62f48c4e55d700cf3eb1b5e33fa051802986b77b874cc351cce539e516370
    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-4bcbi:
    Type:  Secret (a volume populated by a Secret)
    SecretName: default-token-4bcbi
    Optional:  false
  QoS Class:  Guaranteed
  Node-Selectors: <none>
  Tolerations: <none>
Events:
  FirstSeen  LastSeen  Count  From           SubobjectPath  Type  Reason
  -----  -----  -----  ----  -----  -----  -----
  54s       54s       1  {default-scheduler }                  Normal  Scheduled
  54s       54s       1  {kubelet kubernetes-node-wul5}  spec.containers{nginx}  Normal
  53s       53s       1  {kubelet kubernetes-node-wul5}  spec.containers{nginx}  Normal
  53s       53s       1  {kubelet kubernetes-node-wul5}  spec.containers{nginx}  Normal

```

```
 53s      53s      1  {kubelet kubernetes-node-wul5}  spec.containers{nginx}  Normal
```

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    RESTARTS   AGE
nginx-deployment-1006230814-6winp   1/1     Running   0          7m
nginx-deployment-1006230814-fmgu3   1/1     Running   0          7m
nginx-deployment-1370807587-6ekbw   1/1     Running   0          1m
nginx-deployment-1370807587-fg172   0/1     Pending   0          1m
nginx-deployment-1370807587-fz9sd   0/1     Pending   0          1m
```

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  LastSeen   Count  From           SubobjectPath  Type  Reason
    -----  -----  ----  ----  {default-scheduler }          -----
    1m        48s       7      {default-scheduler }          Warning  FailedScheduling
    fit failure on node (kubernetes-node-6ta5): Node didn't have enough resource: CPU, request
    fit failure on node (kubernetes-node-wul5): Node didn't have enough resource: CPU, request
```

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", "name": "nginx-deployment", "namespace": "default", "uid": "4c879808-f161-11e5-9a78-42010af00005", "version": "133447"}, "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", "selfLink": "/api/v1/namespaces/default/pods/nginx-deployment-1006230814-6winp", "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://90315cc9f513c724e9957a4788d3e625a078de84750f244a40f97ae355eb1149
      image: nginx
      imageID: docker://6f62f48c4e55d700cf3eb1b5e33fa051802986b77b874cc351cce539e5163707
      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    AGE     VERSION
kubernetes-node-861h  NotReady  1h      v1.6.0+fff5156
kubernetes-node-bols Ready   1h      v1.6.0+fff5156
kubernetes-node-st6x Ready   1h      v1.6.0+fff5156
kubernetes-node-unaj Ready   1h      v1.6.0+fff5156

$ kubectl describe node kubernetes-node-861h
Name:           kubernetes-node-861h
Role:
Labels:         beta.kubernetes.io/arch=amd64
                beta.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          LastTransitionTime      Reason
  ----        -----  -----                    -----                  -----
  OutOfDisk   Unknown Fri, 08 Sep 2017 16:04:28 +0800
  MemoryPressure Unknown Fri, 08 Sep 2017 16:04:28 +0800
  DiskPressure Unknown Fri, 08 Sep 2017 16:04:28 +0800
  Ready       Unknown Fri, 08 Sep 2017 16:04:28 +0800
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:
  Namespace          Name
  ----              ---
  .....             ----
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"
  selfLink: /api/v1/nodes/kubernetes-node-861h
  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-gfffc5a86098dc01
  kubeletVersion: v0.21.1-185-gfffc5a86098dc01
  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

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

[Edit This Page](#)

Auditing

FEATURE STATE: Kubernetes v1.10 beta

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

- The version names contain beta (e.g. v2beta3).
- Code is well tested. Enabling the feature is considered safe. Enabled by default.
- Support for the overall feature will not be dropped, though details may change.

- The schema and/or semantics of objects may change in incompatible ways in a subsequent beta or stable release. When this happens, we will provide instructions for migrating to the next version. This may require deleting, editing, and re-creating API objects. The editing process may require some thought. This may require downtime for applications that rely on the feature.
- Recommended for only non-business-critical uses because of potential for incompatible changes in subsequent releases. If you have multiple clusters that can be upgraded independently, you may be able to relax this restriction.
- **Please do try our beta features and give feedback on them! After they exit beta, it may not be practical for us to make more changes.**

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
- Multi-cluster setup
- Log Collector Examples
- Legacy Audit

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:

```
audit-policy.yaml docs/tasks/debug-application-cluster
apiVersion: audit.k8s.io/v1beta1 # 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 group73
        resources: ["configmaps"]
  # This rule only applies to resources in the "kube-system" namespace.
  # The empty string "" can be used to select non-namespaced resources.
  namespaces: ["kube-system"]

  # Log configmap and secret changes in all other namespaces at the Metadata level.
  - level: Metadata
    resources:
```

```
audit-policy.yaml docs/tasks/debug-application-cluster
```

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/v1beta1
kind: Policy
rules:
- level: Metadata
```

The audit profile used by GCE should be used as reference by admins constructing their own audit profiles.

Audit backends

Audit backends persist audit events to an external storage. Kube-apiserver out of the box provides two backends:

- Log backend, which writes events to a disk
- Webhook backend, which sends events to an external API

In both cases, audit events structure is defined by the API in the `audit.k8s.io` API group. The current version of the API is `v1beta1`.

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` defines 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.

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.

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.

Multi-cluster setup

If you're extending the Kubernetes API with the aggregation layer, you can also set up audit logging for the aggregated apiserver. To do this, pass the configuration options in the same format as described above to the aggregated apiserver and set up the log ingesting pipeline to pick up audit logs. Different apiservers can have different audit configurations and different audit policies.

Log Collector Examples

Use fluentd to collect and distribute audit events from log file

Fluentd is an open source data collector for unified logging layer. In this example, we will use fluentd to split audit events by different namespaces.

1. install fluentd, fluent-plugin-forest and fluent-plugin-rewrite-tag-filter in the kube-apiserver node
2. create a config file for fluentd

```
$ cat <<EOF > /etc/fluentd/config
# fluentd conf runs in the same host with kube-apiserver
<source>
  @type tail
  # audit log path of kube-apiserver
  path /var/log/audit
  pos_file /var/log/audit.pos
  format json
  time_key time
  time_format %Y-%m-%dT%H:%M:%S.%NZ
  tag audit
</source>

<filter audit>
  #https://github.com/fluent/fluent-plugin-rewrite-tag-filter/issues/13
  type record_transformer
  enable_ruby
  <record>
    namespace ${record["objectRef"].nil? ? "none":(record["objectRef"]["namespace"].nil?"":record["objectRef"]["namespace"])}
  </record>
</filter>

<match audit>
  # route audit according to namespace element in context
  @type rewrite_tag_filter
  rewriterule1 namespace ^(.+) ${tag}.$1
</match>

<filter audit.*>
  @type record_transformer
  remove_keys namespace
</filter>

<match audit.*>
  @type forest
  subtype file
  remove_prefix audit
  <template>
    time_slice_format %Y%m%d%H
    compress gz
    path /var/log/audit-${tag}.*.log
    format json
  </template>
</match>
```

```

        include_time_key true
    </template>
</match>
1. start fluentd
$ fluentd -c /etc/fluentd/config -vv
1. start kube-apiserver with the following options:
--audit-policy-file=/etc/kubernetes/audit-policy.yaml --audit-log-path=/var/log/kube-audit
1. check audits for different namespaces in /var/log/audit-*.*log

```

Use logstash to collect and distribute audit events from webhook backend

Logstash is an open source, server-side data processing tool. In this example, we will use logstash to collect audit events from webhook backend, and save events of different users into different files.

1. install logstash
2. create config file for logstash

```

$ cat <<EOF > /etc/logstash/config
input{
    http{
        #TODO, figure out a way to use kubeconfig file to authenticate to logstash
        #https://www.elastic.co/guide/en/logstash/current/plugins-inputs-http.html#plugin
        port=>8888
    }
}
filter{
    split{
        # Webhook audit backend sends several events together with EventList
        # split each event here.
        field=>[items]
        # We only need event subelement, remove others.
        remove_field=>[headers, metadata, apiVersion, "@timestamp", kind, "@version", host]
    }
    mutate{
        rename => {items=>event}
    }
}
output{
    file{
        # Audit events from different users will be saved into different files.
        path=>"/var/log/kube-audit-%{[event][user]}[username]/audit"
    }
}

```

```

        }
    }

1. start logstash

$ bin/logstash -f /etc/logstash/config --path.settings /etc/logstash/

1. create a kubeconfig file for kube-apiserver webhook audit backend

$ cat <<EOF > /etc/kubernetes/audit-webhook-kubeconfig
apiVersion: v1
clusters:
- cluster:
    server: http://<ip_of_logstash>:8888
    name: logstash
contexts:
- context:
    cluster: logstash
    user: ""
    name: default-context
current-context: default-context
kind: Config
preferences: {}
users: []
EOF

1. start kube-apiserver with the following options:

--audit-policy-file=/etc/kubernetes/audit-policy.yaml --audit-webhook-config-file=/etc/ku
1. check audits in logstash node's directories /var/log/kube-audit-*/*audit
```

Note that in addition to file output plugin, logstash has a variety of outputs that let users route data where they want. For example, users can emit audit events to elasticsearch plugin which supports full-text search and analytics.

Legacy Audit

Note: Legacy Audit is deprecated and is disabled by default since 1.8 and will be removed in 1.12. To fallback to this legacy audit, disable the advanced auditing feature using the `AdvancedAuditing` feature gate in kube-apiserver:

```
--feature-gates=AdvancedAuditing=false
```

In legacy format, each audit log entry contains two lines:

1. The request line containing a unique ID to match the response and request metadata, such as the source IP, requesting user, impersonation information, resource being requested, etc.

2. The response line containing a unique ID matching the request line and the response code.

Example output for `admin` user listing pods in the `default` namespace:

```
2017-03-21T03:57:09.106841886-04:00 AUDIT: id="c939d2a7-1c37-4ef1-b2f7-4ba9b1e43b53" ip="127.0.0.1" pod="nginx-7455555555-5qj7t" container="nginx" type="AuditLog" level="Info" audit_type="PodList" audit_id="c939d2a7-1c37-4ef1-b2f7-4ba9b1e43b53" audit_ns="default" audit_op="List" audit_time="2017-03-21T03:57:09.106841886Z" audit_version="1.7.0" audit_user="admin" audit_group="system:authenticated" audit_ip="127.0.0.1" audit_container="nginx" audit_pod="nginx-7455555555-5qj7t" audit_namespace="default" audit_request_id="5939d2a7-1c37-4ef1-b2f7-4ba9b1e43b53" audit_resource="pods" audit_resource_version="1" audit_size="1" audit_start_time="2017-03-21T03:57:09.106841886Z" audit_end_time="2017-03-21T03:57:09.108403639Z" audit_duration="1.563ms" audit_log_id="c939d2a7-1c37-4ef1-b2f7-4ba9b1e43b53" audit_log_ns="default" audit_log_op="List" audit_log_time="2017-03-21T03:57:09.108403639Z" audit_log_version="1.7.0" audit_log_user="admin" audit_log_group="system:authenticated" audit_log_ip="127.0.0.1" audit_log_container="nginx" audit_log_pod="nginx-7455555555-5qj7t" audit_log_namespace="default" audit_log_request_id="5939d2a7-1c37-4ef1-b2f7-4ba9b1e43b53" audit_log_resource="pods" audit_log_resource_version="1" audit_log_size="1" audit_log_start_time="2017-03-21T03:57:09.106841886Z" audit_log_end_time="2017-03-21T03:57:09.108403639Z" audit_log_duration="1.563ms"
```

Configuration

Kube-apiserver provides the following options which are responsible for configuring where and how audit logs are handled:

- `audit-log-path` - enables the audit log pointing to a file where the requests are being logged to, ‘-’ means standard out.
- `audit-log-maxage` - specifies maximum number of days to retain old audit log files based on the timestamp encoded in their filename.
- `audit-log-maxbackup` - specifies maximum number of old audit log files to retain.
- `audit-log-maxsize` - specifies maximum size in megabytes of the audit log file before it gets rotated. Defaults to 100MB.

If an audit log file already exists, Kubernetes appends new audit logs to that file. Otherwise, Kubernetes creates an audit log file at the location you specified in `audit-log-path`. If the audit log file exceeds the size you specify in `audit-log-maxsize`, Kubernetes will rename the current log file by appending the current timestamp on the file name (before the file extension) and create a new audit log file. Kubernetes may delete old log files when creating a new log file; you can configure how many files are retained and how old they can be by specifying the `audit-log-maxbackup` and `audit-log-maxage` options.

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

[Edit This Page](#)

Core metrics pipeline

Starting from Kubernetes 1.8, 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
- 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.

Metrics Server

Metrics Server is a cluster-wide aggregator of resource usage data. Starting from Kubernetes 1.8 it's 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. It's supported in Kubernetes 1.7+ (see details below).

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, which was introduced in Kubernetes 1.7.

Learn more about the metrics server in the design doc.

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

[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.

Status	Meaning
<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.

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

[Edit This Page](#)

Debug Pods and Replication Controllers

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.

- Debugging Replication Controllers

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 `kubectl 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 | grep '\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 that `-c ${CONTAINER_NAME}` is optional and can be omitted 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.

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 inspect events related to the replication controller.

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

[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 busybox Pod:

```
$ kubectl run -it --rm --restart=Never busybox --image=busybox sh  
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 "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 that this is the same as if you had 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 -qO- hostnames
wget: bad address 'hostname'
```

So the first thing to check is whether that `Service` actually exists:

```
$ kubectl get svc hostnames
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=9376
service "hostnames" exposed
```

And read it back, just to be sure:

```
$ kubectl get svc hostnames
NAME      CLUSTER-IP   EXTERNAL-IP   PORT(S)   AGE
hostnames  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-0utan

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",
        "selfLink": "/api/v1/namespaces/default/services/hostnames",
        "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 that 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 ? S1 Jul04 25:43 /usr/local/bin/kube-proxy --master=https://192.168.1.1:8443
```

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 a
I1027 22:14:54.038140      5063 proxier.go:352] Setting endpoints for "kube-system/kube-dns:dn
I1027 22:14:54.038164      5063 proxier.go:352] Setting endpoints for "kube-system/kube-dns:dn
I1027 22:14:54.038209      5063 proxier.go:352] Setting endpoints for "default/kubernetes:htt
I1027 22:14:54.038238      5063 proxier.go:429] Not syncing iptables until Services and Endpo
I1027 22:14:54.040048      5063 proxier.go:294] Adding new service "default/kubernetes:https"
I1027 22:14:54.040154      5063 proxier.go:294] Adding new service "kube-system/kube-dns:dns"
I1027 22:14:54.040223      5063 proxier.go:294] Adding new service "kube-system/kube-dns:dns-t
```

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:de
-A KUBE-PORTALS-HOST -d 10.0.1.175/32 -p tcp -m comment --comment "default/hostnames:default"
```

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 M
-A KUBE-SEP-57KPRZ3JQVENLNBR -p tcp -m comment --comment "default/hostnames:" -m tcp -j DNAT
-A KUBE-SEP-WNBA2IHDGP2BOBGZ -s 10.244.1.7/32 -m comment --comment "default/hostnames:" -j M
-A KUBE-SEP-WNBA2IHDGP2BOBGZ -p tcp -m comment --comment "default/hostnames:" -m tcp -j DNAT
-A KUBE-SEP-X3P2623AGDH6CDF3 -s 10.244.2.3/32 -m comment --comment "default/hostnames:" -j M
-A KUBE-SEP-X3P2623AGDH6CDF3 -p tcp -m comment --comment "default/hostnames:" -m tcp -j DNAT
-A KUBE-SERVICES -d 10.0.1.175/32 -p tcp -m comment --comment "default/hostnames: cluster IP"
-A KUBE-SVC-NWV5X2332I40T4T3 -m comment --comment "default/hostnames:" -m statistic --mode r
-A KUBE-SVC-NWV5X2332I40T4T3 -m comment --comment "default/hostnames:" -m statistic --mode r
-A KUBE-SVC-NWV5X2332I40T4T3 -m comment --comment "default/hostnames:" -j KUBE-SEP-57KPRZ3JQ
```

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 ?          S1    00:51 11:11 /usr/local/bin/kubelet --er
```

- 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 email or GitHub.

What's next

Visit troubleshooting document for more information.

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

[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.

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

[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.

```
termination.yaml docs/tasks/debug-application-cluster
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 create -f https://k8s.io/docs/tasks/debug-application-cluster/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 --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}}{{.}}
```

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 use 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.

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

[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](#).

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

[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][sdLogMetrics] and [alerts][sdAlerts], 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 create -f https://k8s.io/docs/tasks/debug-application-cluster/event-exporter-deploy
```

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.

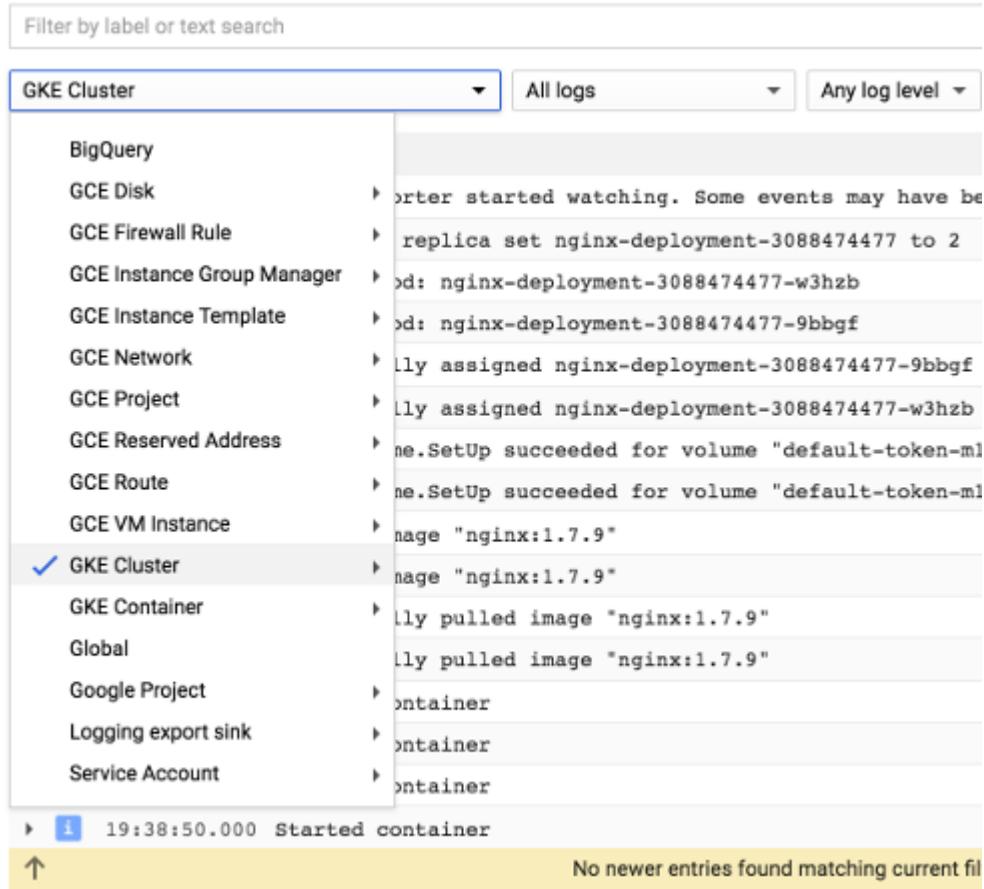
```
event-exporter-deploy.yaml docs/tasks/debug-application-cluster
```

```
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-v0.1.0
  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:v0.1.0
          command: [605, '/event-exporter']
          terminationGracePeriodSeconds: 30
```

```
event-exporter-deploy.yaml docs/tasks/debug-application-cluster
```

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, three dropdown menus are visible: "GKE Cluster" (which is highlighted with a blue border), "All logs", and "Any log level". The main area displays a list of resources on the left and their corresponding log entries on the right. The "GKE Cluster" resource is selected, as indicated by a checkmark icon next to its name. Other listed resources include BigQuery, GCE Disk, GCE Firewall Rule, GCE Instance Group Manager, GCE Instance Template, GCE Network, GCE Project, GCE Reserved Address, GCE Route, GCE VM Instance, GKE Container, Global, Google Project, Logging export sink, and Service Account. The log entries for the GKE Cluster resource show various deployment events, such as pods being created and containers starting. A message at the bottom indicates "No newer entries found matching current filter".

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



- ▶ 1 19:38:41.000 Successfully assigned nginx-deployment-3088474477-9bbgf
- ▶ 1 19:38:41.000 Successfully assigned nginx-deployment-3088474477-w3hzb



Figure 1:

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

[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:

```
shell-demo.yaml docs/tasks/debug-application-cluster
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
```

Create the Pod:

```
kubectl create -f https://k8s.io/docs/tasks/debug-application-cluster/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
```

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 `emptyDir` 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`

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

[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 that Elasticsearch and Kibana cannot be setup automatically in the Kubernetes cluster hosted on Google Kubernetes Engine, you have to deploy it 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 = 6987c098277871b6d69623141276924ab6)
+++ kubernetes-salt.tar.gz uploaded (sha1 = bdfc83ed6b60fa9e3bff9004b542cfcc643464cd0)
Looking for already existing resources
Starting master and configuring firewalls
Created [https://www.googleapis.com/compute/v1/projects/kubernetes-satnam/zones/us-central1-b]
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]
+++ 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.

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-11ds	1/1	Running	0	2h
fluentd-elasticsearch-kubernetes-node-1z9j	1/1	Running	0	2h
kibana-logging-v1-bhp08	1/1	Running	0	2h
kube-dns-v3-7r119	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:

```

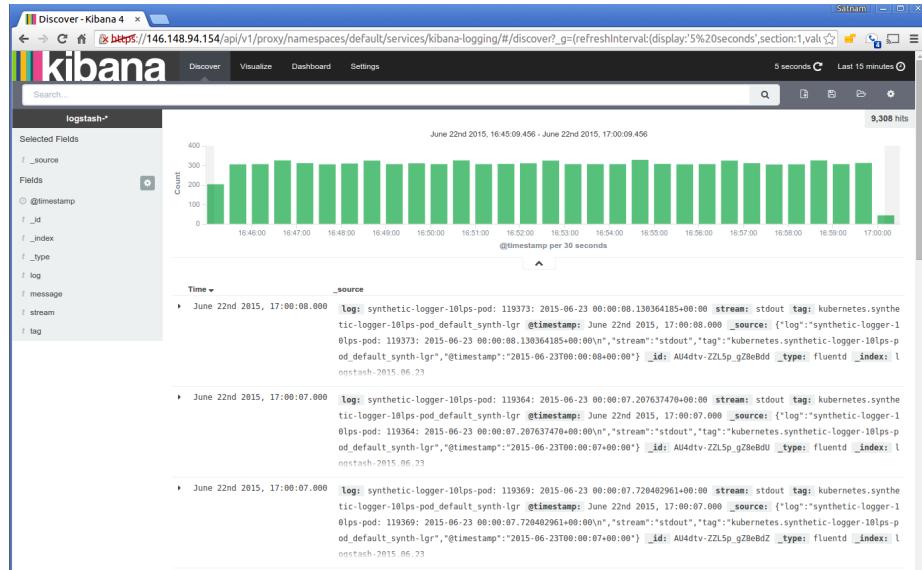
{
  "status": 200,
  "name": "Blitziana",
  "cluster_name": "kubernetes-logging",
  "version": {
    "number": "1.5.2",
    "build_hash": "62ff9868b4c8a0c45860bebb259e21980778ab1c",
    "build_timestamp": "2015-04-27T09:21:06Z",
    "build_snapshot": false,
    "lucene_version": "4.10.4"
  },
  "tagline": "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.

[Edit this Page](#)

[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 create -f https://k8s.io/docs/tasks/debug-application-cluster/fluentd-gcp-confi
```

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 create -f https://k8s.io/docs/tasks/debug-application-cluster/fluentd-gcp-ds.ya
```

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	READY	NODE-SELECTOR
...					
default	fluentd-gcp-v2.0	3	3	3	beta.kubernetes.io/fluentd-ds-
...					

To understand how logging with Stackdriver works, consider the following synthetic log generator pod specification counter-pod.yaml:

```
counter-pod.yaml docs/tasks/debug-application-cluster
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 date once per second, and runs indefinitely. Let's create this pod in the default namespace.

```
kubectl create -f https://k8s.io/docs/tasks/debug-application-cluster/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/docs/tasks/debug-application-cluster/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 |  
...  
"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, that 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 `DaemonSet` 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 for 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.

[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`:

```
node-problem-detector.yaml docs/tasks/debug-application-cluster
```

```
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/
```

```
node-problem-detector.yaml docs/tasks/debug-application-cluster
```

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 create -f https://k8s.io/docs/tasks/debug-application-cluster/node-problem-detector
```

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:

```
node-problem-detector-configmap.yaml
```

```
docs/tasks/debug-application-cluster
```

```
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: 624
              path: /var/log/
            - name: config # Define ConfigMap volume
              configMap:
                name: node-problem-detector-config
```

```
node-problem-detector-configmap.yaml
```

```
docs/tasks/debug-application-cluster
```

- **Step 4:** Re-create the node problem detector with the new yaml file:

```
kubectl delete -f https://k8s.io/docs/tasks/debug-application-cluster/node-problem-detector
kubectl create -f https://k8s.io/docs/tasks/debug-application-cluster/node-problem-detector
```

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)

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

[Edit This Page](#)

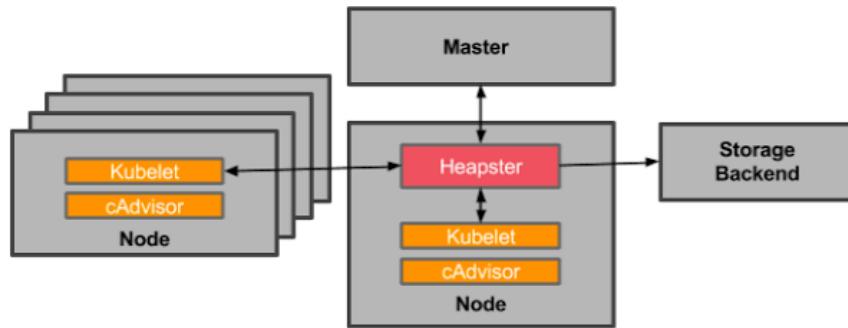
Tools for Monitoring Compute, Storage, and Network Resources

Understanding how an application behaves when deployed is crucial to scaling the application and providing a reliable service. In a Kubernetes cluster, application performance can be examined at many different levels: containers, pods, services, and whole clusters. As part of Kubernetes we want to provide users with detailed resource usage information about their running applications at all these levels. This will give users deep insights into how their applications are performing and where possible application bottlenecks may be found. In comes Heapster, a project meant to provide a base monitoring platform on Kubernetes.

- Storage Backends
- What's next

Heapster is a cluster-wide aggregator of monitoring and event data. It currently supports Kubernetes natively and works on all Kubernetes setups. Heapster

runs as a pod in the cluster, similar to how any Kubernetes application would run. The Heapster pod discovers all nodes in the cluster and queries usage information from the nodes' Kubelets, the on-machine Kubernetes agent. The Kubelet itself fetches the data from cAdvisor. Heapster groups the information by pod along with the relevant labels. This data is then pushed to a configurable backend for storage and visualization. Currently supported backends include InfluxDB (with Grafana for visualization), Google Cloud Monitoring and many others described in more details here. The overall architecture of the service can be seen below:



Let's look at some of the other components in more detail.

cAdvisor

cAdvisor is an open source container resource usage and performance analysis agent. It is purpose-built for containers and supports Docker containers natively. In Kubernetes, cAdvisor is integrated into the Kubelet binary. cAdvisor auto-discovers all containers in the machine and collects CPU, memory, filesystem, and network usage statistics. cAdvisor also provides the overall machine usage by analyzing the 'root' container on the machine.

On most Kubernetes clusters, cAdvisor exposes a simple UI for on-machine containers on port 4194. Here is a snapshot of part of cAdvisor's UI that shows the overall machine usage:



Kubelet

The Kubelet acts as a bridge between the Kubernetes master and the nodes. It manages the pods and containers running on a machine. Kubelet translates each pod into its constituent containers and fetches individual container usage statistics from cAdvisor. It then exposes the aggregated pod resource usage statistics via a REST API.

Storage Backends

InfluxDB and Grafana

A Grafana setup with InfluxDB is a very popular combination for monitoring in the open source world. InfluxDB exposes an easy to use API to write and fetch time series data. Heapster is setup to use this storage backend by default on most Kubernetes clusters. A detailed setup guide can be found [here](#). InfluxDB and Grafana run in Pods. The pod exposes itself as a Kubernetes service which is how Heapster discovers it.

The Grafana container serves Grafana's UI which provides an easy to configure dashboard interface. The default dashboard for Kubernetes contains an example dashboard that monitors resource usage of the cluster and the pods inside of it. This dashboard can easily be customized and expanded. Take a look at the storage schema for InfluxDB [here](#).

Here is a video showing how to monitor a Kubernetes cluster using heapster, InfluxDB and Grafana:



Here is a snapshot of the default Kubernetes Grafana dashboard that shows the CPU and Memory usage of the entire cluster, individual pods and containers:



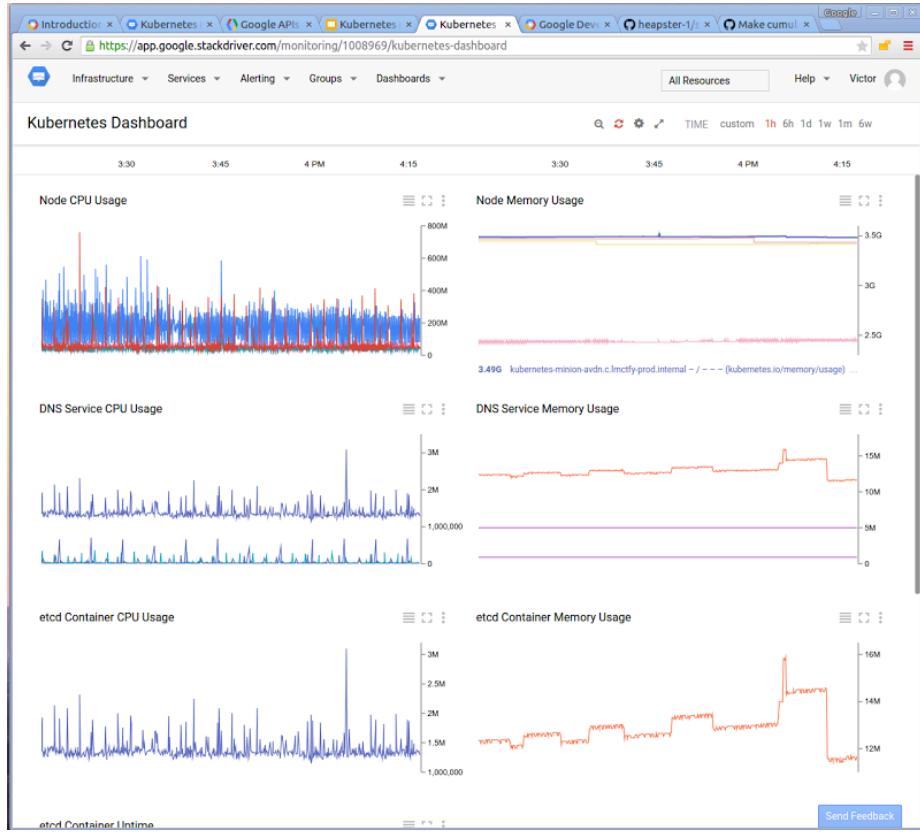
Google Cloud Monitoring

Google Cloud Monitoring is a hosted monitoring service that allows you to visualize and alert on important metrics in your application. Heapster can be setup to automatically push all collected metrics to Google Cloud Monitoring. These metrics are then available in the Cloud Monitoring Console. This storage backend is the easiest to setup and maintain. The monitoring console allows you to easily create and customize dashboards using the exported data.

Here is a video showing how to setup and run a Google Cloud Monitoring backed Heapster:



Here is a snapshot of the Google Cloud Monitoring dashboard showing cluster-wide resource usage.



What's next

Now that you've learned a bit about Heapster, feel free to try it out on your own clusters! The Heapster repository is available on GitHub. It contains detailed instructions to setup Heapster and its storage backends. Heapster runs by default on most Kubernetes clusters, so you may already have it! Feedback is always welcome. Please let us know if you run into any issues via the troubleshooting channels.

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

[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 `kubectl 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 that `-c ${CONTAINER_NAME}` is optional and can be omitted 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 create --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 false alarm, see https://github.com/kubernetes/kubernetes/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 `containerPort`

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.

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

[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 (experimental) high-availability configuration
 - Mitigates: Master VM shutdown or master components (scheduler, API server, controller-managing) crashing
 - Will tolerate one or more simultaneous node or component failures
 - Mitigates: Apiserver backing storage (i.e., etcd's data directory) lost
 - Assuming you used clustered etcd.
- 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.

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

[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`
- 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`
- Turkey: `#tr-users`, `#tr-events`

Mailing List

The Kubernetes / Google Kubernetes Engine mailing list is `kubernetes-users@googlegroups.com`

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

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

[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`.

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.

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

[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
- Enable 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.

Enable apiserver flags

Enable the aggregation layer via the following kube-apiserver flags. They may have already been taken care of by your provider.

```
--requestheader-client-ca-file=<path to aggregator CA cert>
--requestheader-allowed-names=aggregator
--requestheader-extra-headers-prefix=X-Remote-Extra-
--requestheader-group-headers=X-Remote-Group
--requestheader-username-headers=X-Remote-User
--proxy-client-cert-file=<path to aggregator proxy cert>
--proxy-client-key-file=<path to aggregator proxy key>
```

If you are not running kube-proxy on a host running the API server, then you must make sure that the system is enabled with the following apiserver flag:

```
--enable-aggregator-routing=true
```

What's next

- Setup an extension api-server to work with the aggregation layer.
- 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.

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

[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
- 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 1.7.0 or higher.
- Read about custom resources.

Create a CustomResourceDefinition

When you create a new CustomResourceDefinition (CRD), the Kubernetes API Server reacts by creating a new RESTful resource path, 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`:

```
apiVersion: apiextensions.k8s.io/v1beta1
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
  # version name to use for REST API: /apis/<group>/<version>
  version: v1
  # 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

```

And create it:

```
kubectl create -f resourcedefinition.yaml
```

Then a new namespaced RESTful API endpoint is created at:

```
/apis/stable.example.com/v1/namespaces/*/crontabs/...
```

This endpoint URL can then be used to create and manage custom objects. The `kind` of these objects will be `CronTab` from the spec of the `CustomResourceDefinition` object you created above.

Please note that it might take a few seconds for the endpoint to be created. You can watch the `Established` condition of your `CustomResourceDefinition` to be true or watch the discovery information of the API server for your resource to show up.

Create custom objects

After the `CustomResourceDefinition` object has been created, you can create custom objects. Custom objects can contain custom fields. These fields can contain arbitrary JSON. In the following example, the `cronSpec` and `image` custom fields are set in a custom object of kind `CronTab`. The kind `CronTab` comes from the spec of the `CustomResourceDefinition` object you created above.

If you save the following YAML to `my-crontab.yaml`:

```

apiVersion: "stable.example.com/v1"
kind: CronTab
metadata:
  name: my-new-cron-object
spec:
  cronSpec: "* * * * */5"
  image: my-awesome-cron-image

```

and create it:

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

You can then manage your CronTab objects using kubectl. For example:

```
kubectl get crontab
```

Should print a list like this:

NAME	AGE
my-new-cron-object	6s

Note that resource names are not case-sensitive when using kubectl, and you can use either the singular or plural forms defined in the CRD, as well as any short names.

You can also view the raw YAML data:

```
kubectl get ct -o yaml
```

You should see that it contains the custom `cronSpec` and `image` fields from the yaml you used to create it:

```
apiVersion: v1
items:
- apiVersion: stable.example.com/v1
  kind: CronTab
  metadata:
    clusterName: ""
    creationTimestamp: 2017-05-31T12:56:35Z
    deletionGracePeriodSeconds: null
    deletionTimestamp: null
    name: my-new-cron-object
    namespace: default
    resourceVersion: "285"
    selfLink: /apis/stable.example.com/v1/namespaces/default/crontabs/my-new-cron-object
    uid: 9423255b-4600-11e7-af6a-28d2447dc82b
  spec:
    cronSpec: '* * * * */5'
    image: my-awesome-cron-image
kind: List
metadata:
  resourceVersion: ""
  selfLink: ""
```

Delete a CustomResourceDefinition

When you delete a CustomResourceDefinition, the server will uninstall the RESTful API endpoint and **delete all custom objects stored in it**.

```
kubectl delete -f resourcedefinition.yaml
kubectl get crontabs
```

```
Error from server (NotFound): Unable to list "crontabs": the server could not find the requested resource.
```

If you later recreate the same CustomResourceDefinition, it will start out empty.

Advanced topics

Finalizers

Finalizers allow controllers to implement asynchronous pre-delete hooks. Custom objects support finalizers just like built-in objects.

You can add a finalizer to a custom object like this:

```
apiVersion: "stable.example.com/v1"
kind: CronTab
metadata:
  finalizers:
    - finalizer.stable.example.com
```

Finalizers are arbitrary string values, that when present ensure that a hard delete of a resource is not possible while they exist.

The first delete request on an object with finalizers merely sets a value for the `metadata.deletionTimestamp` field instead of deleting it. Once this value is set, entries in the `finalizer` list can only be removed.

This triggers controllers watching the object to execute any finalizers they handle. This will be represented via polling update requests for that object, until all finalizers have been removed and the resource is deleted.

The time period of polling update can be controlled by `metadata.deletionGracePeriodSeconds`.

It is the responsibility of each controller to removes its finalizer from the list.

Kubernetes will only finally delete the object if the list of finalizers is empty, meaning all finalizers are done.

Validation

Validation of custom objects is possible via OpenAPI v3 schema. Additionally, the following restrictions are applied to the schema:

- The fields `default`, `nullable`, `discriminator`, `readOnly`, `writeOnly`, `xml`, `deprecated` and `$ref` cannot be set.
- The field `uniqueItems` cannot be set to true.
- The field `additionalProperties` cannot be set to false.

This feature is `beta` in v1.9. You can disable this feature using the `CustomResourceValidation` feature gate on the kube-apiserver:

```
--feature-gates=CustomResourceValidation=false
```

The schema is defined in the CustomResourceDefinition. In the following example, the CustomResourceDefinition applies the following validations on the custom object:

- `spec.cronSpec` must be a string and must be of the form described by the regular expression.
- `spec.replicas` must be an integer and must have a minimum value of 1 and a maximum value of 10.

Save the CustomResourceDefinition to `resourcedefinition.yaml`:

```
apiVersion: apiextensions.k8s.io/v1beta1
kind: CustomResourceDefinition
metadata:
  name: crontabs.stable.example.com
spec:
  group: stable.example.com
  version: v1
  scope: Namespaced
  names:
    plural: crontabs
    singular: crontab
    kind: CronTab
    shortNames:
      - ct
  validation:
    # openAPIV3Schema is the schema for validating custom objects.
    openAPIV3Schema:
      properties:
        spec:
          properties:
            cronSpec:
              type: string
              pattern: '^(\d+|*)(/(\d+)?(\s+(\d+|*)(/(\d+)?){4})$'
            replicas:
              type: integer
              minimum: 1
              maximum: 10
```

And create it:

```
kubectl create -f resourcedefinition.yaml
```

A request to create a custom object of kind `CronTab` will be rejected if there are invalid values in its fields. In the following example, the custom object contains fields with invalid values:

- `spec.cronSpec` does not match the regular expression.

- `spec.replicas` is greater than 10.

If you save the following YAML to `my-crontab.yaml`:

```
apiVersion: "stable.example.com/v1"
kind: CronTab
metadata:
  name: my-new-cron-object
spec:
  cronSpec: "* * * *"
  image: my-awesome-cron-image
  replicas: 15
```

and create it:

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

you will get an error:

```
The CronTab "my-new-cron-object" is invalid: []: Invalid value: map[string]interface {}{"ap
validation failure list:
spec.cronSpec in body should match '^(\d+|\*)(/d+)?(\s+(\d+|\*)(/d+)?){4}$'
spec.replicas in body should be less than or equal to 10
```

If the fields contain valid values, the object creation request is accepted.

Save the following YAML to `my-crontab.yaml`:

```
apiVersion: "stable.example.com/v1"
kind: CronTab
metadata:
  name: my-new-cron-object
spec:
  cronSpec: "* * * */5"
  image: my-awesome-cron-image
  replicas: 5
```

And create it:

```
kubectl create -f my-crontab.yaml
crontab "my-new-cron-object" created
```

Subresources

Custom resources support `/status` and `/scale` subresources. This feature is **alpha** in v1.10 and may change in backward incompatible ways.

Enable this feature using the `CustomResourceSubresources` feature gate on the kube-apiserver:

```
--feature-gates=CustomResourceSubresources=true
```

When the `CustomResourceSubresources` feature gate is enabled, only the `properties` construct is allowed in the root schema for custom resource validation.

The status and scale subresources can be optionally enabled by defining them in the CustomResourceDefinition.

Status subresource

When the status subresource is enabled, the `/status` subresource for the custom resource is exposed.

- The status and the spec stanzas are represented by the `.status` and `.spec` JSONPaths respectively inside of a custom resource.
- PUT requests to the `/status` subresource take a custom resource object and ignore changes to anything except the status stanza.
- PUT requests to the `/status` subresource only validate the status stanza of the custom resource.
- PUT/POST/PATCH requests to the custom resource ignore changes to the status stanza.
- Any changes to the spec stanza increments the value at `.metadata.generation`.

Scale subresource

When the scale subresource is enabled, the `/scale` subresource for the custom resource is exposed. The `autoscaling/v1.Scale` object is sent as the payload for `/scale`.

To enable the scale subresource, the following values are defined in the CustomResourceDefinition.

- `SpecReplicasPath` defines the JSONPath inside of a custom resource that corresponds to `Scale.Spec.Replicas`.
 - It is a required value.
 - Only JSONPaths under `.spec` and with the dot notation are allowed.
 - If there is no value under the `SpecReplicasPath` in the custom resource, the `/scale` subresource will return an error on GET.
- `StatusReplicasPath` defines the JSONPath inside of a custom resource that corresponds to `Scale.Status.Replicas`.
 - It is a required value.
 - Only JSONPaths under `.status` and with the dot notation are allowed.
 - If there is no value under the `StatusReplicasPath` in the custom resource, the status replica value in the `/scale` subresource will default to 0.

- `LabelSelectorPath` defines the JSONPath inside of a custom resource that corresponds to `Scale.Status.Selector`.
 - It is an optional value.
 - It must be set to work with HPA.
 - Only JSONPaths under `.status` and with the dot notation are allowed.
 - If there is no value under the `LabelSelectorPath` in the custom resource, the status selector value in the `/scale` subresource will default to the empty string.

In the following example, both status and scale subresources are enabled.

Save the CustomResourceDefinition to `resourcedefinition.yaml`:

```
apiVersion: apiextensions.k8s.io/v1beta1
kind: CustomResourceDefinition
metadata:
  name: crontabs.stable.example.com
spec:
  group: stable.example.com
  version: v1
  scope: Namespaced
  names:
    plural: crontabs
    singular: crontab
    kind: CronTab
    shortNames:
      - ct
  # subresources describes the subresources for custom resources.
  subresources:
    # status enables the status subresource.
    status: {}
    # scale enables the scale subresource.
    scale:
      # specReplicasPath defines the JSONPath inside of a custom resource that corresponds to .spec.replicas
      specReplicasPath: .spec.replicas
      # statusReplicasPath defines the JSONPath inside of a custom resource that corresponds to .status.replicas
      statusReplicasPath: .status.replicas
      # labelSelectorPath defines the JSONPath inside of a custom resource that corresponds to .status.labelSelector
      labelSelectorPath: .status.labelSelector
```

And create it:

```
kubectl create -f resourcedefinition.yaml
```

After the CustomResourceDefinition object has been created, you can create custom objects.

If you save the following YAML to `my-crontab.yaml`:

```
apiVersion: "stable.example.com/v1"
kind: CronTab
metadata:
  name: my-new-cron-object
spec:
  cronSpec: "* * * * */5"
  image: my-awesome-cron-image
  replicas: 3
```

and create it:

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

Then new namespaced RESTful API endpoints are created at:

```
/apis/stable.example.com/v1/namespaces/*/crontabs/status
```

and

```
/apis/stable.example.com/v1/namespaces/*/crontabs/scale
```

A custom resource can be scaled using the `kubectl scale` command. For example, the following command sets `.spec.replicas` of the custom resource created above to 5:

```
kubectl scale --replicas=5 crontabs/my-new-cron-object
crontabs "my-new-cron-object" scaled
```

```
kubectl get crontabs my-new-cron-object -o jsonpath='{.spec.replicas}'
5
```

Categories

Categories is a list of grouped resources the custom resource belongs to (eg. `all`). You can use `kubectl get <category-name>` to list the resources belonging to the category. This feature is **beta** and available for custom resources from v1.10.

The following example adds `all` in the list of categories in the CustomResourceDefinition and illustrates how to output the custom resource using `kubectl get all`.

Save the following CustomResourceDefinition to `resourcedefinition.yaml`:

```
apiVersion: apiextensions.k8s.io/v1beta1
kind: CustomResourceDefinition
metadata:
  name: crontabs.stable.example.com
spec:
  group: stable.example.com
  version: v1
```

```

scope: Namespaced
names:
  plural: crontabs
  singular: crontab
  kind: CronTab
  shortNames:
    - ct
  # categories is a list of grouped resources the custom resource belongs to.
  categories:
    - all

```

And create it:

```
kubectl create -f resourcedefinition.yaml
```

After the CustomResourceDefinition object has been created, you can create custom objects.

Save the following YAML to `my-crontab.yaml`:

```

apiVersion: "stable.example.com/v1"
kind: CronTab
metadata:
  name: my-new-cron-object
spec:
  cronSpec: "* * * * */5"
  image: my-awesome-cron-image

```

and create it:

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

You can specify the category using `kubectl get`:

```
kubectl get all
```

and it will include the custom resources of kind `CronTab`:

NAME	AGE
crontabs/my-new-cron-object	3s

What's next

- Learn how to Migrate a ThirdPartyResource to CustomResourceDefinition.
- See CustomResourceDefinition.

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

[Edit This Page](#)

Migrate a ThirdPartyResource to CustomResourceDefinition

This page shows how to migrate data stored in a ThirdPartyResource (TPR) to a CustomResourceDefinition (CRD).

Kubernetes does not automatically migrate existing TPRs. This is due to API changes introduced as part of graduating to beta under a new name and API group. Instead, both TPR and CRD are available and operate independently in Kubernetes 1.7. Users must migrate each TPR one by one to preserve their data before upgrading to Kubernetes 1.8.

The simplest way to migrate is to stop all clients that use a given TPR, then delete the TPR and start from scratch with a CRD. This page describes an optional process that eases the transition by migrating existing TPR data for you **on a best-effort basis**.

- Before you begin
- Migrate TPR data
- 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 exactly 1.7.x** (any patch release), as this is the only version that supports both TPR and CRD.
- If you use a TPR-based custom controller, check with the author of the controller first. Some or all of these steps may be unnecessary if the custom controller handles the migration for you.
- Be familiar with the concept of custom resources, which were known as *third-party resources* until Kubernetes 1.7.
- Be familiar with CustomResourceDefinitions, which are a simple way to implement custom resources.
- **Before performing a migration on real data, conduct a dry run by going through these steps in a test cluster.**

Migrate TPR data

1. Rewrite the TPR definition

Clients that access the REST API for your custom resource should not need any changes. However, you will need to rewrite your TPR definition as a CRD.

Make sure you specify values for the CRD fields that match what the server used to fill in for you with TPR.

For example, if your ThirdPartyResource looks like this:

```
““yaml apiVersion: extensions/v1beta1 kind: ThirdPartyResource metadata: name: cron-tab.stable.example.com description: “A specification of a Pod to run on a cron style schedule” versions:
```

- name: v1 ““

A matching CustomResourceDefinition could look like this:

```
apiVersion: apiextensions.k8s.io/v1beta1
kind: CustomResourceDefinition
metadata:
  name: crontabs.stable.example.com
spec:
  scope: Namespaced
  group: stable.example.com
  version: v1
  names:
    kind: CronTab
    plural: crontabs
    singular: crontab
```

2. Install the CustomResourceDefinition

While the source TPR is still active, install the matching CRD with `kubectl create`. Existing TPR data remains accessible because TPRs take precedence over CRDs when both try to serve the same resource.

After you create the CRD, make sure the *Established* condition goes to True. You can check it with a command like this:

```
kubectl get crd -o 'custom-columns=NAME:{.metadata.name},ESTABLISHED:{.status.condition
```

The output should look like this:

NAME	ESTABLISHED
crontabs.stable.example.com	True

3. Stop all clients that use the TPR

The API server attempts to prevent TPR data for the resource from changing while it copies objects to the CRD, but it can't guarantee consistency in all cases, such as with multiple masters. Stopping clients, such as TPR-based custom controllers, helps to avoid inconsistencies in the copied data.

In addition, clients that watch TPR data do not receive any more events once the migration begins. You must restart them after the migration completes so they start watching CRD data instead.

4. Back up TPR data

In case the data migration fails, save a copy of existing data for the resource:

```
kubectl get crontabs --all-namespaces -o yaml > crontabs.yaml
```

You should also save a copy of the TPR definition if you don't have one already:

```
kubectl get thirdpartyresource cron-tab.stable.example.com -o yaml --export > tpr.yaml
```

5. Delete the TPR definition

Normally, when you delete a TPR definition, the API server tries to clean up any objects stored in that resource. Because a matching CRD exists, the server copies objects to the CRD instead of deleting them.

```
kubectl delete thirdpartyresource cron-tab.stable.example.com
```

6. Verify the new CRD data

It can take up to 10 seconds for the TPR controller to notice when you delete the TPR definition and to initiate the migration. The TPR data remains accessible during this time.

Once the migration completes, the resource begins serving through the CRD. Check that all your objects were correctly copied:

```
kubectl get crontabs --all-namespaces -o yaml
```

If the copy failed, you can quickly revert to the set of objects that existed just before the migration by recreating the TPR definition:

```
kubectl create -f tpr.yaml
```

7. Restart clients

After verifying the CRD data, restart any clients you stopped before the migration, such as custom controllers and other watchers. These clients now access CRD data when they make requests on the same API endpoints that the TPR previously served.

What's next

- Learn more about custom resources.
- Learn more about using CustomResourceDefinitions.
- See CustomResourceDefinition.

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

[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`.

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.

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

[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/
```

```
{
  "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
```

```
{
  "kind": "PodList",
  "apiVersion": "v1",
  "metadata": {
    "selfLink": "/api/v1/namespaces/default/pods",
    "resourceVersion": "33074"
  },
  "items": [
    {
      "metadata": {
        "name": "kubernetes-bootcamp-2321272333-ix8pt",
        "generateName": "kubernetes-bootcamp-2321272333-",
        "namespace": "default",
        "selfLink": "/api/v1/namespaces/default/pods/kubernetes-bootcamp-2321272333-ix8pt",
        "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.

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

[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
- Kubelet certificate rotation is beta in 1.8.0 which means it may change without notice.

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.

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

[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
- Kubelet certificate rotation is beta in 1.8.0 which means it may change without notice.

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.

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

[Edit This Page](#)

Manage TLS Certificates in a Cluster

Every Kubernetes cluster has a cluster root Certificate Authority (CA). The CA is generally used by cluster components to validate the API server's certificate, by the API server to validate kubelet client certificates, etc. To support this, the CA certificate bundle is distributed to every node in the cluster and is distributed as a secret attached to default service accounts. Optionally, your workloads can use this CA to establish trust. Your application can request a certificate signing using the `certificates.k8s.io` API using a protocol that is similar to the ACME draft.

- 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 cluster root 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 `Certificates` field in the `tls.Config` struct.

The CA certificate bundle is automatically mounted into pods using the default service account at the path `/var/run/secrets/kubernetes.io/serviceaccount/ca.crt`. If you are not using the default service account, ask a cluster administrator to build a configmap containing the certificate bundle that you 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",
        "172.168.0.24",
        "10.0.34.2"
    ],
    "CN": "my-pod.my-namespace.pod.cluster.local",
    "key": {
        "algo": "ecdsa",
        "size": 256
    }
}
EOF
```

Where 172.168.0.24 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 create -f -
apiVersion: certificates.k8s.io/v1beta1
kind: CertificateSigningRequest
metadata:
  name: my-svc.my-namespace
spec:
  groups:
```

```

- system:authenticated
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: 172.168.0.24
                           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           CONDITION
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 -d > 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 the cluster. This includes who the Kubernetes API trusts. 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. See here for information on how certificates interact with authentication.

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.

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

[Edit This Page](#)

Set up placement policies in Federation

Note: **Federation V1**, the current Kubernetes federation API which reuses the Kubernetes API resources ‘as is’, is currently considered alpha for many of its features, and there is no clear path to evolve the API to GA. However, there is a **Federation V2** effort in progress to implement a dedicated federation API apart from the Kubernetes API. The details can be found at sig-multicloud community page.

This page shows how to enforce policy-based placement decisions over Federated resources using an external policy engine.

- Before you begin
- Deploying Federation and configuring an external policy engine
- Deploying an external policy engine
- Configuring placement policies via ConfigMaps
- Testing placement policies

Before you begin

You need to have a running Kubernetes cluster (which is referenced as host cluster). Please see one of the getting started guides for installation instructions for your platform.

Deploying Federation and configuring an external policy engine

The Federation control plane can be deployed using `kubefed init`.

After deploying the Federation control plane, you must configure an Admission Controller in the Federation API server that enforces placement decisions received from the external policy engine.

```
kubectl create -f scheduling-policy-admission.yaml
```

Shown below is an example ConfigMap for the Admission Controller:

```
scheduling-policy-admission.yaml docs/tasks/federation
apiVersion: v1
kind: ConfigMap
metadata:
  name: admission
  namespace: federation-system
data:
  config.yml: |
    apiVersion: apiserver.k8s.io/v1alpha1
    kind: AdmissionConfiguration
    plugins:
      - name: SchedulingPolicy
        path: /etc/kubernetes/admission/scheduling-policy-config.yml
  scheduling-policy-config.yml: |
    kubeconfig: /etc/kubernetes/admission/opa-kubeconfig
    opa-kubeconfig: |
      clusters:
        - name: opa-api
          cluster:
            server: http://opa.federation-system.svc.cluster.local:8181/v0/data/kubernetes/p...
      users:
        - name: scheduling-policy
          user:
            token: deadbeefsecret
      contexts:
        - name: default
          context:
            cluster: opa-api
            user: scheduling-policy
    current-context: default
```

The ConfigMap contains three files:

- `config.yml` specifies the location of the `SchedulingPolicy` Admission Controller config file.
- `scheduling-policy-config.yml` specifies the location of the kubeconfig file required to contact the external policy engine. This file can also include a `retryBackoff` value that controls the initial retry backoff delay in milliseconds.
- `opa-kubeconfig` is a standard kubeconfig containing the URL and credentials needed to contact the external policy engine.

Edit the Federation API server deployment to enable the `SchedulingPolicy`

Admission Controller.

```
kubectl -n federation-system edit deployment federation-apiserver
```

Update the Federation API server command line arguments to enable the Admission Controller and mount the ConfigMap into the container. If there's an existing `--enable-admission-plugins` flag, append `,SchedulingPolicy` instead of adding another line.

```
--enable-admission-plugins=SchedulingPolicy
--admission-control-config-file=/etc/kubernetes/admission/config.yml
```

Add the following volume to the Federation API server pod:

```
- name: admission-config
  configMap:
    name: admission
```

Add the following volume mount the Federation API server `apiserver` container:

```
volumeMounts:
- name: admission-config
  mountPath: /etc/kubernetes/admission
```

Deploying an external policy engine

The Open Policy Agent (OPA) is an open source, general-purpose policy engine that you can use to enforce policy-based placement decisions in the Federation control plane.

Create a Service in the host cluster to contact the external policy engine:

```
kubectl create -f policy-engine-service.yaml
```

Shown below is an example Service for OPA.

```
policy-engine-service.yaml docs/tasks/federation
```

```
kind: Service
apiVersion: v1
metadata:
  name: opa
  namespace: federation-system
spec:
  selector:
    app: opa
  ports:
  - name: http
    protocol: TCP
    port: 8181
    targetPort: 8181
```

Create a Deployment in the host cluster with the Federation control plane:

```
kubectl create -f policy-engine-deployment.yaml
```

Shown below is an example Deployment for OPA.

```
policy-engine-deployment.yaml docs/tasks/federation
apiVersion: apps/v1
kind: Deployment
metadata:
  labels:
    app: opa
  name: opa
  namespace: federation-system
spec:
  replicas: 1
  template:
    metadata:
      labels:
        app: opa
      name: opa
    spec:
      containers:
        - name: opa
          image: openpolicyagent/opa:0.4.10
          args:
            - "run"
            - "--server"
        - name: kube-mgmt
          image: openpolicyagent/kube-mgmt:0.2
          args:
            - "-kubeconfig=/srv/kubernetes/kubeconfig"
            - "-cluster=federation/v1beta1/clusters"
          volumeMounts:
            - name: federation-kubeconfig
              mountPath: /srv/kubernetes
              readOnly: true
      volumes:
        - name: federation-kubeconfig
          secret:
            secretName: federation-controller-manager-kubeconfig
```

Configuring placement policies via ConfigMaps

The external policy engine will discover placement policies created in the `kube-federation-scheduling-policy` namespace in the Federation API server.

Create the namespace if it does not already exist:

```
kubectl --context=federation create namespace kube-federation-scheduling-policy
```

Configure a sample policy to test the external policy engine:

```
policy.rego docs/tasks/federation

# OPA supports a high-level declarative language named Rego for authoring and
# enforcing policies. For more information on Rego, visit
# http://openpolicyagent.org.

# Rego policies are namespaced by the "package" directive.
package kubernetes.placement

# Imports provide aliases for data inside the policy engine. In this case, the
# policy simply refers to "clusters" below.
import data.kubernetes.clusters

# The "annotations" rule generates a JSON object containing the key
# "federation.kubernetes.io/replica-set-preferences" mapped to <preferences>.
# The preferences values is generated dynamically by OPA when it evaluates the
# rule.
#
# The SchedulingPolicy Admission Controller running inside the Federation API
# server will merge these annotations into incoming Federated resources. By
# setting replica-set-preferences, we can control the placement of Federated
# ReplicaSets.
#
# Rules are defined to generate JSON values (booleans, strings, objects, etc.)
# When OPA evaluates a rule, it generates a value IF all of the expressions in
# the body evaluate successfully. All rules can be understood intuitively as
# <head> if <body> where <body> is true if <expr-1> AND <expr-2> AND ...
# <expr-N> is true (for some set of data.)
annotations["federation.kubernetes.io/replica-set-preferences"] = preferences {
    input.kind = "ReplicaSet"
    value = {"clusters": cluster_map, "rebalance": true}
    json.marshal(value, preferences)
}

# This "annotations" rule generates a value for the "federation.alpha.kubernetes.io/cluster"
# annotation.
#
# In English, the policy asserts that resources in the "production" namespace
# that are not annotated with "criticality=low" MUST be placed on clusters
# labelled with "on-premises=true".
annotations["federation.alpha.kubernetes.io/cluster-selector"] = selector {
    input.metadata.namespace = "production"
    not input.metadata.annotations.criticality = "low"
    json.marshal([
        "operator": "=",
        "key": "on-premises",
        "values": "[true]",       678
    ], selector)
}

# Generates a set of cluster names that satisfy the incoming Federated
# ReplicaSet's requirements. In this case, just PCI compliance.
replica_set_clusters[cluster_name] {
    clusters[cluster_name]
    not insufficient_pci[cluster_name]
```

Shown below is the command to create the sample policy:

```
kubectl --context=federation -n kube-federation-scheduling-policy create configmap scheduling-policy --from-file=scheduling-policy.yaml
```

This sample policy illustrates a few key ideas:

- Placement policies can refer to any field in Federated resources.
- Placement policies can leverage external context (for example, Cluster metadata) to make decisions.
- Administrative policy can be managed centrally.
- Policies can define simple interfaces (such as the `requires-pci` annotation) to avoid duplicating logic in manifests.

Testing placement policies

Annotate one of the clusters to indicate that it is PCI certified.

```
kubectl --context=federation annotate clusters cluster-name-1 pci-certified=true
```

Deploy a Federated ReplicaSet to test the placement policy.

```
replicaset-example-policy.yaml docs/tasks/federation
apiVersion: apps/v1
kind: ReplicaSet
metadata:
  labels:
    app: nginx-pci
  name: nginx-pci
  annotations:
    requires-pci: "true"
spec:
  replicas: 3
  selector:
    matchLabels:
      app: nginx-pci
  template:
    metadata:
      labels:
        app: nginx-pci
    spec:
      containers:
        - image: nginx
          name: nginx-pci
```

Shown below is the command to deploy a ReplicaSet that *does* match the policy.

```
kubectl --context=federation create -f replicaset-example-policy.yaml
```

Inspect the ReplicaSet to confirm the appropriate annotations have been applied:

```
kubectl --context=federation get rs nginx-pci -o jsonpath='{.metadata.annotations}'
```

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

[Edit This Page](#)

Cross-cluster Service Discovery using Federated Services

Note: Federation V1, the current Kubernetes federation API which reuses the Kubernetes API resources ‘as is’, is currently considered alpha for many of

its features, and there is no clear path to evolve the API to GA. However, there is a **Federation V2** effort in progress to implement a dedicated federation API apart from the Kubernetes API. The details can be found at sig-multicloud community page.

This guide explains how to use Kubernetes Federated Services to deploy a common Service across multiple Kubernetes clusters. This makes it easy to achieve cross-cluster service discovery and availability zone fault tolerance for your Kubernetes applications.

Federated Services are created in much the same way as traditional Kubernetes Services by making an API call which specifies the desired properties of your service. In the case of Federated Services, this API call is directed to the Federation API endpoint, rather than a Kubernetes cluster API endpoint. The API for Federated Services is 100% compatible with the API for traditional Kubernetes Services.

Once created, the Federated Service automatically:

1. Creates matching Kubernetes Services in every cluster underlying your Cluster Federation,
2. Monitors the health of those service “shards” (and the clusters in which they reside), and
3. Manages a set of DNS records in a public DNS provider (like Google Cloud DNS, or AWS Route 53), thus ensuring that clients of your federated service can seamlessly locate an appropriate healthy service endpoint at all times, even in the event of cluster, availability zone or regional outages.

Clients inside your federated Kubernetes clusters (i.e. Pods) will automatically find the local shard of the Federated Service in their cluster if it exists and is healthy, or the closest healthy shard in a different cluster if it does not.

- Before you begin
- Prerequisites
- Hybrid cloud capabilities
- Creating a federated service
- Adding backend pods
- Verifying public DNS records
- Discovering a federated service
- Handling failures of backend pods and whole clusters
- Troubleshooting
- For more information

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 guide assumes that you have a running Kubernetes Cluster Federation installation. If not, then head over to the federation admin guide to learn how to bring up a cluster federation (or have your cluster administrator do this for you). Other tutorials, for example this one by Kelsey Hightower, are also available to help you.

You are also expected to have a basic working knowledge of Kubernetes in general, and Services in particular.

Hybrid cloud capabilities

Federations of Kubernetes Clusters can include clusters running in different cloud providers (e.g. Google Cloud, AWS), and on-premises (e.g. on OpenStack). Simply create all of the clusters that you require, in the appropriate cloud providers and/or locations, and register each cluster's API endpoint and credentials with your Federation API Server (See the federation admin guide for details).

Thereafter, your applications and services can span different clusters and cloud providers as described in more detail below.

Creating a federated service

This is done in the usual way, for example:

```
kubectl --context=federation-cluster create -f services/nginx.yaml
```

The ‘`--context=federation-cluster`’ flag tells `kubectl` to submit the request to the Federation API endpoint, with the appropriate credentials. If you have not yet configured such a context, visit the federation admin guide or one of the administration tutorials to find out how to do so.

As described above, the Federated Service will automatically create and maintain matching Kubernetes services in all of the clusters underlying your federation.

You can verify this by checking in each of the underlying clusters, for example:

```
kubectl --context=gce-asia-east1a get services nginx
NAME      CLUSTER-IP      EXTERNAL-IP      PORT(S)      AGE
nginx    10.63.250.98    104.199.136.89    80/TCP      9m
```

The above assumes that you have a context named ‘gce-asia-east1a’ configured in your client for your cluster in that zone. The name and namespace of the underlying services will automatically match those of the Federated Service that you created above (and if you happen to have had services of the same name and namespace already existing in any of those clusters, they will be automatically adopted by the Federation and updated to conform with the specification of your Federated Service - either way, the end result will be the same).

The status of your Federated Service will automatically reflect the real-time status of the underlying Kubernetes services, for example:

```
$kubectl --context=federation-cluster describe services nginx
```

Name:	nginx
Namespace:	default
Labels:	run=nginx
Annotations:	<none>
Selector:	run=nginx
Type:	LoadBalancer
IP:	10.63.250.98
LoadBalancer Ingress:	104.197.246.190, 130.211.57.243, 104.196.14.231, 104.199.136.89, ...
Port:	http 80/TCP
Endpoints:	<none>
Session Affinity:	None
Events:	<none>

Note the ‘LoadBalancer Ingress’ addresses of your Federated Service correspond with the ‘LoadBalancer Ingress’ addresses of all of the underlying Kubernetes services (once these have been allocated - this may take a few seconds). For inter-cluster and inter-cloud-provider networking between service shards to work correctly, your services need to have an externally visible IP address. Service Type: Loadbalancer is typically used for this, although other options (e.g. External IP’s) exist.

Note also that we have not yet provisioned any backend Pods to receive the network traffic directed to these addresses (i.e. ‘Service Endpoints’), so the Federated Service does not yet consider these to be healthy service shards, and has accordingly not yet added their addresses to the DNS records for this Federated Service (more on this aspect later).

Adding backend pods

To render the underlying service shards healthy, we need to add backend Pods behind them. This is currently done directly against the API endpoints of the underlying clusters (although in future the Federation server will be able to do all this for you with a single command, to save you the trouble). For example, to create backend Pods in 13 underlying clusters:

```
for CLUSTER in asia-east1-c asia-east1-a asia-east1-b \
              europe-west1-d europe-west1-c europe-west1-b \
              us-central1-f us-central1-a us-central1-b us-central1-c \
              us-east1-d us-east1-c us-east1-b
do
  kubectl --context=$CLUSTER run nginx --image=nginx:1.11.1-alpine --port=80
done
```

Note that `kubectl run` automatically adds the `run=nginx` labels required to associate the backend pods with their services.

Verifying public DNS records

Once the above Pods have successfully started and have begun listening for connections, Kubernetes will report them as healthy endpoints of the service in that cluster (via automatic health checks). The Cluster Federation will in turn consider each of these service ‘shards’ to be healthy, and place them in serving by automatically configuring corresponding public DNS records. You can use your preferred interface to your configured DNS provider to verify this. For example, if your Federation is configured to use Google Cloud DNS, and a managed DNS domain ‘example.com’:

```
$ gcloud dns managed-zones describe example-dot-com
creationTime: '2016-06-26T18:18:39.229Z'
description: Example domain for Kubernetes Cluster Federation
dnsName: example.com
id: '3229332181334243121'
kind: dns#managedZone
name: example-dot-com
nameServers:
- ns-cloud-a1.googledomains.com.
- ns-cloud-a2.googledomains.com.
- ns-cloud-a3.googledomains.com.
- ns-cloud-a4.googledomains.com.

$ gcloud dns record-sets list --zone example-dot-com
NAME                                     TYPE    TTL     DATA
example.com.                               NS      21600   ns-cloud-e
example.com.                               OA      21600   ns-cloud-e
```

nginxx.mynameSpace.myfederation.svc.example.com.	A	180	104.197.247.191
nginxx.mynameSpace.myfederation.svc.us-central1-a.example.com.	A	180	104.197.247.191
nginxx.mynameSpace.myfederation.svc.us-central1-b.example.com.	A	180	104.197.247.191
nginxx.mynameSpace.myfederation.svc.us-central1-c.example.com.	A	180	104.197.247.191
nginxx.mynameSpace.myfederation.svc.us-central1-f.example.com.	CNAME	180	nginxx.mynameSpace.myfederation.svc.us-central1-f.example.com.
nginxx.mynameSpace.myfederation.svc.us-central1.example.com.	A	180	104.197.247.191
nginxx.mynameSpace.myfederation.svc.asia-east1-a.example.com.	A	180	130.211.56.221
nginxx.mynameSpace.myfederation.svc.asia-east1-b.example.com.	CNAME	180	nginxx.mynameSpace.myfederation.svc.asia-east1-b.example.com.
nginxx.mynameSpace.myfederation.svc.asia-east1-c.example.com.	A	180	130.211.56.221
nginxx.mynameSpace.myfederation.svc.asia-east1.example.com.	A	180	130.211.56.221
nginxx.mynameSpace.myfederation.svc.europe-west1.example.com.	CNAME	180	nginxx.mynameSpace.myfederation.svc.europe-west1.example.com.
nginxx.mynameSpace.myfederation.svc.europe-west1-d.example.com.	CNAME	180	nginxx.mynameSpace.myfederation.svc.europe-west1-d.example.com.
... etc.			

Note: If your Federation is configured to use AWS Route53, you can use one of the equivalent AWS tools, for example:

```
$ aws route53 list-hosted-zones
```

and

```
$ aws route53 list-resource-record-sets --hosted-zone-id Z3ECL0L9QL0VBX
```

Whatever DNS provider you use, any DNS query tool (for example ‘dig’ or ‘nslookup’) will of course also allow you to see the records created by the Federation for you. Note that you should either point these tools directly at your DNS provider (e.g. dig @ns-cloud-e1.googledomains.com...) or expect delays in the order of your configured TTL (180 seconds, by default) before seeing updates, due to caching by intermediate DNS servers.

Some notes about the above example

1. Notice that there is a normal (‘A’) record for each service shard that has at least one healthy backend endpoint. For example, in us-central1-a, 104.197.247.191 is the external IP address of the service shard in that zone, and in asia-east1-a the address is 130.211.56.221.
2. Similarly, there are regional ‘A’ records which include all healthy shards in that region. For example, ‘us-central1’. These regional records are useful for clients which do not have a particular zone preference, and as a building block for the automated locality and failover mechanism described below.
3. For zones where there are currently no healthy backend endpoints, a CNAME (‘Canonical Name’) record is used to alias (automatically redirect) those queries to the next closest healthy zone. In the example, the service shard in us-central1-f currently has no healthy backend endpoints (i.e. Pods), so a CNAME record has been created to automatically redirect queries to other shards in that region (us-central1 in this case).

4. Similarly, if no healthy shards exist in the enclosing region, the search progresses further afield. In the europe-west1-d availability zone, there are no healthy backends, so queries are redirected to the broader europe-west1 region (which also has no healthy backends), and onward to the global set of healthy addresses ('nginx.mynameSpace.myfederation.svc.example.com').

The above set of DNS records is automatically kept in sync with the current state of health of all service shards globally by the Federated Service system. DNS resolver libraries (which are invoked by all clients) automatically traverse the hierarchy of ‘CNAME’ and ‘A’ records to return the correct set of healthy IP addresses. Clients can then select any one of the returned addresses to initiate a network connection (and fail over automatically to one of the other equivalent addresses if required).

Discovering a federated service

From pods inside your federated clusters

By default, Kubernetes clusters come pre-configured with a cluster-local DNS server (‘KubeDNS’), as well as an intelligently constructed DNS search path which together ensure that DNS queries like “myservice”, “myservice.mynameSpace”, “bobsservice.othernameSpace” etc issued by your software running inside Pods are automatically expanded and resolved correctly to the appropriate service IP of services running in the local cluster.

With the introduction of Federated Services and Cross-Cluster Service Discovery, this concept is extended to cover Kubernetes services running in any other cluster across your Cluster Federation, globally. To take advantage of this extended range, you use a slightly different DNS name (of the form “..”, e.g. myservice.mynameSpace.myfederation) to resolve Federated Services. Using a different DNS name also avoids having your existing applications accidentally traversing cross-zone or cross-region networks and you incurring perhaps unwanted network charges or latency, without you explicitly opting in to this behavior.

So, using our NGINX example service above, and the Federated Service DNS name form just described, let’s consider an example: A Pod in a cluster in the us-central1-f availability zone needs to contact our NGINX service. Rather than use the service’s traditional cluster-local DNS name (“nginx.mynameSpace”, which is automatically expanded to “nginx.mynameSpace.svc.cluster.local”) it can now use the service’s Federated DNS name, which is “nginx.mynameSpace.myfederation”. This will be automatically expanded and resolved to the closest healthy shard of my NGINX service, wherever in the world that may be. If a healthy shard exists in the local cluster, that service’s cluster-local (typically 10.x.y.z) IP address will be returned (by the cluster-local KubeDNS). This is almost exactly equivalent

to non-federated service resolution (almost because KubeDNS actually returns both a CNAME and an A record for local federated services, but applications will be oblivious to this minor technical difference).

But if the service does not exist in the local cluster (or it exists but has no healthy backend pods), the DNS query is automatically expanded to "nginx.mynameSpace.myfederation.svc.us-central1-f.example.com" (i.e. logically "find the external IP of one of the shards closest to my availability zone"). This expansion is performed automatically by KubeDNS, which returns the associated CNAME record. This results in automatic traversal of the hierarchy of DNS records in the above example, and ends up at one of the external IP's of the Federated Service in the local us-central1 region (i.e. 104.197.247.191, 104.197.244.180 or 104.197.245.170).

It is of course possible to explicitly target service shards in availability zones and regions other than the ones local to a Pod by specifying the appropriate DNS names explicitly, and not relying on automatic DNS expansion. For example, "nginx.mynameSpace.myfederation.svc.europe-west1.example.com" will resolve to all of the currently healthy service shards in Europe, even if the Pod issuing the lookup is located in the U.S., and irrespective of whether or not there are healthy shards of the service in the U.S. This is useful for remote monitoring and other similar applications.

From other clients outside your federated clusters

Much of the above discussion applies equally to external clients, except that the automatic DNS expansion described is no longer possible. So external clients need to specify one of the fully qualified DNS names of the Federated Service, be that a zonal, regional or global name. For convenience reasons, it is often a good idea to manually configure additional static CNAME records in your service, for example:

```
eu.nginx.acme.com      CNAME nginx.mynameSpace.myfederation.svc.europe-west1.example.com.  
us.nginx.acme.com      CNAME nginx.mynameSpace.myfederation.svc.us-central1.example.com.  
nginx.acme.com         CNAME nginx.mynameSpace.myfederation.svc.example.com.
```

That way your clients can always use the short form on the left, and always be automatically routed to the closest healthy shard on their home continent. All of the required failover is handled for you automatically by Kubernetes Cluster Federation. Future releases will improve upon this even further.

Handling failures of backend pods and whole clusters

Standard Kubernetes service cluster-IP's already ensure that non-responsive individual Pod endpoints are automatically taken out of service with low latency

(a few seconds). In addition, as alluded above, the Kubernetes Cluster Federation system automatically monitors the health of clusters and the endpoints behind all of the shards of your Federated Service, taking shards in and out of service as required (e.g. when all of the endpoints behind a service, or perhaps the entire cluster or availability zone go down, or conversely recover from an outage). Due to the latency inherent in DNS caching (the cache timeout, or TTL for Federated Service DNS records is configured to 3 minutes, by default, but can be adjusted), it may take up to that long for all clients to completely fail over to an alternative cluster in the case of catastrophic failure. However, given the number of discrete IP addresses which can be returned for each regional service endpoint (see e.g. us-central1 above, which has three alternatives) many clients will fail over automatically to one of the alternative IP's in less time than that given appropriate configuration.

Troubleshooting

I cannot connect to my cluster federation API

Check that your

1. Client (typically kubectl) is correctly configured (including API endpoints and login credentials).
2. Cluster Federation API server is running and network-reachable.

See the federation admin guide to learn how to bring up a cluster federation correctly (or have your cluster administrator do this for you), and how to correctly configure your client.

I can create a federated service successfully against the cluster federation API, but no matching services are created in my underlying clusters

Check that:

1. Your clusters are correctly registered in the Cluster Federation API (`kubectl describe clusters`).
2. Your clusters are all ‘Active’. This means that the cluster Federation system was able to connect and authenticate against the clusters’ endpoints. If not, consult the logs of the federation-controller-manager pod to ascertain what the failure might be.
`kubectl --namespace=federation logs $(kubectl get pods --namespace=federation -l module=federation-controller-manager -o name)`
3. That the login credentials provided to the Cluster Federation API for the clusters have the correct authorization and quota to create services in the

relevant namespace in the clusters. Again you should see associated error messages providing more detail in the above log file if this is not the case.

4. Whether any other error is preventing the service creation operation from succeeding (look for `service-controller` errors in the output of `kubectl logs federation-controller-manager --namespace federation`).

I can create a federated service successfully, but no matching DNS records are created in my DNS provider.

Check that:

1. Your federation name, DNS provider, DNS domain name are configured correctly. Consult the federation admin guide or tutorial to learn how to configure your Cluster Federation system's DNS provider (or have your cluster administrator do this for you).
2. Confirm that the Cluster Federation's service-controller is successfully connecting to and authenticating against your selected DNS provider (look for `service-controller` errors or successes in the output of `kubectl logs federation-controller-manager --namespace federation`).
3. Confirm that the Cluster Federation's service-controller is successfully creating DNS records in your DNS provider (or outputting errors in its logs explaining in more detail what's failing).

Matching DNS records are created in my DNS provider, but clients are unable to resolve against those names

Check that:

1. The DNS registrar that manages your federation DNS domain has been correctly configured to point to your configured DNS provider's name-servers. See for example Google Domains Documentation and Google Cloud DNS Documentation, or equivalent guidance from your domain registrar and DNS provider.

This troubleshooting guide did not help me solve my problem

1. Please use one of our support channels to seek assistance.

For more information

- Federation proposal details use cases that motivated this work.

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

[Edit This Page](#)

Set up Cluster Federation with Kubefed

Note: **Federation V1**, the current Kubernetes federation API which reuses the Kubernetes API resources ‘as is’, is currently considered alpha for many of its features, and there is no clear path to evolve the API to GA. However, there is a **Federation V2** effort in progress to implement a dedicated federation API apart from the Kubernetes API. The details can be found at [sig-multicloud community page](#).

Kubernetes version 1.5 and above includes a new command line tool called **kubefed** to help you administrate your federated clusters. **kubefed** helps you to deploy a new Kubernetes cluster federation control plane, and to add clusters to or remove clusters from an existing federation control plane.

This guide explains how to administer a Kubernetes Cluster Federation using **kubefed**.

Note: **kubefed** is a beta feature in Kubernetes 1.6.

- Before you begin
- Prerequisites
- Getting **kubefed**
- Choosing a host cluster.
- Deploying a federation control plane
- Adding a cluster to a federation
- Removing a cluster from a federation
- Turning down the federation control plane

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 guide assumes that you have a running Kubernetes cluster. Please see one of the getting started guides for installation instructions for your platform.

Getting `kubefed`

Download the client tarball corresponding to the particular release and extract the binaries in the tarball:

Note that until kubernetes versions 1.8.x the federation project was maintained as part of core kubernetes repo. At some point between kubernetes releases 1.8.0 and 1.9.0, it moved into a separate federation repo and is now maintained there. After this move, the federation release information is available at the release page here.

For k8s versions 1.8.x and earlier:

```
curl -LO https://storage.googleapis.com/kubernetes-release/release/${RELEASE-VERSION}/kubernetes-client-linux-amd64.tar.gz
```

Note that the variable `RELEASE-VERSION` should be either appropriately set to or replaced with the actual version needed.

Copy the extracted binary to one of the directories in your `$PATH` and set the executable permission on the binary.

```
sudo cp kubernetes/client/bin/kubefed /usr/local/bin  
sudo chmod +x /usr/local/bin/kubefed
```

For k8s versions 1.9.x and above:

```
curl -LO https://storage.cloud.google.com/kubernetes-federation-release/release/${RELEASE-VERSION}/federation-client-linux-amd64.tar.gz
```

Note that the variable `RELEASE-VERSION` should be replaced with one of the release versions available at federation release page.

Copy the extracted binary to one of the directories in your `$PATH` and set the executable permission on the binary.

```
sudo cp federation/client/bin/kubefed /usr/local/bin  
sudo chmod +x /usr/local/bin/kubefed
```

Install kubectl

You can install a matching version of kubectl using the instructions on the kubectl install page.

Choosing a host cluster.

You'll need to choose one of your Kubernetes clusters to be the *host cluster*. The host cluster hosts the components that make up your federation control plane. Ensure that you have a `kubeconfig` entry in your local `kubeconfig` that corresponds to the host cluster. You can verify that you have the required `kubeconfig` entry by running:

```
kubectl config get-contexts
```

The output should contain an entry corresponding to your host cluster, similar to the following:

CURRENT	NAME	CLUSTER
*	gke_myproject_asia-east1-b_gce-asia-east1	gke_myproject_asia-east1-b_gce-asia-east1

You'll need to provide the `kubeconfig` context (called name in the entry above) for your host cluster when you deploy your federation control plane.

Deploying a federation control plane

To deploy a federation control plane on your host cluster, run `kubefed init` command. When you use `kubefed init`, you must provide the following:

- Federation name
- `--host-cluster-context`, the `kubeconfig` context for the host cluster
- `--dns-provider`, one of '`google-clouddns`', `aws-route53` or `coredns`
- `--dns-zone-name`, a domain name suffix for your federated services

If your host cluster is running in a non-cloud environment or an environment that doesn't support common cloud primitives such as load balancers, you might need additional flags. Please see the on-premises host clusters section below.

The following example command deploys a federation control plane with the name `fellowship`, a host cluster context `rivendell`, and the domain suffix `example.com`:

```
kubefed init fellowship \
--host-cluster-context=rivendell \
--dns-provider="google-clouddns" \
--dns-zone-name="example.com."
```

The domain suffix specified in `--dns-zone-name` must be an existing domain that you control, and that is programmable by your DNS provider. It must also end with a trailing dot.

Once the federation control plane is initialized, query the namespaces:

```
kubectl get namespace --context=fellowship
```

If you do not see the `default` namespace listed (this is due to a bug). Create it yourself with the following command:

```
kubectl create namespace default --context=fellowship
```

The machines in your host cluster must have the appropriate permissions to program the DNS service that you are using. For example, if your cluster is running on Google Compute Engine, you must enable the Google Cloud DNS API for your project.

The machines in Google Kubernetes Engine clusters are created without the Google Cloud DNS API scope by default. If you want to use a Google Kubernetes Engine cluster as a Federation host, you must create it using the `gcloud` command with the appropriate value in the `--scopes` field. You cannot modify a Google Kubernetes Engine cluster directly to add this scope, but you can create a new node pool for your cluster and delete the old one. *Note that this will cause pods in the cluster to be rescheduled.*

To add the new node pool, run:

```
scopes="$(gcloud container node-pools describe --cluster=gke-cluster default-pool --format=JSON | jq -r '.status.addressConfig.aliases[0].name')\ngcloud container node-pools create new-np \  
    --cluster=gke-cluster \  
    --scopes="$scopes,https://www.googleapis.com/auth/ndev.clouddns.readwrite"
```

To delete the old node pool, run:

```
gcloud container node-pools delete default-pool --cluster gke-cluster
```

`kubefed init` sets up the federation control plane in the host cluster and also adds an entry for the federation API server in your local `kubeconfig`. Note that in the beta release in Kubernetes 1.6, `kubefed init` does not automatically set the current context to the newly deployed federation. You can set the current context manually by running:

```
kubectl config use-context fellowship
```

where `fellowship` is the name of your federation.

Basic and token authentication support

`kubefed init` by default only generates TLS certificates and keys to authenticate with the federation API server and writes them to your lo-

cal kubeconfig file. If you wish to enable basic authentication or token authentication for debugging purposes, you can enable them by passing the `--apiserver-enable-basic-auth` flag or the `--apiserver-enable-token-auth` flag.

```
kubefed init fellowship \
--host-cluster-context=rivendell \
--dns-provider="google-clouddns" \
--dns-zone-name="example.com." \
--apiserver-enable-basic-auth=true \
--apiserver-enable-token-auth=true
```

Passing command line arguments to federation components

`kubefed init` bootstraps a federation control plane with default arguments to federation API server and federation controller manager. Some of these arguments are derived from `kubefed init`'s flags. However, you can override these command line arguments by passing them via the appropriate override flags.

You can override the federation API server arguments by passing them to `--apiserver-arg-overrides` and override the federation controller manager arguments by passing them to `--controllermanager-arg-overrides`.

```
kubefed init fellowship \
--host-cluster-context=rivendell \
--dns-provider="google-clouddns" \
--dns-zone-name="example.com." \
--apiserver-arg-overrides="--anonymous-auth=false,--v=4" \
--controllermanager-arg-overrides="--controllers=services=false"
```

Configuring a DNS provider

The Federated service controller programs a DNS provider to expose federated services via DNS names. Certain cloud providers automatically provide the configuration required to program the DNS provider if the host cluster's cloud provider is same as the DNS provider. In all other cases, you have to provide the DNS provider configuration to your federation controller manager which will in-turn be passed to the federated service controller. You can provide this configuration to federation controller manager by storing it in a file and passing the file's local filesystem path to `kubefed init`'s `--dns-provider-config` flag. For example, save the config below in `$HOME/coredns-provider.conf`.

```
[Global]
etcd-endpoints = http://etcd-cluster.ns:2379
zones = example.com.
```

And then pass this file to `kubefed init`:

```
kubefed init fellowship \
--host-cluster-context=rivendell \
--dns-provider="coredns" \
--dns-zone-name="example.com." \
--dns-provider-config="$HOME/coredns-provider.conf"
```

On-premises host clusters

API server service type

`kubefed init` exposes the federation API server as a Kubernetes service on the host cluster. By default, this service is exposed as a load balanced service. Most on-premises and bare-metal environments, and some cloud environments lack support for load balanced services. `kubefed init` allows exposing the federation API server as a `NodePort` service on such environments. This can be accomplished by passing the `--api-server-service-type=NodePort` flag. You can also specify the preferred address to advertise the federation API server by passing the `--api-server-advertise-address=<IP-address>` flag. Otherwise, one of the host cluster's node address is chosen as the default.

```
kubefed init fellowship \
--host-cluster-context=rivendell \
--dns-provider="google-clouddns" \
--dns-zone-name="example.com." \
--api-server-service-type="NodePort" \
--api-server-advertise-address="10.0.10.20"
```

Provisioning storage for etcd

Federation control plane stores its state in `etcd`. `etcd` data must be stored in a persistent storage volume to ensure correct operation across federation control plane restarts. On host clusters that support dynamic provisioning of storage volumes, `kubefed init` dynamically provisions a `PersistentVolume` and binds it to a `PersistentVolumeClaim` to store `etcd` data. If your host cluster doesn't support dynamic provisioning, you can also statically provision a `PersistentVolume`. `kubefed init` creates a `PersistentVolumeClaim` that has the following configuration:

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  annotations:
    volume.alpha.kubernetes.io/storage-class: "yes"
  labels:
    app: federated-cluster
```

```

name: fellowship-federation-apiserver-etcd-claim
namespace: federation-system
spec:
  accessModes:
  - ReadWriteOnce
  resources:
    requests:
      storage: 10Gi

```

To statically provision a `PersistentVolume`, you must ensure that the `PersistentVolume` that you create has the matching storage class, access mode and at least as much capacity as the requested `PersistentVolumeClaim`.

Alternatively, you can disable persistent storage completely by passing `--etcd-persistent-storage=false` to `kubefed init`. However, we do not recommend this because your federation control plane cannot survive restarts in this mode.

```

kubefed init fellowship \
  --host-cluster-context=rivendell \
  --dns-provider="google-clouddns" \
  --dns-zone-name="example.com." \
  --etcd-persistent-storage=false

```

`kubefed init` still doesn't support attaching an existing `PersistentVolumeClaim` to the federation control plane that it bootstraps. We are planning to support this in a future version of `kubefed`.

CoreDNS support

Federated services now support CoreDNS as one of the DNS providers. If you are running your clusters and federation in an environment that does not have access to cloud-based DNS providers, then you can run your own CoreDNS instance and publish the federated service DNS names to that server.

You can configure your federation to use CoreDNS, by passing appropriate values to `kubefed init`'s `--dns-provider` and `--dns-provider-config` flags.

```

kubefed init fellowship \
  --host-cluster-context=rivendell \
  --dns-provider="coredns" \
  --dns-zone-name="example.com." \
  --dns-provider-config="$HOME/coredns-provider.conf"

```

For more information see Setting up CoreDNS as DNS provider for Cluster Federation.

AWS Route53 support

It is possible to utilize AWS Route53 as a cloud DNS provider when the federation controller-manager is run on-premise. The controller-manager Deployment must be configured with AWS credentials since it cannot implicitly gather them from a VM running on AWS.

Currently, `kubefed init` does not read AWS Route53 credentials from the `--dns-provider-config` flag, so a patch must be applied.

Specify AWS Route53 as your DNS provider when initializing your on-premise federation controller-manager by passing the flag `--dns-provider="aws-route53"` to `kubefed init`.

Create a patch file with your AWS credentials:

```
spec:  
  template:  
    spec:  
      containers:  
        - name: controller-manager  
          env:  
            - name: AWS_ACCESS_KEY_ID  
              value: "ABCDEFG1234567890"  
            - name: AWS_SECRET_ACCESS_KEY  
              value: "ABCDEFGHIJKLMNPQRSTUVWXYZ1234567890"
```

Patch the Deployment:

```
kubectl -n federation-system patch deployment controller-manager --patch "$(cat <patch-file>)"
```

Where `<patch-file-name>` is the name of the file you created above.

Adding a cluster to a federation

After you've deployed a federation control plane, you'll need to make that control plane aware of the clusters it should manage.

To join clusters into the federation:

1. Change the context:

```
kubectl config use-context fellowship
```

2. If you are using a managed cluster service, allow the service to access the cluster. To do this, create a `clusterrolebinding` for the account associated with your cluster service:

```
kubectl create clusterrolebinding <your_user>-cluster-admin-binding --clusterrole=clust...
```

3. Join the cluster to the federation, using `kubefed join`, and make sure you provide the following:

- The name of the cluster that you are joining to the federation

- `--host-cluster-context`, the kubeconfig context for the host cluster

For example, this command adds the cluster `gondor` to the federation running on host cluster `rivendell`:

```
kubefed join gondor --host-cluster-context=rivendell
```

A new context has now been added to your kubeconfig named `fellowship` (after the name of your federation).

Note: The name that you provide to the `join` command is used as the joining cluster's identity in federation. If this name adheres to the rules described in the identifiers doc. If the context corresponding to your joining cluster conforms to these rules then you can use the same name in the `join` command. Otherwise, you will have to choose a different name for your cluster's identity.

Naming rules and customization

The cluster name you supply to `kubefed join` must be a valid RFC 1035 label and are enumerated in the Identifiers doc.

Furthermore, federation control plane requires credentials of the joined clusters to operate on them. These credentials are obtained from the local kubeconfig. `kubefed join` uses the cluster name specified as the argument to look for the cluster's context in the local kubeconfig. If it fails to find a matching context, it exits with an error.

This might cause issues in cases where context names for each cluster in the federation don't follow RFC 1035 label naming rules. In such cases, you can specify a cluster name that conforms to the RFC 1035 label naming rules and specify the cluster context using the `--cluster-context` flag. For example, if context of the cluster you are joining is `gondor_needs-no_king`, then you can join the cluster by running:

```
kubefed join gondor --host-cluster-context=rivendell --cluster-context=gondor_needs-no_king
```

Secret name

Cluster credentials required by the federation control plane as described above are stored as a secret in the host cluster. The name of the secret is also derived from the cluster name.

However, the name of a secret object in Kubernetes should conform to the DNS subdomain name specification described in RFC 1123. If this isn't the case, you can pass the secret name to `kubefed join` using the `--secret-name` flag. For example, if the cluster name is `noldor` and the secret name is `11kingdom`, you can join the cluster by running:

```
kubefed join noldor --host-cluster-context=rivendell --secret-name=11kingdom
```

Note: If your cluster name does not conform to the DNS subdomain name specification, all you need to do is supply the secret name via the `--secret-name` flag. `kubefed join` automatically creates the secret for you.

kube-dns configuration

`kube-dns` configuration must be updated in each joining cluster to enable federated service discovery. If the joining Kubernetes cluster is version 1.5 or newer and your `kubefed` is version 1.6 or newer, then this configuration is automatically managed for you when the clusters are joined or unjoined using `kubefed join` or `unjoin` commands.

In all other cases, you must update `kube-dns` configuration manually as described in the Updating KubeDNS section of the admin guide.

Removing a cluster from a federation

To remove a cluster from a federation, run the `kubefed unjoin` command with the cluster name and the federation's `--host-cluster-context`:

```
kubefed unjoin gondor --host-cluster-context=rivendell
```

Turning down the federation control plane

Proper cleanup of federation control plane is not fully implemented in this beta release of `kubefed`. However, for the time being, deleting the federation system namespace should remove all the resources except the persistent storage volume dynamically provisioned for the federation control plane's etcd. You can delete the federation namespace by running the following command:

```
kubectl delete ns federation-system --context=rivendell
```

Note that `rivendell` is the host cluster name, replace that with the appropriate name in your configuration.

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

[Edit This Page](#)

Set up CoreDNS as DNS provider for Cluster Federation

Note: Federation V1, the current Kubernetes federation API which reuses the Kubernetes API resources ‘as is’, is currently considered alpha for many of its features, and there is no clear path to evolve the API to GA. However, there is a **Federation V2** effort in progress to implement a dedicated federation API apart from the Kubernetes API. The details can be found at sig-multicloud community page.

This page shows how to configure and deploy CoreDNS to be used as the DNS provider for Cluster Federation.

- Objectives
- Before you begin
- Deploying CoreDNS and etcd charts
- Deploying Federation with CoreDNS as DNS provider
- Setup CoreDNS server in nameserver resolv.conf chain

Objectives

- Configure and deploy CoreDNS server
- Bring up federation with CoreDNS as dns provider
- Setup CoreDNS server in nameserver lookup chain

Before you begin

- You need to have a running Kubernetes cluster (which is referenced as host cluster). Please see one of the getting started guides for installation instructions for your platform.
- Support for `LoadBalancer` services in member clusters of federation is mandatory to enable CoreDNS for service discovery across federated clusters.

Deploying CoreDNS and etcd charts

CoreDNS can be deployed in various configurations. Explained below is a reference and can be tweaked to suit the needs of the platform and the cluster federation.

To deploy CoreDNS, we shall make use of helm charts. CoreDNS will be deployed with etcd as the backend and should be pre-installed. etcd can also be deployed using helm charts. Shown below are the instructions to deploy etcd.

```
helm install --namespace my-namespace --name etcd-operator stable/etcd-operator
helm upgrade --namespace my-namespace --set cluster.enabled=true etcd-operator stable/etcd-o
```

Note: etcd default deployment configurations can be overridden, suiting the host cluster.

After deployment succeeds, etcd can be accessed with the http://etcd-cluster.my-namespace:2379 endpoint within the host cluster.

The CoreDNS default configuration should be customized to suit the federation. Shown below is the Values.yaml, which overrides the default configuration parameters on the CoreDNS chart.

```
Values.yaml docs/tasks/federation
-----
isClusterService: false
serviceType: "LoadBalancer"
plugins:
  kubernetes:
    enabled: false
  etcd:
    enabled: true
  zones:
    - "example.com."
  endpoint: "http://etcd-cluster.my-namespace:2379"
```

The above configuration file needs some explanation:

- `isClusterService` specifies whether CoreDNS should be deployed as a cluster-service, which is the default. You need to set it to false, so that CoreDNS is deployed as a Kubernetes application service.
- `serviceType` specifies the type of Kubernetes service to be created for CoreDNS. You need to choose either “LoadBalancer” or “NodePort” to make the CoreDNS service accessible outside the Kubernetes cluster.
- Disable `plugins.kubernetes`, which is enabled by default by setting `plugins.kubernetes.enabled` to false.
- Enable `plugins.etcd` by setting `plugins.etcd.enabled` to true.
- Configure the DNS zone (federation domain) for which CoreDNS is authoritative by setting `plugins.etcd.zones` as shown above.
- Configure the etcd endpoint which was deployed earlier by setting `plugins.etcd.endpoint`

Now deploy CoreDNS by running

```
helm install --namespace my-namespace --name coredns -f Values.yaml stable/coredns
```

Verify that both etcd and CoreDNS pods are running as expected.

Deploying Federation with CoreDNS as DNS provider

The Federation control plane can be deployed using `kubefed init`. CoreDNS can be chosen as the DNS provider by specifying two additional parameters.

```
--dns-provider=coredns  
--dns-provider-config=coredns-provider.conf
```

`coredns-provider.conf` has below format:

```
[Global]  
etcd-endpoints = http://etcd-cluster.my-namespace:2379  
zones = example.com.  
coredns-endpoints = <coredns-server-ip>:<port>
```

- `etcd-endpoints` is the endpoint to access etcd.
- `zones` is the federation domain for which CoreDNS is authoritative and is same as `-dns-zone-name` flag of `kubefed init`.
- `coredns-endpoints` is the endpoint to access CoreDNS server. This is an optional parameter introduced from v1.7 onwards.

Note: *plugins.etcd.zones* in CoreDNS configuration and *-dns-zone-name* flag to `kubefed init` should match.

Setup CoreDNS server in nameserver resolv.conf chain

Note: The following section applies only to versions prior to v1.7 and will be automatically taken care of if the `coredns-endpoints` parameter is configured in `coredns-provider.conf` as described in section above.

Once the federation control plane is deployed and federated clusters are joined to the federation, you need to add the CoreDNS server to the pod's nameserver resolv.conf chain in all the federated clusters as this self hosted CoreDNS server is not discoverable publicly. This can be achieved by adding the below line to `dnsmasq` container's arg in `kube-dns` deployment.

```
--server=/example.com./<CoreDNS endpoint>
```

Replace `example.com` above with federation domain.

Now the federated cluster is ready for cross-cluster service discovery!

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

[Edit This Page](#)

Set up placement policies in Federation

Note: Federation V1, the current Kubernetes federation API which reuses the Kubernetes API resources ‘as is’, is currently considered alpha for many of its features, and there is no clear path to evolve the API to GA. However, there is a **Federation V2** effort in progress to implement a dedicated federation API apart from the Kubernetes API. The details can be found at sig-multicluster community page.

This page shows how to enforce policy-based placement decisions over Federated resources using an external policy engine.

- Before you begin
- Deploying Federation and configuring an external policy engine
- Deploying an external policy engine
- Configuring placement policies via ConfigMaps
- Testing placement policies

Before you begin

You need to have a running Kubernetes cluster (which is referenced as host cluster). Please see one of the getting started guides for installation instructions for your platform.

Deploying Federation and configuring an external policy engine

The Federation control plane can be deployed using `kubefed init`.

After deploying the Federation control plane, you must configure an Admission Controller in the Federation API server that enforces placement decisions received from the external policy engine.

```
kubectl create -f scheduling-policy-admission.yaml
```

Shown below is an example ConfigMap for the Admission Controller:

```
scheduling-policy-admission.yaml docs/tasks/federation
apiVersion: v1
kind: ConfigMap
metadata:
  name: admission
  namespace: federation-system
data:
  config.yml: |
    apiVersion: apiserver.k8s.io/v1alpha1
    kind: AdmissionConfiguration
    plugins:
      - name: SchedulingPolicy
        path: /etc/kubernetes/admission/scheduling-policy-config.yml
  scheduling-policy-config.yml: |
    kubeconfig: /etc/kubernetes/admission/opa-kubeconfig
  opa-kubeconfig: |
    clusters:
      - name: opa-api
        cluster:
          server: http://opa.federation-system.svc.cluster.local:8181/v0/data/kubernetes/p...
    users:
      - name: scheduling-policy
        user:
          token: deadbeefsecret
    contexts:
      - name: default
        context:
          cluster: opa-api
          user: scheduling-policy
  current-context: default
```

The ConfigMap contains three files:

- `config.yml` specifies the location of the `SchedulingPolicy` Admission Controller config file.
- `scheduling-policy-config.yml` specifies the location of the kubeconfig file required to contact the external policy engine. This file can also include a `retryBackoff` value that controls the initial retry backoff delay in milliseconds.
- `opa-kubeconfig` is a standard kubeconfig containing the URL and credentials needed to contact the external policy engine.

Edit the Federation API server deployment to enable the `SchedulingPolicy`

Admission Controller.

```
kubectl -n federation-system edit deployment federation-apiserver
```

Update the Federation API server command line arguments to enable the Admission Controller and mount the ConfigMap into the container. If there's an existing `--enable-admission-plugins` flag, append `,SchedulingPolicy` instead of adding another line.

```
--enable-admission-plugins=SchedulingPolicy
--admission-control-config-file=/etc/kubernetes/admission/config.yml
```

Add the following volume to the Federation API server pod:

```
- name: admission-config
  configMap:
    name: admission
```

Add the following volume mount the Federation API server `apiserver` container:

```
volumeMounts:
- name: admission-config
  mountPath: /etc/kubernetes/admission
```

Deploying an external policy engine

The Open Policy Agent (OPA) is an open source, general-purpose policy engine that you can use to enforce policy-based placement decisions in the Federation control plane.

Create a Service in the host cluster to contact the external policy engine:

```
kubectl create -f policy-engine-service.yaml
```

Shown below is an example Service for OPA.

```
policy-engine-service.yaml docs/tasks/federation
```

```
kind: Service
apiVersion: v1
metadata:
  name: opa
  namespace: federation-system
spec:
  selector:
    app: opa
  ports:
  - name: http
    protocol: TCP
    port: 8181
    targetPort: 8181
```

Create a Deployment in the host cluster with the Federation control plane:

```
kubectl create -f policy-engine-deployment.yaml
```

Shown below is an example Deployment for OPA.

```
policy-engine-deployment.yaml docs/tasks/federation
apiVersion: apps/v1
kind: Deployment
metadata:
  labels:
    app: opa
  name: opa
  namespace: federation-system
spec:
  replicas: 1
  template:
    metadata:
      labels:
        app: opa
      name: opa
    spec:
      containers:
        - name: opa
          image: openpolicyagent/opa:0.4.10
          args:
            - "run"
            - "--server"
        - name: kube-mgmt
          image: openpolicyagent/kube-mgmt:0.2
          args:
            - "-kubeconfig=/srv/kubernetes/kubeconfig"
            - "-cluster=federation/v1beta1/clusters"
          volumeMounts:
            - name: federation-kubeconfig
              mountPath: /srv/kubernetes
              readOnly: true
      volumes:
        - name: federation-kubeconfig
          secret:
            secretName: federation-controller-manager-kubeconfig
```

Configuring placement policies via ConfigMaps

The external policy engine will discover placement policies created in the `kube-federation-scheduling-policy` namespace in the Federation API server.

Create the namespace if it does not already exist:

```
kubectl --context=federation create namespace kube-federation-scheduling-policy
```

Configure a sample policy to test the external policy engine:

```
policy.rego docs/tasks/federation

# OPA supports a high-level declarative language named Rego for authoring and
# enforcing policies. For more information on Rego, visit
# http://openpolicyagent.org.

# Rego policies are namespaced by the "package" directive.
package kubernetes.placement

# Imports provide aliases for data inside the policy engine. In this case, the
# policy simply refers to "clusters" below.
import data.kubernetes.clusters

# The "annotations" rule generates a JSON object containing the key
# "federation.kubernetes.io/replica-set-preferences" mapped to <preferences>.
# The preferences values is generated dynamically by OPA when it evaluates the
# rule.
#
# The SchedulingPolicy Admission Controller running inside the Federation API
# server will merge these annotations into incoming Federated resources. By
# setting replica-set-preferences, we can control the placement of Federated
# ReplicaSets.
#
# Rules are defined to generate JSON values (booleans, strings, objects, etc.)
# When OPA evaluates a rule, it generates a value IF all of the expressions in
# the body evaluate successfully. All rules can be understood intuitively as
# <head> if <body> where <body> is true if <expr-1> AND <expr-2> AND ...
# <expr-N> is true (for some set of data.)
annotations["federation.kubernetes.io/replica-set-preferences"] = preferences {
    input.kind = "ReplicaSet"
    value = {"clusters": cluster_map, "rebalance": true}
    json.marshal(value, preferences)
}

# This "annotations" rule generates a value for the "federation.alpha.kubernetes.io/cluster"
# annotation.
#
# In English, the policy asserts that resources in the "production" namespace
# that are not annotated with "criticality=low" MUST be placed on clusters
# labelled with "on-premises=true".
annotations["federation.alpha.kubernetes.io/cluster-selector"] = selector {
    input.metadata.namespace = "production"
    not input.metadata.annotations.criticality = "low"
    json.marshal([
        "operator": "=",
        "key": "on-premises",
        "values": "[true]",           709
    ], selector)
}

# Generates a set of cluster names that satisfy the incoming Federated
# ReplicaSet's requirements. In this case, just PCI compliance.
replica_set_clusters[cluster_name] {
    clusters[cluster_name]
    not insufficient_pci[cluster_name]
```

Shown below is the command to create the sample policy:

```
kubectl --context=federation -n kube-federation-scheduling-policy create configmap scheduling-policy --from-file=scheduling-policy.yaml
```

This sample policy illustrates a few key ideas:

- Placement policies can refer to any field in Federated resources.
- Placement policies can leverage external context (for example, Cluster metadata) to make decisions.
- Administrative policy can be managed centrally.
- Policies can define simple interfaces (such as the `requires-pci` annotation) to avoid duplicating logic in manifests.

Testing placement policies

Annotate one of the clusters to indicate that it is PCI certified.

```
kubectl --context=federation annotate clusters cluster-name-1 pci-certified=true
```

Deploy a Federated ReplicaSet to test the placement policy.

```
replicaset-example-policy.yaml docs/tasks/federation
apiVersion: apps/v1
kind: ReplicaSet
metadata:
  labels:
    app: nginx-pci
  name: nginx-pci
  annotations:
    requires-pci: "true"
spec:
  replicas: 3
  selector:
    matchLabels:
      app: nginx-pci
  template:
    metadata:
      labels:
        app: nginx-pci
    spec:
      containers:
        - image: nginx
          name: nginx-pci
```

Shown below is the command to deploy a ReplicaSet that *does* match the policy.

```
kubectl --context=federation create -f replicaset-example-policy.yaml
```

Inspect the ReplicaSet to confirm the appropriate annotations have been applied:

```
kubectl --context=federation get rs nginx-pci -o jsonpath='{.metadata.annotations}'
```

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

[Edit This Page](#)

Performing 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 that DaemonSet revisions only roll forward. That is to say, after a rollback is complete, 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.

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

[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
- Caveat: Updating DaemonSet created from Kubernetes version 1.5 or before
- 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`: This is the default update strategy for backward-compatibility. 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: 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.

Caveat: Updating DaemonSet created from Kubernetes version 1.5 or before

If you try a rolling update on a DaemonSet that was created from Kubernetes version 1.5 or before, a rollout will be triggered when you *first* change the DaemonSet update strategy to `RollingUpdate`, no matter if DaemonSet template is modified or not. If the DaemonSet template is not changed, all existing DaemonSet pods will be restarted (deleted and created).

Therefore, make sure you want to trigger a rollout before you first switch the strategy to `RollingUpdate`.

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 create -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 that this will cause service disruption if the deleted pods are not controlled by any controllers, or if the pods aren't replicated. This doesn't 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

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

[Edit This Page](#)

Performing 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 that DaemonSet revisions only roll forward. That is to say, after a rollback is complete, 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.

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

[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 ServicesA software offering maintained by a third-party provider. from Service BrokersAn 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-incubator/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	VERSION	DESCRIPTION
svc-cat/catalog	0.0.1	service-catalog API server and controller-manag...

Enable RBAC

Your Kubernetes cluster must have RBAC enabled, which requires your Tiller Pod(s) to have `cluster-admin` access.

If you are using Minikube, run the `minikube start` command with the following flag:

```
minikube start --extra-config=apiserver.Authorization.Mode=RBAC
```

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 -O
```

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 install svc-cat/catalog \
--name catalog --namespace catalog
```

What's next

- View sample service brokers.
- Explore the `kubernetes-incubator/service-catalog` project.

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

[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 ServicesA software offering maintained by a third-party provider. from Service BrokersAn

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-incubator/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	VERSION	DESCRIPTION
svc-cat/catalog	0.0.1	service-catalog API server and controller-manag...

Enable RBAC

Your Kubernetes cluster must have RBAC enabled, which requires your Tiller Pod(s) to have `cluster-admin` access.

If you are using Minikube, run the `minikube start` command with the following flag:

```
minikube start --extra-config=apiserver.Authorization.Mode=RBAC
```

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 -O
```

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 install svc-cat/catalog \
--name catalog --namespace catalog
```

What's next

- View sample service brokers.
- Explore the kubernetes-incubator/service-catalog project.

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

[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 ServicesA software offering maintained by a third-party provider. from Service BrokersAn 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 the Service Catalog Installer tool to easily install or uninstall Service Catalog on your Kubernetes cluster. This CLI tool is installed as `sc` in your local environment.

- 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=kubelet
```

Install sc in your local environment

Install the `sc` CLI tool using the `go get` command:

```
go get github.com/GoogleCloudPlatform/k8s-service-catalog/installer/cmd/sc
```

After running the above command, `sc` should 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.

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

[Edit This Page](#)

Federated Cluster

Note: Federation V1, the current Kubernetes federation API which reuses the Kubernetes API resources ‘as is’, is currently considered alpha for many of its features, and there is no clear path to evolve the API to GA. However, there is a **Federation V2** effort in progress to implement a dedicated federation API apart from the Kubernetes API. The details can be found at sig-multicloud community page.

This guide explains how to use Clusters API resource in a Federation control plane.

Different than other Kubernetes resources, such as Deployments, Services and ConfigMaps, clusters only exist in the federation context, i.e. those requests must be submitted to the federation api-server.

- Before you begin
- Listing Clusters
- Creating a Federated Cluster
- Deleting a Federated Cluster
- Labeling Clusters
- ClusterSelector Annotation
- Clusters API reference

Before you begin

- This guide assumes that you have a running Kubernetes Cluster Federation installation. If not, then head over to the federation admin guide to learn how to bring up a cluster federation (or have your cluster administrator do this for you). Other tutorials, such as Kelsey Hightower's Federated Kubernetes Tutorial, might also help you create a Federated Kubernetes cluster.
- You should also have a basic working knowledge of Kubernetes in general.

Listing Clusters

To list the clusters available in your federation, you can use kubectl by running:

```
kubectl --context=federation get clusters
```

The `--context=federation` flag tells kubectl to submit the request to the Federation apiserver instead of sending it to a Kubernetes cluster. If you submit it to a k8s cluster, you will receive an error saying

```
the server doesn't have a resource type "clusters"
```

If you passed the correct Federation context but received a message error saying

```
No resources found.
```

it means that you haven't added any cluster to the Federation yet.

Creating a Federated Cluster

Creating a `cluster` resource in federation means joining it to the federation. To do so, you can use `kubefed join`. Basically, you need to give the new cluster a

name and say what is the name of the context that corresponds to a cluster that hosts the federation. The following example command adds the cluster `gondor` to the federation running on host cluster `rivendell`:

```
kubefed join gondor --host-cluster-context=rivendell
```

You can find more details on how to do that in the respective section in the kubefed guide.

Deleting a Federated Cluster

Converse to creating a cluster, deleting a cluster means unjoining this cluster from the federation. This can be done with `kubefed unjoin` command. To remove the `gondor` cluster, just do:

```
kubefed unjoin gondor --host-cluster-context=rivendell
```

You can find more details on unjoin in the kubefed guide.

Labeling Clusters

You can label clusters the same way as any other Kubernetes object, which can help with grouping clusters and can also be leveraged by the `ClusterSelector`.

```
kubectl --context=rivendell label cluster gondor key1=value1 key2=value2
```

ClusterSelector Annotation

Starting in Kubernetes 1.7, there is alpha support for directing objects across the federated clusters with the annotation `federation.alpha.kubernetes.io/cluster-selector`. The `ClusterSelector` is conceptually similar to `nodeSelector`, but instead of selecting against labels on nodes, it selects against labels on federated clusters.

The annotation value must be JSON formatted and must be parsable into the `ClusterSelector` API type. For example: `[{"key": "load", "operator": "Lt", "values": ["10"]}]`. Content that doesn't parse correctly will throw an error and prevent distribution of the object to any federated clusters. Objects of type ConfigMap, Secret, Daemonset, Service and Ingress are included in the alpha implementation.

Here is an example `ClusterSelector` annotation, which will only select clusters WITH the label `pci=true` and WITHOUT the label `environment=test`:

```
metadata:  
  annotations:  
    federation.alpha.kubernetes.io/cluster-selector: '[{"key": "pci", "operator": "In", "values": ["true"]}, {"key": "environment", "operator": "NotIn", "values":
```

```
["test"]}]'
```

The *key* is matched against label names on the federated clusters.

The *values* are matched against the label values on the federated clusters.

The possible *operators* are: `In`, `NotIn`, `Exists`, `DoesNotExist`, `Gt`, `Lt`.

The *values* field is expected to be empty when `Exists` or `DoesNotExist` is specified and may include more than one string when `In` or `NotIn` are used.

Currently, only integers are supported with `Gt` or `Lt`.

Clusters API reference

The full clusters API reference is currently in `federation/v1beta1` and more details can be found in the Federation API reference page.

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

[Edit This Page](#)

Federated Cluster

Note: `Federation V1`, the current Kubernetes federation API which reuses the Kubernetes API resources ‘as is’, is currently considered alpha for many of its features, and there is no clear path to evolve the API to GA. However, there is a `Federation V2` effort in progress to implement a dedicated federation API apart from the Kubernetes API. The details can be found at sig-multicluster community page.

This guide explains how to use Clusters API resource in a Federation control plane.

Different than other Kubernetes resources, such as Deployments, Services and ConfigMaps, clusters only exist in the federation context, i.e. those requests must be submitted to the federation api-server.

- Before you begin
- Listing Clusters
- Creating a Federated Cluster
- Deleting a Federated Cluster
- Labeling Clusters
- ClusterSelector Annotation
- Clusters API reference

Before you begin

- This guide assumes that you have a running Kubernetes Cluster Federation installation. If not, then head over to the federation admin guide to learn how to bring up a cluster federation (or have your cluster administrator do this for you). Other tutorials, such as Kelsey Hightower’s Federated Kubernetes Tutorial, might also help you create a Federated Kubernetes cluster.
- You should also have a basic working knowledge of Kubernetes in general.

Listing Clusters

To list the clusters available in your federation, you can use `kubectl` by running:

```
kubectl --context=federation get clusters
```

The `--context=federation` flag tells `kubectl` to submit the request to the Federation apiserver instead of sending it to a Kubernetes cluster. If you submit it to a k8s cluster, you will receive an error saying

```
the server doesn't have a resource type "clusters"
```

If you passed the correct Federation context but received a message error saying `No resources found.`

it means that you haven’t added any cluster to the Federation yet.

Creating a Federated Cluster

Creating a `cluster` resource in federation means joining it to the federation. To do so, you can use `kubefed join`. Basically, you need to give the new cluster a name and say what is the name of the context that corresponds to a cluster that hosts the federation. The following example command adds the cluster `gondor` to the federation running on host cluster `rivendell`:

```
kubefed join gondor --host-cluster-context=rivendell
```

You can find more details on how to do that in the respective section in the `kubefed` guide.

Deleting a Federated Cluster

Converse to creating a cluster, deleting a cluster means unjoining this cluster from the federation. This can be done with `kubefed unjoin` command. To remove the `gondor` cluster, just do:

```
kubefed unjoin gondor --host-cluster-context=rivendell
```

You can find more details on unjoin in the kubefed guide.

Labeling Clusters

You can label clusters the same way as any other Kubernetes object, which can help with grouping clusters and can also be leveraged by the ClusterSelector.

```
kubectl --context=rivendell label cluster gondor key1=value1 key2=value2
```

ClusterSelector Annotation

Starting in Kubernetes 1.7, there is alpha support for directing objects across the federated clusters with the annotation `federation.alpha.kubernetes.io/cluster-selector`. The `ClusterSelector` is conceptually similar to `nodeSelector`, but instead of selecting against labels on nodes, it selects against labels on federated clusters.

The annotation value must be JSON formatted and must be parsable into the ClusterSelector API type. For example: `[{"key": "load", "operator": "Lt", "values": ["10"]}]`. Content that doesn't parse correctly will throw an error and prevent distribution of the object to any federated clusters. Objects of type ConfigMap, Secret, Daemonset, Service and Ingress are included in the alpha implementation.

Here is an example ClusterSelector annotation, which will only select clusters WITH the label `pci=true` and WITHOUT the label `environment=test`:

```
metadata:  
  annotations:  
    federation.alpha.kubernetes.io/cluster-selector: '[{"key": "pci", "operator":  
      "In", "values": ["true"]}, {"key": "environment", "operator": "NotIn", "values":  
      ["test"]}]'
```

The `key` is matched against label names on the federated clusters.

The `values` are matched against the label values on the federated clusters.

The possible `operators` are: `In`, `NotIn`, `Exists`, `DoesNotExist`, `Gt`, `Lt`.

The `values` field is expected to be empty when `Exists` or `DoesNotExist` is specified and may include more than one string when `In` or `NotIn` are used.

Currently, only integers are supported with `Gt` or `Lt`.

Clusters API reference

The full clusters API reference is currently in `federation/v1beta1` and more details can be found in the Federation API reference page.

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

[Edit This Page](#)

Federated ConfigMap

Note: **Federation V1**, the current Kubernetes federation API which reuses the Kubernetes API resources ‘as is’, is currently considered alpha for many of its features, and there is no clear path to evolve the API to GA. However, there is a **Federation V2** effort in progress to implement a dedicated federation API apart from the Kubernetes API. The details can be found at [sig-multicluster community page](#).

This guide explains how to use ConfigMaps in a Federation control plane.

Federated ConfigMaps are very similar to the traditional Kubernetes ConfigMaps and provide the same functionality. Creating them in the federation control plane ensures that they are synchronized across all the clusters in federation.

- Before you begin
- Creating a Federated ConfigMap
- Updating a Federated ConfigMap
- Deleting a Federated ConfigMap

Before you begin

- This guide assumes that you have a running Kubernetes Cluster Federation installation. If not, then head over to the federation admin guide to learn how to bring up a cluster federation (or have your cluster administrator do this for you). Other tutorials, such as Kelsey Hightower’s Federated Kubernetes Tutorial, might also help you create a Federated Kubernetes cluster.
- You should also have a basic working knowledge of Kubernetes in general and ConfigMaps in particular.

Creating a Federated ConfigMap

The API for Federated ConfigMap is 100% compatible with the API for traditional Kubernetes ConfigMap. You can create a ConfigMap by sending a request to the federation apiserver.

You can do that using kubectl by running:

```
kubectl --context=federation-cluster create -f myconfigmap.yaml
```

The `--context=federation-cluster` flag tells kubectl to submit the request to the Federation apiserver instead of sending it to a Kubernetes cluster.

Once a Federated ConfigMap is created, the federation control plane will create a matching ConfigMap in all underlying Kubernetes clusters. You can verify this by checking each of the underlying clusters, for example:

```
kubectl --context=gce-asia-east1a get configmap myconfigmap
```

The above assumes that you have a context named ‘gce-asia-east1a’ configured in your client for your cluster in that zone.

These ConfigMaps in underlying clusters will match the Federated ConfigMap.

Updating a Federated ConfigMap

You can update a Federated ConfigMap as you would update a Kubernetes ConfigMap; however, for a Federated ConfigMap, you must send the request to the federation apiserver instead of sending it to a specific Kubernetes cluster. The federation control plane ensures that whenever the Federated ConfigMap is updated, it updates the corresponding ConfigMaps in all underlying clusters to match it.

Deleting a Federated ConfigMap

You can delete a Federated ConfigMap as you would delete a Kubernetes ConfigMap; however, for a Federated ConfigMap, you must send the request to the federation apiserver instead of sending it to a specific Kubernetes cluster.

For example, you can do that using kubectl by running:

```
kubectl --context=federation-cluster delete configmap
```

Note that at this point, deleting a Federated ConfigMap will not delete the corresponding ConfigMaps from underlying clusters. You must delete the underlying ConfigMaps manually. We intend to fix this in the future.

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

[Edit This Page](#)

Federated DaemonSet

Note: Federation V1, the current Kubernetes federation API which reuses the Kubernetes API resources ‘as is’, is currently considered alpha for many of its features, and there is no clear path to evolve the API to GA. However, there is a **Federation V2** effort in progress to implement a dedicated federation API apart from the Kubernetes API. The details can be found at [sig-multicloud community page](#).

This guide explains how to use DaemonSets in a federation control plane.

DaemonSets in the federation control plane (“Federated Daemonsets” in this guide) are very similar to the traditional Kubernetes DaemonSets and provide the same functionality. Creating them in the federation control plane ensures that they are synchronized across all the clusters in federation.

- Before you begin
- Creating a Federated Daemonset
- Updating a Federated Daemonset
- Deleting a Federated Daemonset

Before you begin

- This guide assumes that you have a running Kubernetes Cluster Federation installation. If not, then head over to the federation admin guide to learn how to bring up a cluster federation (or have your cluster administrator do this for you). Other tutorials, such as Kelsey Hightower’s Federated Kubernetes Tutorial, might also help you create a Federated Kubernetes cluster.
- You are also expected to have a basic working knowledge of Kubernetes in general and DaemonSets in particular.

Creating a Federated Daemonset

The API for Federated Daemonset is 100% compatible with the API for traditional Kubernetes DaemonSet. You can create a DaemonSet by sending a request to the federation apiserver.

You can do that using kubectl by running:

```
kubectl --context=federation-cluster create -f mydaemonset.yaml
```

The `--context=federation-cluster` flag tells kubectl to submit the request to the Federation apiserver instead of sending it to a Kubernetes cluster.

Once a Federated Daemonset is created, the federation control plane will create a matching DaemonSet in all underlying Kubernetes clusters. You can verify this by checking each of the underlying clusters, for example:

```
kubectl --context=gce-asia-east1a get daemonset mydaemonset
```

The above assumes that you have a context named ‘gce-asia-east1a’ configured in your client for your cluster in that zone.

Updating a Federated Daemonset

You can update a Federated Daemonset as you would update a Kubernetes DaemonSet; however, for a Federated Daemonset, you must send the request to the federation apiserver instead of sending it to a specific Kubernetes cluster. The federation control plane ensures that whenever the Federated Daemonset is updated, it updates the corresponding DaemonSets in all underlying clusters to match it.

Deleting a Federated Daemonset

You can delete a Federated Daemonset as you would delete a Kubernetes DaemonSet; however, for a Federated Daemonset, you must send the request to the federation apiserver instead of sending it to a specific Kubernetes cluster.

For example, you can do that using kubectl by running:

```
kubectl --context=federation-cluster delete daemonset mydaemonset
```

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

[Edit This Page](#)

Federated Deployment

Note: Federation V1, the current Kubernetes federation API which reuses the Kubernetes API resources ‘as is’, is currently considered alpha for many of its features, and there is no clear path to evolve the API to GA. However, there is a **Federation V2** effort in progress to implement a dedicated federation API apart from the Kubernetes API. The details can be found at [sig-multicloud community page](#).

This guide explains how to use Deployments in the Federation control plane.

Deployments in the federation control plane (referred to as “Federated Deployments” in this guide) are very similar to the traditional Kubernetes Deployment and provide the same functionality. Creating them in the federation control plane ensures that the desired number of replicas exist across the registered clusters.

FEATURE STATE: Kubernetes 1.5 alpha

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

- The version names contain alpha (e.g. v1alpha1).
- Might be buggy. Enabling the feature may expose bugs. Disabled by default.
- Support for feature may be dropped at any time without notice.
- The API may change in incompatible ways in a later software release without notice.
- Recommended for use only in short-lived testing clusters, due to increased risk of bugs and lack of long-term support.

Some features (such as full rollout compatibility) are still in development.

- Before you begin
- Creating a Federated Deployment
- Updating a Federated Deployment
- Deleting a Federated Deployment

Before you begin

- This guide assumes that you have a running Kubernetes Cluster Federation installation. If not, then head over to the federation admin guide to learn how to bring up a cluster federation (or have your cluster administrator do this for you). Other tutorials, such as Kelsey Hightower’s Federated Kubernetes Tutorial, might also help you create a Federated Kubernetes cluster.
- You should also have a basic working knowledge of Kubernetes in general and Deployments in particular.

Creating a Federated Deployment

The API for Federated Deployment is compatible with the API for traditional Kubernetes Deployment. You can create a Deployment by sending a request to the federation apiserver.

You can do that using kubectl by running:

```
kubectl --context=federation-cluster create -f mydeployment.yaml
```

The ‘`--context=federation-cluster`’ flag tells kubectl to submit the request to the Federation apiserver instead of sending it to a Kubernetes cluster.

Once a Federated Deployment is created, the federation control plane will create a Deployment in all underlying Kubernetes clusters. You can verify this by checking each of the underlying clusters, for example:

```
kubectl --context=gce-asia-east1a get deployment mydep
```

The above assumes that you have a context named ‘gce-asia-east1a’ configured in your client for your cluster in that zone.

These Deployments in underlying clusters will match the federation Deployment *except* in the number of replicas and revision-related annotations. Federation control plane ensures that the sum of replicas in each cluster combined matches the desired number of replicas in the Federated Deployment.

Spreading Replicas in Underlying Clusters

By default, replicas are spread equally in all the underlying clusters. For example: if you have 3 registered clusters and you create a Federated Deployment with `spec.replicas = 9`, then each Deployment in the 3 clusters will have `spec.replicas=3`. To modify the number of replicas in each cluster, you can specify `FederatedReplicaSetPreference` as an annotation with key `federation.kubernetes.io/deployment-preferences` on Federated Deployment.

Updating a Federated Deployment

You can update a Federated Deployment as you would update a Kubernetes Deployment; however, for a Federated Deployment, you must send the request to the federation apiserver instead of sending it to a specific Kubernetes cluster. The federation control plane ensures that whenever the Federated Deployment is updated, it updates the corresponding Deployments in all underlying clusters to match it. So if the rolling update strategy was chosen then the underlying cluster will do the rolling update independently and `maxSurge` and `maxUnavailable` will apply only to individual clusters. This behavior may change in the future.

If your update includes a change in number of replicas, the federation control plane will change the number of replicas in underlying clusters to ensure that their sum remains equal to the number of desired replicas in Federated Deployment.

Deleting a Federated Deployment

You can delete a Federated Deployment as you would delete a Kubernetes Deployment; however, for a Federated Deployment, you must send the request to the federation apiserver instead of sending it to a specific Kubernetes cluster.

For example, you can do that using kubectl by running:

```
kubectl --context=federation-cluster delete deployment mydep
```

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

[Edit This Page](#)

Federated Events

Note: Federation V1, the current Kubernetes federation API which reuses the Kubernetes API resources ‘as is’, is currently considered alpha for many of its features, and there is no clear path to evolve the API to GA. However, there is a **Federation V2** effort in progress to implement a dedicated federation API apart from the Kubernetes API. The details can be found at sig-multicloud community page.

This guide explains how to use events in federation control plane to help in debugging.

- Prerequisites
- View federation events

Prerequisites

This guide assumes that you have a running Kubernetes Cluster Federation installation. If not, then head over to the federation admin guide to learn how to bring up a cluster federation (or have your cluster administrator do this for you). Other tutorials, for example this one by Kelsey Hightower, are also available to help you.

You are also expected to have a basic working knowledge of Kubernetes in general.

View federation events

Events in federation control plane (referred to as “federation events” in this guide) are very similar to the traditional Kubernetes Events providing the same

functionality. Federation Events are stored only in federation control plane and are not passed on to the underlying Kubernetes clusters.

Federation controllers create events as they process API resources to surface to the user, the state that they are in. You can get all events from federation apiserver by running:

```
kubectl --context=federation-cluster get events
```

The standard kubectl get, update, delete commands will all work.

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

[Edit This Page](#)

Federated Horizontal Pod AutoScalers (HPA)

FEATURE STATE: Kubernetes v1.10 alpha

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

- The version names contain alpha (e.g. v1alpha1).
- Might be buggy. Enabling the feature may expose bugs. Disabled by default.
- Support for feature may be dropped at any time without notice.
- The API may change in incompatible ways in a later software release without notice.
- Recommended for use only in short-lived testing clusters, due to increased risk of bugs and lack of long-term support.

Note: Federation V1, the current Kubernetes federation API which reuses the Kubernetes API resources ‘as is’, is currently considered alpha for many of its features, and there is no clear path to evolve the API to GA. However, there is a Federation V2 effort in progress to implement a dedicated federation API apart from the Kubernetes API. The details can be found at [sig-multicloud community page](#).

This guide explains how to use federated horizontal pod autoScalers (HPAs) in the federation control plane.

HPAs in the federation control plane are similar to the traditional Kubernetes HPAs, and provide the same functionality. Creating an HPA targeting a federated object in the federation control plane ensures that the desired number of replicas of the target object are scaled across the registered clusters, instead of a single cluster. Also, the control plane keeps monitoring the status of each individual HPA in the federated clusters and ensures the workload replicas move

where they are needed most by manipulating the min and max limits of the HPA objects in the federated clusters.

- Before you begin
- Creating a federated HPA
- Updating a federated ReplicaSet
- Deleting a federated HPA
- Alternative ways to use federated HPA
- Conclusion

Before you begin

- This guide assumes that you have a running Kubernetes Cluster Federation installation. If not, then head over to the federation admin guide to learn how to bring up a cluster federation (or have your cluster administrator do this for you). Other tutorials, such as Kelsey Hightower's Federated Kubernetes Tutorial, might also help you create a Federated Kubernetes cluster.
- You are also expected to have a basic working knowledge of Kubernetes in general and HPAs in particular.

The federated HPA is an alpha feature. The API is not enabled by default on the federated API server. To use this feature, the user or the admin deploying the federation control plane needs to run the federated API server with option `--runtime-config=api/all=true` to enable all APIs, including alpha APIs. Additionally, the federated HPA only works when used with CPU utilization metrics.

Creating a federated HPA

The API for federated HPAs is 100% compatible with the API for traditional Kubernetes HPA. You can create an HPA by sending a request to the federation API server.

You can do that with kubectl by running:

```
cat <<EOF | kubectl --context=federation-cluster create -f -
apiVersion: autoscaling/v1
kind: HorizontalPodAutoscaler
metadata:
  name: php-apache
  namespace: default
spec:
  scaleTargetRef:
    apiVersion: apps/v1beta1
```

```
kind: Deployment
  name: php-apache
  minReplicas: 1
  maxReplicas: 10
  targetCPUUtilizationPercentage: 50
EOF
```

The `--context=federation-cluster` flag tells `kubectl` to submit the request to the federation API server instead of sending it to a Kubernetes cluster.

Once a federated HPA is created, the federation control plane partitions and creates the HPA in all underlying Kubernetes clusters. As of Kubernetes V1.7, cluster selectors can also be used to restrict any federated object, including the HPAs in a subset of clusters.

You can verify the creation by checking each of the underlying clusters. For example, with a context named `gce-asia-east1a` configured in your client for your cluster in that zone:

```
kubectl --context=gce-asia-east1a get HPA php-apache
```

The HPA in the underlying clusters will match the federation HPA except in the number of min and max replicas. The federation control plane ensures that the sum of max replicas in each cluster matches the specified max replicas on the federated HPA object, and the sum of minimum replicas will be greater than or equal to the minimum specified on the federated HPA object.

Note: A particular cluster cannot have a minimum replica sum of 0.

Spreading HPA min and max replicas in underlying clusters

By default, first max replicas are spread equally in all the underlying clusters, then min replicas are distributed to those clusters that received their maximum value. This means that each cluster will get an HPA if the specified max replicas are greater than the total clusters participating in this federation, and some clusters will be skipped if specified max replicas are less than the total clusters participating in the federation.

For example: if you have 3 registered clusters and you create a federated HPA with `spec.maxReplicas = 9`, and `spec.minReplicas = 2`, then each HPA in the 3 clusters will get `spec.maxReplicas=3` and `spec.minReplicas = 1`.

Currently the default distribution is only available on the federated HPA, but in the future, users preferences could also be specified to control and/or restrict this distribution.

Updating a federated ReplicaSet

You can update a federated HPA as you would update a Kubernetes HPA; however, for a federated HPA, you must send the request to the federation API server instead of sending it to a specific Kubernetes cluster. The Federation control plane ensures that whenever the federated HPA is updated, it updates the corresponding HPA in all underlying clusters to match it.

If your update includes a change in the number of replicas, the federation control plane will change the number of replicas in underlying clusters to ensure that the sum of the max and min replicas remains matched as specified in the previous section.

Deleting a federated HPA

You can delete a federated HPA as you would delete a Kubernetes HPA; however, for a federated HPA, you must send the request to the federation API server instead of sending it to a specific Kubernetes cluster. It should also be noted that for the federated resource to be deleted from all underlying clusters, cascading deletion should be used.

For example, you can do that using `kubectl` by running:

```
kubectl --context=federation-cluster delete HPA php-apache
```

Alternative ways to use federated HPA

To a federation user interacting with federated control plane (or simply federation), the interaction is almost identical to interacting with a normal Kubernetes cluster (but with a limited set of APIs that are federated). As both Deployments and HorizontalPodAutoscalers are now federated, `kubectl` commands like `kubectl run` and `kubectl autoscale` work on federation. Given this fact, the mechanism specified in horizontal pod autoscaler walkthrough will also work when used with federation. Care however will need to be taken that when generating load on a target deployment, it should be done against a specific federated cluster (or multiple clusters) not the federation.

Conclusion

The use of federated HPA is to ensure workload replicas move to the cluster(s) where they are needed most, or in other words where the load is beyond expected threshold. The federated HPA feature achieves this by manipulating the min and max replicas on the HPAs it creates in the federated clusters. It does not directly monitor the target object metrics from the federated clusters.

It actually relies on the in-cluster HPA controllers to monitor the metrics and update relevant fields. The in-cluster HPA controller monitors the target pod metrics and updates the fields like desired replicas (after metrics based calculations) and current replicas (observing the current status of in cluster pods). The federated HPA controller, on the other hand, monitors only the cluster-specific HPA object fields and updates the min replica and max replica fields of those in cluster HPA objects, which have replicas matching thresholds.

For example, if a cluster has both desired replicas and current replicas the same as the max replicas, and averaged current CPU utilization still higher than the target CPU utilization (all of which are fields on local HPA object), then the target app in this cluster needs more replicas, and the scaling is currently restricted by max replicas set on this local HPA object. In such a scenario, the federated HPA controller scans all clusters and tries to find clusters which do not have such a condition (meaning the desired replicas are less than the max, and current averaged CPU utilization is lower than the threshold). If it finds such a cluster, it reduces the max replica on the HPA in this cluster and increases the max replicas on the HPA in the cluster which needed the replicas.

There are many other similar conditions which the federated HPA controller checks and moves the max replicas and min replicas around the local HPAs in federated clusters to eventually ensure that the replicas move (or remain) in the cluster(s) which need them.

For more information, see “federated HPA design proposal”.

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

[Edit This Page](#)

Federated Ingress

Note: **Federation V1**, the current Kubernetes federation API which reuses the Kubernetes API resources ‘as is’, is currently considered alpha for many of its features, and there is no clear path to evolve the API to GA. However, there is a **Federation V2** effort in progress to implement a dedicated federation API apart from the Kubernetes API. The details can be found at [sig-multicloud community page](#).

This page explains how to use Kubernetes Federated Ingress to deploy a common HTTP(S) virtual IP load balancer across a federated service running in multiple Kubernetes clusters. As of v1.4, clusters hosted in Google Cloud (both Google Kubernetes Engine and GCE, or both) are supported. This makes it easy to deploy a service that reliably serves HTTP(S) traffic originating from web clients around the globe on a single, static IP address. Low network latency, high

fault tolerance and easy administration are ensured through intelligent request routing and automatic replica relocation (using Federated ReplicaSets). Clients are automatically routed, via the shortest network path, to the cluster closest to them with available capacity (despite the fact that all clients use exactly the same static IP address). The load balancer automatically checks the health of the pods comprising the service, and avoids sending requests to unresponsive or slow pods (or entire unresponsive clusters).

Federated Ingress is released as an alpha feature, and supports Google Cloud Platform (Google Kubernetes Engine, GCE and hybrid scenarios involving both) in Kubernetes v1.4. Work is under way to support other cloud providers such as AWS, and other hybrid cloud scenarios (e.g. services spanning private on-premises as well as public cloud Kubernetes clusters).

You create Federated Ingresses in much the same way as traditional Kubernetes Ingresses: by making an API call which specifies the desired properties of your logical ingress point. In the case of Federated Ingress, this API call is directed to the Federation API endpoint, rather than a Kubernetes cluster API endpoint. The API for Federated Ingress is 100% compatible with the API for traditional Kubernetes Services.

Once created, the Federated Ingress automatically:

- Creates matching Kubernetes Ingress objects in every cluster underlying your Cluster Federation
- Ensures that all of these in-cluster ingress objects share the same logical global L7 (that is, HTTP(S)) load balancer and IP address
- Monitors the health and capacity of the service shards (that is, your pods) behind this ingress in each cluster
- Ensures that all client connections are routed to an appropriate healthy backend service endpoint at all times, even in the event of pod, cluster, availability zone or regional outages

Note that in the case of Google Cloud, the logical L7 load balancer is not a single physical device (which would present both a single point of failure, and a single global network routing choke point), but rather a truly global, highly available load balancing managed service, globally reachable via a single, static IP address.

Clients inside your federated Kubernetes clusters (Pods) will be automatically routed to the cluster-local shard of the Federated Service backing the Ingress in their cluster if it exists and is healthy, or the closest healthy shard in a different cluster if it does not. Note that this involves a network trip to the HTTP(s) load balancer, which resides outside your local Kubernetes cluster but inside the same GCP region.

- Before you begin
- Creating a federated ingress
- Adding backend services and pods

- Hybrid cloud capabilities
- Discovering a federated ingress
- Handling failures of backend pods and whole clusters
- Troubleshooting
- What's next

Before you begin

This document assumes that you have a running Kubernetes Cluster Federation installation. If not, then see the federation admin guide to learn how to bring up a cluster federation (or have your cluster administrator do this for you). Other tutorials, for example this one by Kelsey Hightower, are also available to help you.

You must also have a basic working knowledge of Kubernetes in general, and Ingress in particular.

Creating a federated ingress

You can create a federated ingress in any of the usual ways, for example, using kubectl:

```
kubectl --context=federation-cluster create -f myingress.yaml
```

For example ingress YAML configurations, see the Ingress User Guide. The ‘--context=federation-cluster’ flag tells kubectl to submit the request to the Federation API endpoint, with the appropriate credentials. If you have not yet configured such a context, see the federation admin guide or one of the administration tutorials to find out how to do so.

The Federated Ingress automatically creates and maintains matching Kubernetes ingresses in all of the clusters underlying your federation. These cluster-specific ingresses (and their associated ingress controllers) configure and manage the load balancing and health checking infrastructure that ensures that traffic is load balanced to each cluster appropriately.

You can verify this by checking in each of the underlying clusters. For example:

```
kubectl --context=gce-asia-east1a get ingress myingress
NAME      HOSTS      ADDRESS          PORTS      AGE
myingress *          130.211.5.194    80, 443   1m
```

The above assumes that you have a context named ‘gce-asia-east1a’ configured in your client for your cluster in that zone. The name and namespace of the underlying ingress automatically matches those of the Federated Ingress that you created above (and if you happen to have had ingresses of the same name and namespace already existing in any of those clusters, they will be automatically

adopted by the Federation and updated to conform with the specification of your Federated Ingress. Either way, the end result will be the same).

The status of your Federated Ingress automatically reflects the real-time status of the underlying Kubernetes ingresses. For example:

```
kubectl --context=federation-cluster describe ingress myingress
```

```
Name:          myingress
Namespace:    default
Address:      130.211.5.194
TLS:
  tls-secret terminates
Rules:
  Host  Path      Backends
  ----  ----      -----
  * *  echoheaders-https:80 (10.152.1.3:8080,10.152.2.4:8080)
Annotations:
  https-target-proxy:      k8s-tps-default-myiningress--ff1107f83ed600c0
  target-proxy:            k8s-tp-default-myiningress--ff1107f83ed600c0
  url-map:                k8s-um-default-myiningress--ff1107f83ed600c0
  backends:               {"k8s-be-30301--ff1107f83ed600c0": "Unknown"}
  forwarding-rule:        k8s-fw-default-myiningress--ff1107f83ed600c0
  https-forwarding-rule:  k8s-fws-default-myiningress--ff1107f83ed600c0
Events:
  FirstSeen  LastSeen   Count  From           SubobjectPath  Type   Reason  Message
  -----  -----  -----  ----  -----  -----  -----  -----
  3m       3m        1  {loadbalancer-controller }  Normal  ADD  default/myingress
  2m       2m        1  {loadbalancer-controller }  Normal  CREATE ip: 130.211.5.194
```

Note that:

- The address of your Federated Ingress corresponds with the address of all of the underlying Kubernetes ingresses (once these have been allocated - this may take up to a few minutes).
- You have not yet provisioned any backend Pods to receive the network traffic directed to this ingress (that is, ‘Service Endpoints’ behind the service backing the Ingress), so the Federated Ingress does not yet consider these to be healthy shards and will not direct traffic to any of these clusters.
- The federation control system automatically reconfigures the load balancer controllers in all of the clusters in your federation to make them consistent, and allows them to share global load balancers. But this reconfiguration can only complete successfully if there are no pre-existing Ingresses in those clusters (this is a safety feature to prevent accidental breakage of existing ingresses). So, to ensure that your federated ingresses function correctly, either start with new, empty clusters, or make sure that you delete (and recreate if necessary) all pre-existing Ingresses in the clusters comprising your federation.

Adding backend services and pods

To render the underlying ingress shards healthy, you need to add backend Pods behind the service upon which the Ingress is based. There are several ways to achieve this, but the easiest is to create a Federated Service and Federated ReplicaSet. To create appropriately labelled pods and services in the 13 underlying clusters of your federation:

```
kubectl --context=federation-cluster create -f services/nginx.yaml  
kubectl --context=federation-cluster create -f myreplicaset.yaml
```

Note that in order for your federated ingress to work correctly on Google Cloud, the node ports of all of the underlying cluster-local services need to be identical. If you're using a federated service this is easy to do. Simply pick a node port that is not already being used in any of your clusters, and add that to the spec of your federated service. If you do not specify a node port for your federated service, each cluster will choose its own node port for its cluster-local shard of the service, and these will probably end up being different, which is not what you want.

You can verify this by checking in each of the underlying clusters. For example:

```
kubectl --context=gce-asia-east1a get services nginx  
NAME      CLUSTER-IP      EXTERNAL-IP      PORT(S)      AGE  
nginx     10.63.250.98    104.199.136.89   80/TCP      9m
```

Hybrid cloud capabilities

Federations of Kubernetes Clusters can include clusters running in different cloud providers (for example, Google Cloud, AWS), and on-premises (for example, on OpenStack). However, in Kubernetes v1.4, Federated Ingress is only supported across Google Cloud clusters.

Discovering a federated ingress

Ingress objects (in both plain Kubernetes clusters, and in federations of clusters) expose one or more IP addresses (via the Status.Loadbalancer.Ingress field) that remains static for the lifetime of the Ingress object (in future, automatically managed DNS names might also be added). All clients (whether internal to your cluster, or on the external network or internet) should connect to one of these IP or DNS addresses. All client requests are automatically routed, via the shortest network path, to a healthy pod in the closest cluster to the origin of the request. So for example, HTTP(S) requests from internet users in Europe will be routed directly to the closest cluster in Europe that has available capacity.

If there are no such clusters in Europe, the request will be routed to the next closest cluster (typically in the U.S.).

Handling failures of backend pods and whole clusters

Ingresses are backed by Services, which are typically (but not always) backed by one or more ReplicaSets. For Federated Ingresses, it is common practise to use the federated variants of Services and ReplicaSets for this purpose.

In particular, Federated ReplicaSets ensure that the desired number of pods are kept running in each cluster, even in the event of node failures. In the event of entire cluster or availability zone failures, Federated ReplicaSets automatically place additional replicas in the other available clusters in the federation to accommodate the traffic which was previously being served by the now unavailable cluster. While the Federated ReplicaSet ensures that sufficient replicas are kept running, the Federated Ingress ensures that user traffic is automatically redirected away from the failed cluster to other available clusters.

Troubleshooting

I cannot connect to my cluster federation API.

Check that your:

1. Client (typically `kubectl`) is correctly configured (including API endpoints and login credentials).
2. Cluster Federation API server is running and network-reachable.

See the federation admin guide to learn how to bring up a cluster federation correctly (or have your cluster administrator do this for you), and how to correctly configure your client.

I can create a Federated Ingress/service/replicaset successfully against the cluster federation API, but no matching ingress/services/replicaset are created in my underlying clusters.

Check that:

1. Your clusters are correctly registered in the Cluster Federation API. (`kubectl describe clusters`)
2. Your clusters are all ‘Active’. This means that the cluster Federation system was able to connect and authenticate against the clusters’ endpoints. If not, consult the event logs of the federation-controller-manager pod to ascertain what the failure might be. (`kubectl --namespace=federation logs $(kubectl get pods`

- ```
--namespace=federation -l module=federation-controller-manager
-o name)
```
3. That the login credentials provided to the Cluster Federation API for the clusters have the correct authorization and quota to create ingresses/services/replicaset in the relevant namespace in the clusters. Again you should see associated error messages providing more detail in the above event log file if this is not the case.
  4. Whether any other error is preventing the service creation operation from succeeding (look for `ingress-controller`, `service-controller` or `replicaset-controller`, errors in the output of `kubectl logs federation-controller-manager --namespace federation`).

**I can create a federated ingress successfully, but request load is not correctly distributed across the underlying clusters.**

Check that:

1. The services underlying your federated ingress in each cluster have identical node ports. See above for further explanation.
2. The load balancer controllers in each of your clusters are of the correct type (“GLBC”) and have been correctly reconfigured by the federation control plane to share a global GCE load balancer (this should happen automatically). If they are of the correct type, and have been correctly reconfigured, the UID data item in the GLBC configmap in each cluster will be identical across all clusters. See the GLBC docs for further details. If this is not the case, check the logs of your federation controller manager to determine why this automated reconfiguration might be failing.
3. No ingresses have been manually created in any of your clusters before the above reconfiguration of the load balancer controller completed successfully. Ingresses created before the reconfiguration of your GLBC will interfere with the behavior of your federated ingresses created after the reconfiguration (see the GLBC docs for further information). To remedy this, delete any ingresses created before the cluster joined the federation (and had its GLBC reconfigured), and recreate them if necessary.

## What's next

- If you need assistance, use one of the support channels to seek assistance.
  - For details about use cases that motivated this work, see Federation proposal.

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

[Edit This Page](#)

## Federated Jobs

**Note:** Federation V1, the current Kubernetes federation API which reuses the Kubernetes API resources ‘as is’, is currently considered alpha for many of its features, and there is no clear path to evolve the API to GA. However, there is a **Federation V2** effort in progress to implement a dedicated federation API apart from the Kubernetes API. The details can be found at sig-multicloud community page.

This guide explains how to use jobs in the federation control plane.

Jobs in the federation control plane (referred to as “federated jobs” in this guide) are similar to the traditional Kubernetes jobs, and provide the same functionality. Creating jobs in the federation control plane ensures that the desired number of parallelism and completions exist across the registered clusters.

- Before you begin
- Creating a federated job
- Updating a federated job
- Deleting a federated job

### Before you begin

- This guide assumes that you have a running Kubernetes Cluster Federation installation. If not, then head over to the federation admin guide to learn how to bring up a cluster federation (or have your cluster administrator do this for you). Other tutorials, such as Kelsey Hightower’s Federated Kubernetes Tutorial, might also help you create a Federated Kubernetes cluster.
- You are also expected to have a basic working knowledge of Kubernetes in general and jobs in particular.

### Creating a federated job

The API for federated jobs is fully compatible with the API for traditional Kubernetes jobs. You can create a job by sending a request to the federation apiserver.

You can do that using kubectl by running:

```
kubectl --context=federation-cluster create -f myjob.yaml
```

The ‘--context=federation-cluster’ flag tells kubectl to submit the request to the federation API server instead of sending it to a Kubernetes cluster.

Once a federated job is created, the federation control plane creates a job in all underlying Kubernetes clusters. You can verify this by checking each of the underlying clusters, for example:

```
kubectl --context=gce-asia-east1a get job myjob
```

The previous example assumes that you have a context named `gce-asia-east1a` configured in your client for your cluster in that zone.

The jobs in the underlying clusters match the federated job except in the number of parallelism and completions. The federation control plane ensures that the sum of the parallelism and completions in each cluster matches the desired number of parallelism and completions in the federated job.

### Spreading job tasks in underlying clusters

By default, parallelism and completions are spread equally in all underlying clusters. For example: if you have 3 registered clusters and you create a federated job with `spec.parallelism = 9` and `spec.completions = 18`, then each job in the 3 clusters has `spec.parallelism = 3` and `spec.completions = 6`. To modify the number of parallelism and completions in each cluster, you can specify `ReplicaAllocationPreferences` as an annotation with key `federation.kubernetes.io/job-preferences` on the federated job.

### Updating a federated job

You can update a federated job as you would update a Kubernetes job; however, for a federated job, you must send the request to the federation API server instead of sending it to a specific Kubernetes cluster. The federation control plane ensures that whenever the federated job is updated, it updates the corresponding job in all underlying clusters to match it.

If your update includes a change in number of parallelism and completions, the federation control plane changes the number of parallelism and completions in underlying clusters to ensure that their sum remains equal to the number of desired parallelism and completions in federated job.

### Deleting a federated job

You can delete a federated job as you would delete a Kubernetes job; however, for a federated job, you must send the request to the federation API server instead of sending it to a specific Kubernetes cluster.

For example, with kubectl:

```
kubectl --context=federation-cluster delete job myjob
```

**Note:** Deleting a federated job will not delete the corresponding jobs from underlying clusters. You must delete the underlying jobs manually.

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

[Edit This Page](#)

## Federated Namespaces

**Note:** **Federation V1**, the current Kubernetes federation API which reuses the Kubernetes API resources ‘as is’, is currently considered alpha for many of its features, and there is no clear path to evolve the API to GA. However, there is a **Federation V2** effort in progress to implement a dedicated federation API apart from the Kubernetes API. The details can be found at [sig-multicloud community page](#).

This guide explains how to use Namespaces in Federation control plane.

Namespaces in federation control plane (referred to as “federated Namespaces” in this guide) are very similar to the traditional Kubernetes Namespaces providing the same functionality. Creating them in the federation control plane ensures that they are synchronized across all the clusters in federation.

- Before you begin
- Creating a Federated Namespace
- Updating a Federated Namespace
- Deleting a Federated Namespace

### Before you begin

- This guide assumes that you have a running Kubernetes Cluster Federation installation. If not, then head over to the federation admin guide to learn how to bring up a cluster federation (or have your cluster administrator do this for you). Other tutorials, such as Kelsey Hightower’s Federated Kubernetes Tutorial, might also help you create a Federated Kubernetes cluster.
- You are also expected to have a basic working knowledge of Kubernetes in general and Namespaces in particular.

## **Creating a Federated Namespace**

The API for Federated Namespaces is 100% compatible with the API for traditional Kubernetes Namespaces. You can create a Namespace by sending a request to the federation apiserver.

You can do that using kubectl by running:

```
kubectl --context=federation-cluster create -f myns.yaml
```

The ‘--context=federation-cluster’ flag tells kubectl to submit the request to the Federation apiserver instead of sending it to a Kubernetes cluster.

Once a federated Namespace is created, the federation control plane will create a matching Namespace in all underlying Kubernetes clusters. You can verify this by checking each of the underlying clusters, for example:

```
kubectl --context=gce-asia-east1a get namespaces myns
```

The above assumes that you have a context named ‘gce-asia-east1a’ configured in your client for your cluster in that zone. The name and spec of the underlying Namespace will match those of the Federated Namespace that you created above.

## **Updating a Federated Namespace**

You can update a federated Namespace as you would update a Kubernetes Namespace, just send the request to federation apiserver instead of sending it to a specific Kubernetes cluster. Federation control plan will ensure that whenever the federated Namespace is updated, it updates the corresponding Namespaces in all underlying clusters to match it.

## **Deleting a Federated Namespace**

You can delete a federated Namespace as you would delete a Kubernetes Namespace, just send the request to federation apiserver instead of sending it to a specific Kubernetes cluster.

For example, you can do that using kubectl by running:

```
kubectl --context=federation-cluster delete ns myns
```

As in Kubernetes, deleting a federated Namespace will delete all resources in that Namespace from the federation control plane.

Note that at this point, deleting a federated Namespace will not delete the corresponding Namespace and resources in those Namespaces from underlying clusters. Users are expected to delete them manually. We intend to fix this in the future.

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

[Edit This Page](#)

## Federated ReplicaSets

**Note:** **Federation V1**, the current Kubernetes federation API which reuses the Kubernetes API resources ‘as is’, is currently considered alpha for many of its features, and there is no clear path to evolve the API to GA. However, there is a **Federation V2** effort in progress to implement a dedicated federation API apart from the Kubernetes API. The details can be found at [sig-multicloud community page](#).

This guide explains how to use ReplicaSets in the Federation control plane.

ReplicaSets in the federation control plane (referred to as “federated ReplicaSets” in this guide) are very similar to the traditional Kubernetes ReplicaSets, and provide the same functionality. Creating them in the federation control plane ensures that the desired number of replicas exist across the registered clusters.

- Before you begin
- Creating a Federated ReplicaSet
- Updating a Federated ReplicaSet
- Deleting a Federated ReplicaSet

### Before you begin

- This guide assumes that you have a running Kubernetes Cluster Federation installation. If not, then head over to the federation admin guide to learn how to bring up a cluster federation (or have your cluster administrator do this for you). Other tutorials, such as Kelsey Hightower’s Federated Kubernetes Tutorial, might also help you create a Federated Kubernetes cluster.
- You are also expected to have a basic working knowledge of Kubernetes in general and ReplicaSets in particular.

### Creating a Federated ReplicaSet

The API for Federated ReplicaSet is 100% compatible with the API for traditional Kubernetes ReplicaSet. You can create a ReplicaSet by sending a request to the federation apiserver.

You can do that using kubectl by running:

```
kubectl --context=federation-cluster create -f myrs.yaml
```

The `--context=federation-cluster` flag tells kubectl to submit the request to the Federation apiserver instead of sending it to a Kubernetes cluster.

Once a federated ReplicaSet is created, the federation control plane will create a ReplicaSet in all underlying Kubernetes clusters. You can verify this by checking each of the underlying clusters, for example:

```
kubectl --context=gce-asia-east1a get rs myrs
```

The above assumes that you have a context named ‘gce-asia-east1a’ configured in your client for your cluster in that zone.

The ReplicaSets in the underlying clusters will match the federation ReplicaSet except in the number of replicas. The federation control plane will ensure that the sum of the replicas in each cluster match the desired number of replicas in the federation ReplicaSet.

## Spreading Replicas in Underlying Clusters

By default, replicas are spread equally in all the underlying clusters. For example: if you have 3 registered clusters and you create a federated ReplicaSet with `spec.replicas = 9`, then each ReplicaSet in the 3 clusters will have `spec.replicas=3`. To modify the number of replicas in each cluster, you can add an annotation with key `federation.kubernetes.io/replica-set-preferences` to the federated ReplicaSet. The value of the annotation is a serialized JSON that contains fields shown in the following example:

```
{
 "rebalance": true,
 "clusters": {
 "foo": {
 "minReplicas": 10,
 "maxReplicas": 50,
 "weight": 100
 },
 "bar": {
 "minReplicas": 10,
 "maxReplicas": 100,
 "weight": 200
 }
 }
}
```

The `rebalance` boolean field specifies whether replicas already scheduled and running may be moved in order to match current state to the specified preferences. The `clusters` object field contains a map where users can specify the

constraints for replica placement across the clusters (`foo` and `bar` in the example). For each cluster, you can specify the minimum number of replicas that should be assigned to it (default is zero), the maximum number of replicas the cluster can accept (default is unbounded) and a number expressing the relative weight of preferences to place additional replicas to that cluster.

## Updating a Federated ReplicaSet

You can update a federated ReplicaSet as you would update a Kubernetes ReplicaSet; however, for a federated ReplicaSet, you must send the request to the federation apiserver instead of sending it to a specific Kubernetes cluster. The Federation control plane ensures that whenever the federated ReplicaSet is updated, it updates the corresponding ReplicaSet in all underlying clusters to match it. If your update includes a change in number of replicas, the federation control plane will change the number of replicas in underlying clusters to ensure that their sum remains equal to the number of desired replicas in federated ReplicaSet.

## Deleting a Federated ReplicaSet

You can delete a federated ReplicaSet as you would delete a Kubernetes ReplicaSet; however, for a federated ReplicaSet, you must send the request to the federation apiserver instead of sending it to a specific Kubernetes cluster.

For example, you can do that using kubectl by running:

```
kubectl --context=federation-cluster delete rs myrs
```

Note that at this point, deleting a federated ReplicaSet will not delete the corresponding ReplicaSets from underlying clusters. You must delete the underlying ReplicaSets manually. We intend to fix this in the future.

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

[Edit This Page](#)

## Federated Secrets

**Note: Federation V1**, the current Kubernetes federation API which reuses the Kubernetes API resources ‘as is’, is currently considered alpha for many of its features, and there is no clear path to evolve the API to GA. However, there is a **Federation V2** effort in progress to implement a dedicated federation API

apart from the Kubernetes API. The details can be found at sig-multicluster community page.

This guide explains how to use secrets in Federation control plane.

Secrets in federation control plane (referred to as “federated secrets” in this guide) are very similar to the traditional Kubernetes Secrets providing the same functionality. Creating them in the federation control plane ensures that they are synchronized across all the clusters in federation.

- Prerequisites
- Creating a Federated Secret
- Updating a Federated Secret
- Deleting a Federated Secret

## Prerequisites

This guide assumes that you have a running Kubernetes Cluster Federation installation. If not, then head over to the federation admin guide to learn how to bring up a cluster federation (or have your cluster administrator do this for you). Other tutorials, for example this one by Kelsey Hightower, are also available to help you.

You are also expected to have a basic working knowledge of Kubernetes in general and Secrets in particular.

## Creating a Federated Secret

The API for Federated Secret is 100% compatible with the API for traditional Kubernetes Secret. You can create a secret by sending a request to the federation apiserver.

You can do that using kubectl by running:

```
kubectl --context=federation-cluster create -f mysecret.yaml
```

The `--context=federation-cluster` flag tells kubectl to submit the request to the Federation apiserver instead of sending it to a Kubernetes cluster.

Once a federated secret is created, the federation control plane will create a matching secret in all underlying Kubernetes clusters. You can verify this by checking each of the underlying clusters, for example:

```
kubectl --context=gce-asia-east1a get secret mysecret
```

The above assumes that you have a context named ‘gce-asia-east1a’ configured in your client for your cluster in that zone.

These secrets in underlying clusters will match the federated secret.

## Updating a Federated Secret

You can update a federated secret as you would update a Kubernetes secret; however, for a federated secret, you must send the request to the federation apiserver instead of sending it to a specific Kubernetes cluster. The Federation control plan ensures that whenever the federated secret is updated, it updates the corresponding secrets in all underlying clusters to match it.

## Deleting a Federated Secret

You can delete a federated secret as you would delete a Kubernetes secret; however, for a federated secret, you must send the request to the federation apiserver instead of sending it to a specific Kubernetes cluster.

For example, you can do that using `kubectl` by running:

```
kubectl --context=federation-cluster delete secret mysecret
```

Note that at this point, deleting a federated secret will not delete the corresponding secrets from underlying clusters. You must delete the underlying secrets manually. We intend to fix this in the future.

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

[Edit This Page](#)

## Extend `kubectl` with plugins

**FEATURE STATE:** Kubernetes v1.10 alpha

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

- The version names contain alpha (e.g. v1alpha1).
- Might be buggy. Enabling the feature may expose bugs. Disabled by default.
- Support for feature may be dropped at any time without notice.
- The API may change in incompatible ways in a later software release without notice.
- Recommended for use only in short-lived testing clusters, due to increased risk of bugs and lack of long-term support.

This guide shows you how to install and write extensions for `kubectl`. Usually called *plugins* or *binary extensions*, this feature allows you to extend the default set of commands available in `kubectl` by adding new subcommands to perform new tasks and extend the set of features available in the main distribution of `kubectl`.

- Before you begin
- Installing kubectl plugins
- Writing kubectl plugins
- What's next

## Before you begin

You need to have a working `kubectl` binary installed. Note that plugins were officially introduced as an alpha feature in the v1.8.0 release. So, while some parts of the plugins feature were already available in previous versions, a `kubectl` version of 1.8.0 or later is recommended.

Until a GA version is released, plugins will only be available under the `kubectl plugin` subcommand.

## Installing kubectl plugins

A plugin is nothing more than a set of files: at least a `plugin.yaml` descriptor, and likely one or more binary, script, or assets files. To install a plugin, copy those files to one of the locations in the filesystem where `kubectl` searches for plugins.

Note that Kubernetes does not provide a package manager or something similar to install or update plugins, so it's your responsibility to place the plugin files in the correct location. We recommend that each plugin is located on its own directory, so installing a plugin that is distributed as a compressed file is as simple as extracting it to one of the locations specified in the Plugin loader section.

## Plugin loader

The plugin loader is responsible for searching plugin files in the filesystem locations specified below, and checking if the plugin provides the minimum amount of information required for it to run. Files placed in the right location that don't provide the minimum amount of information, for example an incomplete `plugin.yaml` descriptor, are ignored.

## Search order

The plugin loader uses the following search order:

1.  `${KUBECTL_PLUGINS_PATH}` If specified, the search stops here.
2.  `${XDG_DATA_DIRS}/kubectl/plugins`
3.  `~/.kube/plugins`

If the `KUBECTL_PLUGINS_PATH` environment variable is present, the loader uses it as the only location to look for plugins. The `KUBECTL_PLUGINS_PATH` environment variable is a list of directories. In Linux and Mac, the list is colon-delimited. In Windows, the list is semicolon-delimited.

If `KUBECTL_PLUGINS_PATH` is not present, the loader searches these additional locations:

First, one or more directories specified according to the XDG System Directory Structure specification. Specifically, the loader locates the directories specified by the `XDG_DATA_DIRS` environment variable, and then searches `kubectl/plugins` directory inside of those. If `XDG_DATA_DIRS` is not specified, it defaults to `/usr/local/share:/usr/share`.

Second, the `plugins` directory under the user's `kubeconfig` dir. In most cases, this is `~/.kube/plugins`.

```
Loads plugins from both /path/to/dir1 and /path/to/dir2
KUBECTL_PLUGINS_PATH=/path/to/dir1:/path/to/dir2 kubectl plugin -h
```

## Writing kubectl plugins

You can write a plugin in any programming language or script that allows you to write command-line commands. A plugin does not necessarily need to have a binary component. It could rely entirely on operating system utilities like `echo`, `sed`, or `grep`. Or it could rely on the `kubectl` binary.

The only strong requirement for a `kubectl` plugin is the `plugin.yaml` descriptor file. This file is responsible for declaring at least the minimum attributes required to register a plugin and must be located under one of the locations specified in the Search order section.

### The `plugin.yaml` descriptor

The descriptor file supports the following attributes:

```
name: "targaryen" # REQUIRED: the plugin command name, to be invoked under
shortDesc: "Dragonized plugin" # REQUIRED: the command short description, for help
longDesc: "" # the command long description, for help
example: "" # command example(s), for help
command: "./dracarys" # REQUIRED: the command, binary, or script to invoke when
flags: # flags supported by the plugin
 - name: "heat" # REQUIRED for each flag: flag name
 shorthand: "h" # short version of the flag name
 desc: "Fire heat" # REQUIRED for each flag: flag description
 defValue: "extreme" # default value of the flag
tree: # allows the declaration of subcommands
```

```
- ... # subcommands support the same set of attributes
```

The preceding descriptor declares the `kubectl plugin targaryen` plugin, which has one flag named `-h | --heat`. When the plugin is invoked, it calls the `dracarys` binary or script, which is located in the same directory as the descriptor file. The Accessing runtime attributes section describes how the `dracarys` command accesses the flag value and other runtime context.

### Recommended directory structure

It is recommended that each plugin has its own subdirectory in the filesystem, preferably with the same name as the plugin command. The directory must contain the `plugin.yaml` descriptor and any binary, script, asset, or other dependency it might require.

For example, the directory structure for the `targaryen` plugin could look like this:

```
~/.kube/plugins/
 targaryen
 plugin.yaml
 dracarys
```

### Accessing runtime attributes

In most use cases, the binary or script file you write to support the plugin must have access to some contextual information provided by the plugin framework. For example, if you declared flags in the descriptor file, your plugin must have access to the user-provided flag values at runtime. The same is true for global flags. The plugin framework is responsible for doing that, so plugin writers don't need to worry about parsing arguments. This also ensures the best level of consistency between plugins and regular `kubectl` commands.

Plugins have access to runtime context attributes through environment variables. So to access the value provided through a flag, for example, just look for the value of the proper environment variable using the appropriate function call for your binary or script.

The supported environment variables are:

- `KUBECTL_PLUGINS_CALLER`: The full path to the `kubectl` binary that was used in the current command invocation. As a plugin writer, you don't have to implement logic to authenticate and access the Kubernetes API. Instead, you can invoke `kubectl` to obtain the information you need, through something like `kubectl get --raw=/apis`.

- `KUBECTL_PLUGINS_CURRENT_NAMESPACE`: The current namespace that is the context for this call. This is the actual namespace to be used, meaning it was already processed in terms of the precedence between what was provided through the kubeconfig, the `--namespace` global flag, environment variables, and so on.
- `KUBECTL_PLUGINS_DESCRIPTOR_*`: One environment variable for every attribute declared in the `plugin.yaml` descriptor. For example, `KUBECTL_PLUGINS_DESCRIPTOR_NAME`, `KUBECTL_PLUGINS_DESCRIPTOR_COMMAND`.
- `KUBECTL_PLUGINS_GLOBAL_FLAG_*`: One environment variable for every global flag supported by kubectl. For example, `KUBECTL_PLUGINS_GLOBAL_FLAG_NAMESPACE`, `KUBECTL_PLUGINS_GLOBAL_FLAG_V`.
- `KUBECTL_PLUGINS_LOCAL_FLAG_*`: One environment variable for every local flag declared in the `plugin.yaml` descriptor. For example, `KUBECTL_PLUGINS_LOCAL_FLAG_HEAT` in the preceding `targaryen` example.

## What's next

- Check the repository for some more examples of plugins.
- In case of any questions, feel free to reach out to the CLI SIG team.
- Binary plugins is still an alpha feature, so this is the time to contribute ideas and improvements to the codebase. We're also excited to hear about what you're planning to implement with plugins, so let us know!

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

[Edit This Page](#)

## Manage HugePages

**FEATURE STATE:** Kubernetes v1.10 beta

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

- The version names contain beta (e.g. v2beta3).
- Code is well tested. Enabling the feature is considered safe. Enabled by default.
- Support for the overall feature will not be dropped, though details may change.
- The schema and/or semantics of objects may change in incompatible ways in a subsequent beta or stable release. When this happens, we will provide instructions for migrating to the next version. This may require deleting,

editing, and re-creating API objects. The editing process may require some thought. This may require downtime for applications that rely on the feature.

- Recommended for only non-business-critical uses because of potential for incompatible changes in subsequent releases. If you have multiple clusters that can be upgraded independently, you may be able to relax this restriction.
- **Please do try our beta features and give feedback on them! After they exit beta, it may not be practical for us to make more changes.**

Kubernetes supports the allocation and consumption of pre-allocated huge pages by applications in a Pod as a **beta** feature. This page describes how users can consume huge pages and the current limitations.

- Before you begin
- API
- Future

## Before you begin

1. Kubernetes nodes must pre-allocate huge pages in order for the node to report its huge page capacity. A node may only pre-allocate huge pages for a single size.

The nodes will automatically discover and report all huge page resources as a schedulable resource.

## API

Huge pages can be consumed via container level resource requirements using the resource name `hugepages-<size>`, where size is the most compact binary notation using integer values supported on a particular node. For example, if a node supports 2048KiB page sizes, it will expose a schedulable resource `hugepages-2Mi`. Unlike CPU or memory, huge pages do not support overcommit.

```
apiVersion: v1
kind: Pod
metadata:
 generateName: hugepages-volume-
spec:
 containers:
 - image: fedora:latest
 command:
 - sleep
```

```

- inf
name: example
volumeMounts:
- mountPath: /hugepages
 name: hugepage
resources:
 limits:
 hugepages-2Mi: 100Mi
volumes:
- name: hugepage
 emptyDir:
 medium: HugePages

```

- Huge page requests must equal the limits. This is the default if limits are specified, but requests are not.
- Huge pages are isolated at a pod scope, container isolation is planned in a future iteration.
- EmptyDir volumes backed by huge pages may not consume more huge page memory than the pod request.
- Applications that consume huge pages via `shmget()` with `SHM_HUGETLB` must run with a supplemental group that matches `proc/sys/vm/hugetlb_shm_group`.
- Huge page usage in a namespace is controllable via `ResourceQuota` similar to other compute resources like `cpu` or `memory` using the `hugepages-<size>` token.

## Future

- Support container isolation of huge pages in addition to pod isolation.
- NUMA locality guarantees as a feature of quality of service.
- LimitRange support.

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

[Edit This Page](#)

## Schedule GPUs

Kubernetes includes **experimental** support for managing NVIDIA GPUs spread across nodes. The support for NVIDIA GPUs was added in v1.6 and has gone through multiple backwards incompatible iterations. This page describes how users can consume GPUs across different Kubernetes versions and the current limitations.

- v1.8 onwards

- Clusters containing different types of NVIDIA GPUs
- v1.6 and v1.7
- Future

## v1.8 onwards

**From 1.8 onwards, the recommended way to consume GPUs is to use device plugins.**

To enable GPU support through device plugins before 1.10, the `DevicePlugins` feature gate has to be explicitly set to true across the system: `--feature-gates="DevicePlugins=true"`. This is no longer required starting from 1.10.

Then you have to install NVIDIA drivers on the nodes and run an NVIDIA GPU device plugin (see below).

When the above conditions are true, Kubernetes will expose `nvidia.com/gpu` as a schedulable resource.

You can consume these GPUs from your containers by requesting `nvidia.com/gpu` just like you request `cpu` or `memory`. However, there are some limitations in how you specify the resource requirements when using GPUs: - GPUs are only supposed to be specified in the `limits` section, which means: \* You can specify GPU `limits` without specifying `requests` because Kubernetes will use the limit as the request value by default. \* You can specify GPU in both `limits` and `requests` but these two values must be equal. \* You cannot specify GPU `requests` without specifying `limits`. - Containers (and pods) do not share GPUs. There's no overcommitting of GPUs. - Each container can request one or more GPUs. It is not possible to request a fraction of a GPU.

Here's an example:

```
apiVersion: v1
kind: Pod
metadata:
 name: cuda-vector-add
spec:
 restartPolicy: OnFailure
 containers:
 - name: cuda-vector-add
 # https://github.com/kubernetes/kubernetes/blob/v1.7.11/test/images/nvidia-cuda/Dockerfile
 image: "k8s.gcr.io/cuda-vector-add:v0.1"
 resources:
 limits:
 nvidia.com/gpu: 1 # requesting 1 GPU
```

## Deploying NVIDIA GPU device plugin

There are currently two device plugin implementations for NVIDIA GPUs:

### Official NVIDIA GPU device plugin

The official NVIDIA GPU device plugin has the following requirements: - Kubernetes nodes have to be pre-installed with NVIDIA drivers. - Kubernetes nodes have to be pre-installed with nvidia-docker 2.0 - nvidia-container-runtime must be configured as the default runtime for docker instead of runc. - NVIDIA drivers  $\approx$  361.93

To deploy the NVIDIA device plugin once your cluster is running and the above requirements are satisfied:

```
For Kubernetes v1.8
kubectl create -f https://raw.githubusercontent.com/NVIDIA/k8s-device-plugin/v1.8/nvidia-device-plugin.yaml

For Kubernetes v1.9
kubectl create -f https://raw.githubusercontent.com/NVIDIA/k8s-device-plugin/v1.9/nvidia-device-plugin.yaml
```

Report issues with this device plugin to [NVIDIA/k8s-device-plugin](https://github.com/NVIDIA/k8s-device-plugin).

### NVIDIA GPU device plugin used by GKE/GCE

The NVIDIA GPU device plugin used by GKE/GCE doesn't require using nvidia-docker and should work with any container runtime that is compatible with the Kubernetes Container Runtime Interface (CRI). It's tested on Container-Optimized OS and has experimental code for Ubuntu from 1.9 onwards.

On your 1.9 cluster, you can use the following commands to install the NVIDIA drivers and device plugin:

```
Install NVIDIA drivers on Container-Optimized OS:
kubectl create -f https://raw.githubusercontent.com/GoogleCloudPlatform/container-engine-accelerators/1.9.0-alpha.1/nvidia-driver/nvidia-device-plugin.yaml

Install NVIDIA drivers on Ubuntu (experimental):
kubectl create -f https://raw.githubusercontent.com/GoogleCloudPlatform/container-engine-accelerators/1.9.0-alpha.1/ubuntu/nvidia-device-plugin.yaml

Install the device plugin:
kubectl create -f https://raw.githubusercontent.com/kubernetes/kubernetes/release-1.9/cluster/addons/nvidia-device-plugin/nvidia-device-plugin.yaml
```

Report issues with this device plugin and installation method to [GoogleCloudPlatform/container-engine-accelerators](https://github.com/GoogleCloudPlatform/container-engine-accelerators).

## Clusters containing different types of NVIDIA GPUs

If different nodes in your cluster have different types of NVIDIA GPUs, then you can use Node Labels and Node Selectors to schedule pods to appropriate nodes.

For example:

```
Label your nodes with the accelerator type they have.
kubectl label nodes <node-with-k80> accelerator=nvidia-tesla-k80
kubectl label nodes <node-with-p100> accelerator=nvidia-tesla-p100
```

Specify the GPU type in the pod spec:

```
apiVersion: v1
kind: Pod
metadata:
 name: cuda-vector-add
spec:
 restartPolicy: OnFailure
 containers:
 - name: cuda-vector-add
 # https://github.com/kubernetes/kubernetes/blob/v1.7.11/test/images/nvidia-cuda/Dockerfile
 image: "k8s.gcr.io/cuda-vector-add:v0.1"
 resources:
 limits:
 nvidia.com/gpu: 1
 nodeSelector:
 accelerator: nvidia-tesla-p100 # or nvidia-tesla-k80 etc.
```

This will ensure that the pod will be scheduled to a node that has the GPU type you specified.

## v1.6 and v1.7

To enable GPU support in 1.6 and 1.7, a special **alpha** feature gate **Accelerators** has to be set to true across the system: `--feature-gates="Accelerators=true"`. It also requires using the Docker Engine as the container runtime.

Further, the Kubernetes nodes have to be pre-installed with NVIDIA drivers. Kubelet will not detect NVIDIA GPUs otherwise.

When you start Kubernetes components after all the above conditions are true, Kubernetes will expose `alpha.kubernetes.io/nvidia-gpu` as a schedulable resource.

You can consume these GPUs from your containers by requesting `alpha.kubernetes.io/nvidia-gpu` just like you request `cpu` or `memory`. However, there are some limitations in how you specify the resource requirements when using GPUs: - GPUs are only

supposed to be specified in the `limits` section, which means:

- \* You can specify GPU `limits` without specifying `requests` because Kubernetes will use the limit as the request value by default.
- \* You can specify GPU in both `limits` and `requests` but these two values must be equal.
- \* You cannot specify GPU `requests` without specifying `limits`.

- Containers (and pods) do not share GPUs. There's no overcommitting of GPUs.
- Each container can request one or more GPUs. It is not possible to request a fraction of a GPU.

When using `alpha.kubernetes.io/nvidia-gpu` as the resource, you also have to mount host directories containing NVIDIA libraries (`libcuda.so`, `libnvidia.so` etc.) to the container.

Here's an example:

```
apiVersion: v1
kind: Pod
metadata:
 name: cuda-vector-add
spec:
 restartPolicy: OnFailure
 containers:
 - name: cuda-vector-add
 # https://github.com/kubernetes/kubernetes/blob/v1.7.11/test/images/nvidia-cuda/Dockerfile
 image: "k8s.gcr.io/cuda-vector-add:v0.1"
 resources:
 limits:
 alpha.kubernetes.io/nvidia-gpu: 1 # requesting 1 GPU
 volumeMounts:
 - name: "nvidia-libraries"
 mountPath: "/usr/local/nvidia/lib64"
 volumes:
 - name: "nvidia-libraries"
 hostPath:
 path: "/usr/lib/nvidia-375"
```

The `Accelerators` feature gate and `alpha.kubernetes.io/nvidia-gpu` resource works on 1.8 and 1.9 as well. It will be deprecated in 1.10 and removed in 1.11.

## Future

- Support for hardware accelerators in Kubernetes is still in alpha.
- Better APIs will be introduced to provision and consume accelerators in a scalable manner.
- Kubernetes will automatically ensure that applications consuming GPUs get the best possible performance.

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